



# **Intel® Integrated Performance Primitives Reference Manual: Volume 1**

***November 2009***

World Wide Web: <http://www.intel.com>

Disclaimer and Legal Information

Document Number: A24968-026US



<b>Version</b>	<b>Version Information</b>	<b>Date</b>
-1001	Original Issue	09/2000
-1002	Documents Intel IPP 1.0 final release. Functions NormDiff, AutoCorr, and ZeroMean have been added. Derivatives Functions section have been revised.	02/2001
-1101	Documents Intel IPP 1.1 beta release.	04/2001
-2001	Documents Intel IPP 2.0 beta release. General audio coding, MP3, and transcendental vector functions have been added. Speech recognition API have been revised.	08/2001
-2002	Documents Intel IPP 2.0 release. New Intel IPP common functions have been added. The set of arithmetic, vector initialization, statistical, and filtering functions have been expanded.	11/2001
-3001	Documents Intel IPP 3.0 pre-beta. New speech codec functions have been added. The set of conversion and windowing functions have been expanded. Speech recognition API have been updated.	04/2002
-3002	Documents Intel IPP 3.0 beta release.	06/2002
-3003	Documents Intel IPP 3.0 beta update.	09/2002
-3004	Documents Intel IPP 3.0 release.	11/2002
-4001	Documents Intel IPP 4.0 beta. New functions for cross-architecture development have been added.	05/2003
-4002	Documents Intel IPP 4.0 release.	10/2003
-012	Documents Intel IPP 4.1 beta release.	04/2004
-013	Documents Intel IPP 4.1 release. Added new statistical functions and flavors for arithmetic and filtering functions.	07/2004
-014	Documents Intel IPP 5.0 beta release. New data compression functions and functions for regular expressions have been added. The set of arithmetic, conversion, transform, and filtering functions have been expanded.	03/2005
-015	Documents Intel 5.0 release. Added new data compression functions and flavors for vector initialization and Fourier transform functions.	08/2005
-016	Documents Intel IPP 5.1 beta release.	10/2005
-017	Documents Intel IPP 5.1 release. Added new flavors for arithmetic functions.	02/2006
-018	Documents Intel IPP 5.2 beta release. Added new data compression functions for bzip2 compatibility, new functions for the Extended AMR Wideband (AMRWB+) Speech Codec, vector conversion and initialization. Added new flavors for arithmetic functions.	09/2006
-019	Documents Intel IPP 5.2 release. Added descriptions of new conversion and statistical functions, SBR Encoder functions for audio coding. Added more new code examples for different functions.	01/2007

Version	Version Information	Date
-020	Documents Intel IPP 5.3 beta release. Added descriptions of new speech coding functions (EC subband DT controller), new flavors of Goertz functions. Added new code examples.	06/2007
-021	Documents Intel IPP 5.3 release. Added descriptions of new speech coding functions (RT Audio), new arithmetic functions, new section on DTS Audio Coding.	09/2007
-022	Documents Intel IPP 6.0 beta release. Added descriptions of new speech coding functions (G.729.1 codec, voice enhancement functions), new data compression (ZLIB, LZO coding), fixed-accuracy arithmetic, filtering, data integrity functions, new generated functions for transforms with a fixed length.	02/2008
-023	Documents Intel IPP 6.0 release. Added descriptions of new speech coding functions (noise reduction), new functions for regular expressions (multiple patterns), radix sorting, zero crossing measure, data compression (ZLIB). Documented new common functions and internationalization functions.	08/2008
-024	Documents Intel IPP 6.1 beta release. Added descriptions of new speech recognition (noise detect) and data compression (ZLIB) functions.	01/2009
-025	Documents Intel IPP 6.1 release. Added descriptions of new core functions (dispatching control), new transform functions (Walsh-Hadamard transform, DCT type IV, DFT for real signal of fixed length). Updated description of Fixed-Accuracy Arithmetic functions.	03/2009
-026	Documents Intel IPP 6.1 Update 3 release. Added new code examples for String Functions.	11/2009



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See [http://www.intel.com/products/processor\\_number](http://www.intel.com/products/processor_number) for details.

MPEG-1, MPEG-2, MPEG-4, H.261, H.263, H.264, MP3, DV, VC-1, MJPEG, AC3, AAC, G.711, G.722, G.722.1, G.722.2, AMRWB, Extended AMRWB (AMRWB+), G.167, G.168, G.169, G.723.1, G.726, G.728, G.729, G.729.1, GSM AMR, GSM FR are international standards promoted by ISO, IEC, ITU, ETSI, 3GPP and other organizations. Implementations of these standards, or the standard enabled platforms may require licenses from various entities, including Intel Corporation.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Inside, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

\* Other names and brands may be claimed as the property of others.

Copyright© 2000-2009, Intel Corporation. All rights reserved.

---

---

# Contents

<b>Version Information.....</b>	<b>3</b>
<b>Legal Information.....</b>	<b>5</b>
<b>Chapter 1: Overview</b>	
About This Software .....	51
Hardware and Software Requirements.....	51
Platforms Supported.....	52
Cross-Architecture Alignment .....	52
API Changes in Version 5.0.....	52
Technical Support.....	53
Intel IPP Code Samples.....	53
About This Manual .....	53
Manual Organization .....	53
Function Descriptions .....	55
Enumerators.....	55
Audience for This Manual .....	60
Related Publications.....	60
Notational Conventions .....	60
<b>Chapter 2: Intel® Integrated Performance Primitives Concepts</b>	
Basic Features.....	63
Function Naming.....	63
Data-Domain.....	64
Name.....	64
Data Types.....	64

Descriptor.....	67
Parameters.....	67
Structures and Enumerators.....	68
Library Version Structure.....	68
Complex Data Structures.....	68
Function Context Structures.....	69
Enumerators.....	69
Data Ranges.....	74
Data Alignment.....	74
Integer Scaling.....	75
Error Reporting.....	75
Code Examples.....	81

### **Chapter 3: Support Functions**

Version Information Functions.....	87
ippGetLibVersion.....	87
Memory Allocation Functions.....	88
ippMalloc.....	89
ippFree.....	90
Common Functions.....	91
ippGetStatusString.....	91
ippGetCpuType.....	92
ippGetCpuClocks.....	94
ippGetCpuFreqMhz.....	95
ippGetCpuFeatures.....	95
ippGetNumCoresOnDie.....	98
ippGetMaxCacheSizeB.....	98
ippSetFlushToZero.....	99
ippSetDenormsToZero.....	100
ippAlignPtr.....	100
ippSetNumThreads.....	101
ippGetNumThreads.....	102

ippMalloc.....	102
ippFree.....	103
Dispatcher Control Functions.....	103
ippStaticInit.....	103
ippInit.....	104
ippStaticInitCpu.....	105
ippInitCpu.....	106
ippEnableCpu.....	106
Internationalization Functions.....	107
ippMeassgeCatalogOpenI18n.....	107
ippMessageCatalogCloseI18n.....	108
ippGetMessageStatusI18n.....	109
StatusToMessageI18n.....	110

## **Chapter 4: Vector Initialization Functions**

Vector Initialization Functions.....	113
Copy.....	113
PackBits.....	116
Move.....	117
Set.....	119
Zero.....	120
Sample-Generating Functions.....	121
Tone Generating Functions.....	122
ToneInitAllocQ15.....	122
ToneFree.....	123
ToneGetStateSizeQ15.....	124
ToneInitQ15.....	124
ToneQ15.....	126
Tone_Direct.....	128
ToneQ15_Direct.....	129
Triangle-Generating Functions.....	130
TriangleInitAllocQ15.....	132

TriangleFree.....	133
TrianglrGetStateSizeQ15.....	134
TriangleInitQ15.....	135
TriangleQ15.....	136
Triangle_Direct.....	137
TriangleQ15_Direct.....	140
Uniform Distribution Functions.....	141
RandUniformInitAlloc.....	141
RandFree.....	142
RandUniformInit.....	143
RandUniformGetSize.....	144
RandUniform.....	144
RandUniform_Direct.....	147
Gaussian Distribution Functions.....	148
RandGaussInitAlloc.....	148
RandGaussFree.....	149
RandGaussInit.....	150
RandGaussGetSize.....	151
RandGauss.....	151
RandGauss_Direct.....	152
Special Vector Functions.....	154
VectorJaehne.....	154
VectorSlope.....	155
VectorRamp.....	157

## **Chapter 5: Essential Functions**

Logical and Shift Functions.....	166
AndC.....	166
And.....	167
OrC.....	168
Or.....	169
XorC.....	170

Xor.....	171
Not.....	172
LShiftC.....	173
RShiftC.....	174
Arithmetic Functions.....	176
AddC.....	177
Add.....	179
AddProductC.....	182
AddProduct.....	183
MulC.....	184
Mul.....	186
SubC.....	189
SubCRev.....	192
Sub.....	194
DivC.....	196
DivCRev.....	198
Div.....	199
Div_Round.....	203
Abs.....	205
Sqr.....	206
Sqrt.....	208
Cubrt.....	211
Exp.....	212
Ln.....	214
10Log10.....	216
SumLn.....	217
Arctan.....	218
Normalize.....	219
Cauchy, CauchyD, CauchyDD2.....	221
Conversion Functions.....	222
SortAscend, SortDescend.....	223
SortIndexAscend, SortIndexDwscend.....	224

SortRadixAscend, SortRadixDescend.....	226
SortRadixAscend, SortRadixDescend.....	228
SwapBytes.....	231
Convert.....	233
Join.....	239
JoinScaled.....	241
SplitScaled.....	243
Conj.....	244
ConjFlip.....	245
Magnitude.....	246
MagSquared.....	248
Phase.....	249
PowerSpectr.....	250
Real.....	252
Imag.....	253
RealToCplx.....	254
CplxToReal.....	256
DemodulateFM.....	257
Threshold.....	258
Threshold_LT, Threshold_GT.....	262
Threshold_LTAbs, Threshold_GTAb.....	266
Threshold_LTVal, Threshold_GTVal, Threshold_LTValGTVal.....	268
Threshold_LTInv.....	273
CartToPolar.....	276
PolarToCart.....	278
MaxOrder.....	279
Preemphasize.....	280
Flip.....	281
FindNearestOne.....	282
FindNearest.....	283
Viterbi Decoder Functions.....	284
GetVarPointDV.....	284



CalcStatesDV.....	285
BuilSymbITableDV4D.....	286
UpdatePathMetricsDV.....	287
Windowing Functions.....	287
Understanding Window Functions.....	288
WinBartlett.....	289
WinBlackman.....	291
WinHamming.....	295
WinHan.....	297
WinKaiser.....	299
Statistical Functions.....	301
Sum.....	301
Max.....	303
MaxIndx.....	304
MaxAbs.....	305
MaxAbsIndx.....	306
Min.....	307
MinIndx.....	308
MinAbs.....	309
MinAbsIndx.....	310
MinMax.....	311
MinMaxIndx.....	312
Mean.....	313
StdDev.....	315
MeanStdDev.....	316
Norm.....	318
NormDiff.....	320
DotProd.....	323
MaxEvery, MinEvery.....	326
ZeroCrossing.....	327
CountInRange.....	329
Sampling Functions.....	330

SampleUp.....	330
SampleDown.....	332

## Chapter 6: Filtering Functions

Convolution and Correlation Functions.....	335
Conv.....	336
ConvBiased.....	337
ConvCyclic.....	339
AutoCorr.....	339
CrossCorr.....	342
UpdateLinear.....	345
UpdatePower.....	347
Filtering Functions.....	348
SumWindow.....	351
FIR Filter Functions.....	352
FIRInitAlloc.....	353
FIRStreamInitAlloc.....	355
FIRMRInitAlloc.....	357
FIRMRStreamInitAlloc.....	360
FIRFree.....	362
FIRInit.....	363
FIRStreamInit.....	366
FIRMRInit.....	367
FIRMRStreamInit.....	370
FIRGetStateSize, FIRMRGetStateSize.....	372
FIRStreamGetStateSize, FIRMRStreamGetStateSize.....	375
FIRGetTaps.....	376
FIRSetTaps.....	378
FIRGetDlyLine.....	380
FIRSetDlyLine.....	382
FIROne.....	383
FIR.....	385

FIROne_Direct.....	394
FIR_Direct.....	398
FIRMR_Direct.....	402
FIRSparseInit.....	406
FIRSparseGetStateSize.....	408
FIRSparse.....	409
FIR Filter Coefficient Generating Functions .....	410
FIRGenLowpass.....	411
FIRGenHighpass.....	414
FIRGenBandPass.....	416
FIRGenBandstop.....	419
Single-Rate FIR LMS Filter Functions.....	421
FIRLMSInitAlloc.....	421
FIRLMSFree.....	422
FIRLMSGetTaps.....	423
FIRLMSGetDlyLine.....	424
FIRLMSSetDlyLine.....	425
FIRLMS.....	426
FIRLMSOne_Direct.....	428
Multi-Rate FIR LMS Filter Functions.....	431
FIRLMSMRInitAlloc.....	431
FIRLMSMRFree.....	433
FIRLMSMRSetMu.....	433
FIRLMSMRUpdateTaps.....	434
FIRLMSMRGetTaps.....	435
FIRLMSMRSetTaps.....	436
FIRLMSMRGetTapsPointer.....	437
FIRLMSMRGetDlyLine.....	438
FIRLMSMRSetDlyLine.....	439
FIRLMSMRGetDlyVal.....	440
FIRLMSMRPutVal.....	441
FIRLMSMROne.....	441

FIRLMSMROneVal.....	442
IIR Filter Functions.....	443
IIRInitAlloc.....	446
IIRInitAlloc_BiQuad.....	448
IIRFree.....	450
IIRInit.....	452
IIRInit_BiQuad.....	454
IIRGetStateSize.....	457
IIRGetStateSize_BiQuad.....	458
IIRSetTaps.....	459
IIRGetDlyLine.....	461
IIRSetDlyLine.....	463
IIROne.....	464
IIR.....	466
IIROne_Direct.....	474
IIROne_BiQuadDirect.....	475
IIR_Direct.....	477
IIR_BiQuadDirect.....	478
IIRsparseInit.....	479
IIRsparseGetStateSize.....	481
IIRsparse.....	482
IIRGenLowpass, IIRGenHighpass.....	484
Median Filter Functions.....	486
FilterMedian.....	487

## Chapter 7: Transform Functions

Fourier Transform Functions.....	493
Special Arguments.....	493
Packed Formats.....	494
Format Conversion Functions.....	496
ConjPack.....	496
ConjPerm.....	498

ConjCcs.....	500
Functions for Packed Data Multiplication.....	502
MulPack.....	503
MulPerm.....	505
MulPackConj.....	507
Fast Fourier Transform Functions.....	508
FFTInitAlloc.....	508
FFTFree.....	510
FFTInit.....	512
FFTGetSize.....	514
FFTGetBufSize.....	516
FFTFwd_CToC.....	517
FFTInv_CToC.....	521
FFTFwd_RToPack, FFTFwd_RToPerm, FFTFwd_RToCCS.....	524
FFTInv_PackToR, FFTInv_PermToR, FFTInv_CCSToR....	528
Discrete Fourier Transform Functions.....	532
DFTInitAlloc.....	532
DFTFree.....	534
DFTGetBufSize.....	535
DFTFwd_CToC.....	536
DFTInv_CToC.....	540
DFTFwd_RToPack, DFTFwd_RToPerm, DFTFwd_RToCCS.....	543
DFTInv_PackToR, DFTInv_PermToR, DFTInv_CCSToR....	548
DFTOutOrdInitAlloc.....	552
DFTOutOrdFree.....	553
DFTOutOrdGetBufSize.....	554
DFTOutOrdFwd_CToC.....	555
DFTOutOrdInv_CToC.....	556
DFT for a Given Frequency (Goertzel) Functions.....	557
Goertz.....	557

GoertzTwo.....	560
Hartley Transform Functions.....	562
Hartley.....	562
Walsh-Hadamard Transform Functions.....	564
WHT.....	564
WHTGetBufferSize.....	566
Discrete Cosine Transform Functions .....	566
DCTFwdInitAlloc.....	567
DCTInvInitAlloc.....	568
DCTFwdFree.....	569
DCTInvFree.....	569
DCTFwdGetBufSize.....	570
DCTInvGetBufSize.....	571
DCTFwdInit.....	572
DCTInvInit.....	573
DCTFwdGetSize.....	574
DCTInvGetSize.....	575
DCTFwd.....	576
DCTInv.....	578
DCT4InitAlloc.....	582
DCT4Free.....	583
DCT4Init.....	583
DCT4GetSize.....	584
DCT4.....	585
Hilbert Transform Functions.....	586
HilbertInitAlloc.....	586
HilbertFree.....	587
Hilbert.....	588
Wavelet Transform Functions.....	590
Transforms for Fixed Filter Banks.....	594
WTHaarFwd, WTHaarInv.....	594
Transforms for User Filter Banks.....	598

WTFwdInitAlloc, WTInvInitAlloc.....	599
WTFwdInitFree, WTInvFree.....	601
WTFwd.....	603
WTFwdSetDlyLine, WTFwdGetDlyLine.....	606
WTInv.....	608
WTInvSetDlyLine, WTInvGetDlyLine.....	611
Wavelet Transforms Example.....	614

## **Chapter 8: Speech Recognition Functions**

Basic Arithmetics.....	627
AddAllRowSum.....	627
SumColumn.....	629
SumRow.....	631
SubRow.....	633
CopyColumn_Indirect.....	634
BlockDMatrixInitAlloc.....	636
BlockDMatrixFree.....	637
NthMaxElement.....	637
VecMatMul.....	638
MatVecMul.....	640
Feature Processing.....	642
ZeroMean.....	642
CompensateOffset.....	643
SignChangeRate.....	645
LinearPrediction.....	646
Durbin.....	649
Schur.....	650
LPToSpectrum.....	652
LPToCepstrum.....	653
CepstrumToLP.....	654
LPToReflection.....	655
ReflectionToLP.....	656

ReflectionToAR.....	657
ReflectionToTilt.....	659
PitchmarkToF0Cand.....	660
UnitCurve.....	661
LPToLSP.....	663
LSPToLP.....	664
MelToLinear.....	665
LinearToMel.....	666
CopyWithPadding.....	667
MelFBankGetSize.....	668
MelFBankInit.....	669
MelFBankInitAlloc.....	670
MelLinFBankInitAlloc.....	672
EmptyFBankInitAlloc.....	674
FBankFree.....	675
FBankGetCenters.....	676
FBankSetCenters.....	676
FBankGetCoeffs.....	677
FBankSetCoeffs.....	678
EvalFBank.....	679
DCTLifterGetSize_MulC0.....	680
DCTLifterInit_MulC0.....	681
DCTLifterInitAlloc.....	682
DCTLifterFree.....	684
DCTLifter.....	684
NormEnergy.....	688
SumMeanVar.....	690
NewVar.....	692
RecSqrt.....	693
AccCovarianceMatrix.....	694
CompensateOffset.....	696
Derivative Functions .....	698



CopyColumn.....	699
EvalDelta.....	700
Delta.....	701
DeltaDelta.....	704
Pitch Super Resolution .....	707
CrossCorrCoeffDecim.....	707
CrossCorrCoeff.....	709
CrossCorrCoeffInterpolation.....	710
Model Evaluation.....	711
AddNRows.....	712
ScaleLM.....	713
LogAdd.....	714
LogSub.....	715
LogSum.....	716
MahDistSingle.....	717
MahDist.....	718
MahDistMultiMix.....	720
LogGaussSingle.....	721
LogGauss.....	724
LogGaussMultiMix.....	727
LogGaussMax.....	729
LogGaussMaxMultiMix.....	732
LogGaussAdd.....	735
LogGaussAddMultiMix.....	737
LogGaussMixture.....	739
LogGaussMixtureSelect.....	742
BuildSignTable.....	744
FillShortlist_Row.....	745
FillShortlist_Column.....	748
DTW.....	750
Model Estimation.....	751
MeanColumn.....	752

VarColumn.....	753
MeanVarColumn.....	754
WeightedMeanColumn.....	755
WeightedVarColumn.....	756
WeightedMeanVarColumn.....	757
NormalizeColumn.....	761
NormalizeInRange.....	762
MeanVarAcc.....	764
GaussianDist.....	765
GaussianSplit.....	766
GaussianMerge.....	767
Entropy.....	768
SinC.....	769
ExpNegSqr.....	770
BhatDist.....	771
UpdateMean.....	772
UpdateVar.....	773
UpdateWeight.....	774
UpdateGConst.....	775
OutProbPreCalc.....	776
DcsClustLAccumulate.....	778
DcsClustLCompute.....	779
Model Adaptation.....	780
AddMulColumn.....	780
AddMulRow.....	781
QRTransColumn.....	782
DotProdColumn.....	783
MulColumn.....	784
SumColumnAbs.....	785
SumColumnSqr.....	786
SumRowAbs.....	787
SumRowSqr.....	787

SVD, SVDSort.....	788
WeightedSum.....	790
Vector Quantization.....	791
FormVector.....	791
CdbkGetSize.....	794
CdbkInit.....	795
CdbkInitAlloc.....	796
CdbkFree.....	798
GetCdbkSize.....	799
GetCodebook.....	799
VQ.....	800
VQSingle.....	801
SplitVQ.....	803
FormVectorVQ.....	804
Polyphase Resampling.....	806
ResamplePolyphaseInit.....	806
ResamplePolyphaseGetSize.....	807
ResamplePolyphaseSetFilter.....	808
ResamplePolyphaseGetFilter.....	809
ResamplePolyphaseInitAlloc.....	812
ResamplePolyphaseFree.....	814
ResamplePolyphase.....	815
Advanced Aurora Functions.....	819
SmoothedPowerSpectrumAurora.....	819
NoiseSpectrumUpdate_Aurora.....	820
WienerFilterDesign_Aurora.....	821
MelfBankInitAlloc_Aurora.....	822
TabsCalculation_Aurora.....	823
ResidualFilter_Aurora.....	824
WaveProcessing_Aurora.....	824
LowHighFilter_Aurora.....	825
HighBandCoding_Aurora.....	826

BlindEqualization_Aurora.....	827
DeltaDelta_Aurora.....	828
VADGetBufSize_Aurora.....	831
VADInit_Aurora.....	831
VADDecision_Aurora.....	832
VADFlush_Aurora.....	833
Ephraim-Malah Noise Suppressor.....	834
Noise Suppressor Architecture.....	834
Data Structures .....	835
Filter Update Functions.....	835
FilterUpdateEMNS.....	835
FilterUpdateWiener.....	836
Noise Floor Estimation Functions.....	838
GetSizeMCRA.....	838
InitMCRA.....	838
AltInitMCRA.....	839
UpdateNoisePSDMCRA.....	840
Voice Activity Detector .....	841
Voice Activity Detector Architecture.....	841
Voice Activity Detection Functions.....	842
FindPeaks.....	842
PeriodicityLSPE.....	843
Periodicity.....	845

## **Chapter 9: Speech Coding Functions**

Rounding Mode.....	848
Notational Conventions.....	849
Definitions.....	849
Data Structures .....	850
Common Functions.....	851
ConvPartial.....	852
InterpolateC_NR.....	853

Mul_NR.....	854
MulC_NR.....	856
MulPowerC_NR.....	857
AutoScale.....	858
DotProdAutoScale.....	859
InvSqrt.....	860
AutoCorr.....	861
AutoCorrLagMax.....	862
AutoCorr_NormE.....	863
CrossCorr.....	865
CroosCorrLagMax.....	866
SynthesisFilter.....	867
G.729 Functions .....	869
Basic Functions.....	871
DotProd_G729.....	871
Interpolate_G729.....	872
Linear Prediction Analysis Functions.....	873
AutoCorr_G729.....	873
LevinsonDurbin_G729.....	875
LPCToLSP_G729.....	877
LSFToLSP_G729.....	879
LSFQuant_G729.....	880
LSFDecode_G729.....	881
LSFDecodeErased_G729.....	882
LSPToLPC_G729.....	883
LSPQuant_G729.....	884
LSPToLSF_G729.....	888
LagWindow_G729.....	889
Codebook Search Functions.....	890
OpenLoopPitchSearch_G729.....	890
AdaptiveCodebookSearch_G729.....	891
DecodeAdaptiveVector_G729.....	892

FixedCodebookSearch_G729.....	893
GainCodebookSearch_G729.....	896
ToeplizMatrix_G729.....	898
Codebook Gain Functions.....	899
DecodeGain_G729.....	900
GainControl_G729.....	901
GainQuant_G729.....	903
AdaptiveCodebookContribution_G729.....	904
AdaptiveCodebookGain_G729.....	905
Filter Functions.....	906
ResidualFilter_G729.....	906
SynthesisFilter_G729.....	908
LongTermPostFilter_G729.....	910
ShortTermPostFilter_G729.....	911
TiltCompensation_G729.....	913
HarmonicFilter.....	914
HighPassFilterSize_G729.....	915
HighPassFilterInit_G729.....	916
HighPassFilter_G729.....	916
IIR16s_G729.....	917
PhaseDispersionGetStateSize_G729D.....	919
PhaseDispersionInit_G729D.....	919
PhaseDispersionUpdate_G729D.....	920
PhaseDispersion_G729D.....	920
Preemphasize_G729A.....	921
WinHybridGetStateSize_G729E.....	922
WinHybridInit_G729E.....	923
WinHybrid_G729E.....	923
RandomNoiseExcitation_G729B.....	925
FilteredExcitation_G729.....	926
G.729.1 Functions.....	926
FilterHighpassGetStateSize_G7291.....	928

FilterHighpassInit_G7291.....	928
FilterHighpass_G7291.....	929
FilterLowpass_G7291.....	930
QMFFGetStateSize_G7291.....	931
QMFFInit_G7291.....	931
QMFFEncode_G7291.....	932
QMFFDecode_G7291.....	933
LSFFDecode_G7291.....	934
AdaptiveCodebookSearch_G7291.....	934
AdaptiveCodebookGain_G7291.....	936
AlgebraicCodebookSearch_G7291.....	937
GainQuant_G7291.....	938
EnvelopTime_G7291.....	939
EnvelopFrequency_G7291.....	940
GenerateExcitationGetStateSize_G7291.....	940
GenerateExcitationInit_G7291.....	941
GenerateExcitation_G7291.....	942
ShapeEnvelopTime_G7291.....	942
ShapeEnvelopFrequency_G7291.....	943
CompressEnvelopTime_G7291.....	944
MDCTFwd_G7291.....	945
MDCTInv_G7291.....	945
MDCTQuantFwd_G7291.....	946
MDCTQuantInv_G7291.....	947
MDCTPostProcess_G7291.....	947
GainControl_G7291.....	948
TiltCompensation_G7291.....	949
QuantParam_G7291.....	950
G.723.1 Functions.....	950
Linear Prediction Analysis Functions.....	951
AutoCorr_G723.....	952
AutoCorr_NormE_G723.....	953

LevinsonDurbin_G723.....	954
LPCToLSF_G723.....	956
LSFToLPC_G723.....	957
LSFDecode_G723.....	958
LSFQuant_G723.....	959
Codebook Search Functions.....	960
OpenLoopPitchSearch_G723.....	960
ACELPFixedCodebookSearch_G723.....	961
AdaptiveCodebookSearch_G723.....	963
MPMLQFixedCodebookSearch_G723.....	964
ToeplitzMatrix_G723.....	966
Gain Quantization.....	967
GainQuant_G723.....	967
GainControl_G723.....	969
Filtering Functions.....	970
HighPassFilter_G723.....	970
IIR16s_G723.....	971
SynthesisFilter_G723.....	973
TiltCompensation_G723.....	974
HarmonicSearch_G723.....	975
HarmonicNoiseSubtract_G723.....	975
DecodeAdaptiveVector_G723.....	976
PitchPostFilter_G723.....	977
GSM-AMR Functions .....	978
Basic Functions.....	980
Interpolate_GSMAMR.....	980
FFTFwd_RToPerm_GSMAMR.....	981
LP Analysis and Quantization Functions .....	982
AutoCorr_GSMAMR.....	982
LevinsonDurbin_GSMAMR.....	984
LPCToLSP_GSMAMR.....	985
LSPToLPC_GSMAMR.....	986



LSFToLSP_GSMAMR.....	988
LSPQuant_GSMAMR.....	988
QuantLSPDecode_GSMAMR.....	990
Adaptive Codebook Functions .....	991
Open-Loop Pitch Search (OLP).....	992
OpenLoopPitchSearchNonDTX_GSMAMR.....	994
OpenLoopPitchSearchDTXVAD1_GSMAMR.....	996
OpenLoopPitchSearchDTXVAD2_GSMAMR.....	999
ImpulseResponseTarget_GSMAMR.....	1002
AdaptiveCodebookSearch_GSMAMR.....	1004
AdaptiveCodebookDecode_GSMAMR.....	1005
AdaptiveCodebookGain_GSMAMR.....	1007
Fixed Codebook Search .....	1009
AlgebraicCodebookSearch_GSMAMR.....	1009
FixedCodebookDecode_GSMAMR.....	1011
Discontinuous Transmission (DTX) .....	1012
Preemphasize_GSMAMR.....	1012
VAD1_GSMAMR.....	1013
VAD2_GSMAMR.....	1015
EncDTXSID_GSMAMR.....	1018
EncDTXHandler_GSMAMR.....	1019
EncDTXBuffer_GSMAMR, DecDTXBuffer_GSMAMR.....	1021
Post Processing .....	1022
PostFilter_GSMAMR.....	1022
AMR Wideband Functions.....	1024
ResidualFilter_AMRWB.....	1025
LPC Analysis Functions.....	1026
LPCToISP_AMRWB.....	1026
ISPToLPC_AMRWB.....	1028
ISPToISF_Norm_AMRWB.....	1029
ISFToISP_AMRWB.....	1030
Open-loop Pitch Analysis Functions.....	1031

OpenLoopPitchSearch_AMRWB.....	1031
Filtering Functions.....	1032
HighPassFilterGetSize_AMRWB.....	1032
HighPassFilterInit_AMRWB.....	1033
HighPassFilter_AMRWB.....	1034
HighPassFilterGetDlyLine_AMRWB.....	1035
HighPassFilterSetDlyLine_AMRWB.....	1036
Preemphasize_AMRWB.....	1037
Deemphasize_AMRWB.....	1038
SynthesisFilter_AMRWB.....	1039
Discontinuous Transmission (DTX) Functions.....	1040
VADGetSize_AMRWB.....	1040
VADInit_AMRWB.....	1040
VAD_AMRWB.....	1041
VADGetEnergyLevel_AMRWB.....	1042
Codebook Search Functions.....	1042
AlgebraicCodebookSearch_AMRWB.....	1043
AlgebraicCodebookDecode_AMRWB.....	1048
AdaptiveCodebookGainCoeff_AMRWB.....	1049
AdaptiveCodebookSearch_AMRWB.....	1050
AdaptiveCodebookDecodeGetSize_AMRWB.....	1052
AdaptiveCodebookDecodeInit_AMRWB.....	1052
AdaptiveCodebookDecodeUpdate_AMRWB.....	1053
AdaptiveCodebookDecode_AMRWB.....	1054
Quantization Functions.....	1055
ISFQuant_AMRWB.....	1055
ISFQuantDecode_AMRWB.....	1057
ISFQuantDTX_AMRWB.....	1058
ISFQuantDecodeDTX_AMRWB.....	1059
GainQuant_AMRWB.....	1059
DecodeGain_AMRWB.....	1061
EncDTXBuffer_AMRWB.....	1062

DecDTXBuffer_AMRWB.....	1063
AMR Wideband Plus Functions.....	1064
SNR_AMRWBE.....	1067
OpenLoopPitchSearch_AMRWBE.....	1069
LPCToISP_AMRWBE.....	1070
Filtering Functions.....	1071
SynthesisFilter_AMRWBE.....	1071
Deemphasize_AMRWBE.....	1072
FIRGenMidBand_AMRWBE.....	1072
PostFilterLowBand_AMRWBE.....	1073
Fast Fourier Transform Functions.....	1074
FFTFwd_RToPerm_AMRWBE.....	1075
FFTIInv_PermToR_AMRWBE.....	1076
Codebook Search Functions.....	1076
AdaptiveCodebookSearch_AMRWBE.....	1077
AdaptiveCodebookDecode_AMRWBE.....	1078
Resample Functions.....	1079
Downsample_AMRWBE.....	1079
Upsample_AMRWBE.....	1080
BandSplit_AMRWBE.....	1081
BandJoin_AMRWBE.....	1082
BandSplitDownsample_AMRWBE.....	1083
BandJoinUpsample_AMRWBE.....	1084
ResamplePolyphase_AMRWBE.....	1085
Quantization Functions.....	1087
ISFQuantDecode_AMRWBE.....	1087
ISFQuantDecodeHighBand_AMRWBE.....	1088
ISFQuantHighBand_AMRWBE.....	1089
GainQuant_AMRWBE.....	1090
QuantTCX_AMRWBE.....	1091
GainQuantTCX_AMRWBE.....	1092
GainDecodeTCX_AMRWBE.....	1094

EncodeMux_AMRWBE.....	1095
DecodeDemux_AMRWBE.....	1096
GSM Full Rate Functions.....	1096
RPEQuantDecode_GSMFR.....	1097
Deemphasize_GSMFR.....	1098
ShortTermAnalysisFilter_GSMFR.....	1099
ShortTermSynthesisFilter_GSMFR.....	1100
HighPassFilter_GSMFR.....	1101
Schur_GSMFR.....	1102
WeightingFilter_GSMFR.....	1102
Preemphasize_GSMFR.....	1103
G.722.1 Functions.....	1104
DCTFwd_G722.....	1105
DCTInv_G722.....	1106
DecomposeMLTToDCT_G722.....	1107
DecomposeDCTToMLT_G722.....	1108
HuffmanEncode_G722.....	1110
G.726 Functions.....	1110
EncodeGetStateSize_G72.....	1111
EncodeInit_G726.....	1112
Encode_G726.....	1112
DecodeGetStateSize_G726.....	1113
DecodeInit_G726.....	1114
Decode_G726.....	1115
G.728 Functions.....	1115
IIRGetStateSize_G728.....	1117
IIR_Init_G728.....	1117
IIR_G728.....	1118
SynthesisFilterGetStateSize_G728.....	1119
SynthesisFilterInit_G728.....	1119
SynthesisFilter_G728.....	1120
CombinedFilterGetStateSize_G728.....	1121

CombinedFilterInit_G728.....	1121
CombinedFilter_G728.....	1122
PostFilterGetStateSize_G728.....	1123
PostFilterInit_G728.....	1124
PostFilter_G728.....	1124
PostFilterAdapterGetStateSize_G728.....	1126
PostFilterAdapterStateInit_G728.....	1126
LPCInverseFilter_G728.....	1127
PitchPeriodExtraction_G728.....	1128
WinHybridGetStateSize_G728.....	1129
WinHybridInit_G728.....	1129
WinHybrid_G728.....	1130
LevinsonDurbin_G728.....	1132
CodebookSearch_G728.....	1133
CodebookSearchTCQ_G728.....	1134
ImpulseResponseEnergy_G728.....	1135
Voice Enhancement Functions.....	1136
Echo Canceller Functions.....	1138
SubbandProcessGetSize.....	1138
SubbandProcessInit.....	1139
SubbandAnalysis.....	1140
SubbandSynthesis.....	1141
SubbandControllerGetSize_EC.....	1143
SubbandControllerInit_EC.....	1144
SubbandControllerUpdate_EC.....	1145
SubbandController_EC.....	1146
SubbandControllerReset_EC.....	1147
SubbandControllerDTGetSize_EC.....	1148
SubbandControllerDTInit_EC.....	1149
SubbandControllerDTReset_EC.....	1150
SubbandControllerDT_EC.....	1150
SubbandControllerDTUpdate_EC.....	1152

ToneDetectGetStateSize_EC.....	1154
ToneDetectInit_EC.....	1154
ToneDetect_EC.....	1155
FullbandControllerGetSize_EC.....	1156
FullbandControllerInit_EC.....	1157
FullbandControllerUpdate_EC.....	1158
FullbandController_EC.....	1159
FullbandControllerReset_EC.....	1160
FIR_EC.....	1161
FIRSubband_EC, FIRSubbandLow_EC.....	1162
FIRSubbandCoeffUpdate_EC, FIRSubbandLowCoeffUpdate_EC.....	1163
NLMS_EC.....	1165
Noise Reduction Functions.....	1166
FilterNoiseGetStateSize.....	1166
FilterNoiseInit.....	1167
FilterNoiseLevel.....	1168
FilterNoiseDetect_EC.....	1169
FilterNoiseDetectModerate_EC.....	1170
FilterNoiseSetMode_EC.....	1171
FilterNoise.....	1172
Code Examples.....	1173
Automatic Audio Level Control Functions.....	1174
ALCGetStateSize_G169.....	1175
ALCInit_G169.....	1175
ALCSetLevel_G169.....	1176
ALCSetGain_G169.....	1177
ALC_G169.....	1177
G722 Sub-Band ADPCM Speech Codec Functions.....	1178
SBADPCMEncodeStateSize_G722.....	1179
SBADPCMEncodeInit_G722.....	1179
SBADPCMEncode_G722.....	1180

---

QMFEncode_G722.....	1181
SBADPCMDdecodeStateSize_G722.....	1182
SBADPCMDdecodeInit_G722.....	1182
SBADPCMDdecode_G722.....	1183
QMFDecode_G722.....	1184
Companding Functions.....	1185
MuLawToLin.....	1186
LinToMuLaw.....	1187
ALawToLin.....	1188
LinToALaw.....	1189
MuLawToALaw.....	1190
ALawToMuLaw.....	1191
RT Audio Functions.....	1192
LPCToLSP_RTA.....	1193
LSPToLPC_RTA.....	1194
LevinsonDurbin_RTA.....	1194
QMFFGetStateSize_RTA.....	1195
QMFFInit_RTA.....	1196
QMFFDecode_RTA.....	1196
QMFFEncode_RTA.....	1197
PostFilterGetStateSize_RTA.....	1198
PostFilterInit_RTA.....	1199
PostFilter_RTA.....	1199
AdaptiveCodebookSearch_RTA.....	1200
FixedCodebookSearch_RTA, FixedCodebookSearchRandom_RTA.....	1201
LSPQuant_RTA.....	1203
HighPassFilter_RTA.....	1204
BandPassFilter_RTA.....	1205

## Chapter 10: Audio Coding Functions

Interleaved to Multi-Row Format Conversion Functions.....	1213
---	------

Interleave.....	1213
Deinterleave.....	1215
Spectral Data Prequantization Functions.....	1217
Pow34.....	1217
Pow43.....	1221
Pow43Scale.....	1222
Scale Factors Calculation Functions.....	1226
CalcSF.....	1226
Mantissa Conversion and Scaling Functions.....	1228
Scale.....	1228
MakeFloat.....	1230
Modified Discrete Cosine Transform Functions.....	1232
MDCTFwdInitAlloc, MDCTInvInitAlloc.....	1232
MDCTFwdInit, MDCTInvInit.....	1234
MDCTFwdFree, MDCTInvFree.....	1235
MDCTFwdGetSize, MDCTInvGetSize.....	1236
MDCTFwdGetBufSize, MDCTInvGetBufSize.....	1237
MDCTFwd, MDCTInv.....	1238
Block Filtering Functions.....	1242
FIRBlockInitAlloc.....	1242
FIRBlockFree.....	1243
FIRBlockOne.....	1244
Frequency Domain Prediction Functions.....	1247
FDPInitAlloc.....	1247
FDPInit.....	1248
FDPFree.....	1249
FDPGetSize.....	1249
FDPReset.....	1250
FDPResetSfb.....	1250
FDPResetGroup.....	1251
FDPFwd.....	1253
FDPInv.....	1253



VLC Functions.....	1255
VLCDecodeEscBlock_MP3.....	1255
VLCDecodeEscBlock_AAC.....	1258
VLCDecodeUTupleEscBlock_MP3.....	1260
VLCDecodeUTupleEscBlock_AAC.....	1261
VLCCountEscBits_MP3.....	1262
VLCCountEscBits_AAC.....	1263
VLCEncodeEscBlock_MP3.....	1264
VLCEncodeEscBlock_AAC.....	1266
Psychoacoustic Functions.....	1267
Spread.....	1267
Vector Quantization Functions.....	1269
VQCodeBookInitAlloc.....	1269
VQCodeBookInit.....	1270
VQCodeBookFree.....	1271
VQCodeBookGetSize.....	1271
VQPreliminarySelect.....	1272
VQMainSelect.....	1274
VQIndexSelect.....	1276
VQReconstruction.....	1279
MP3 Audio Coding Functions.....	1280
Macros and Constants.....	1280
Data Structures.....	1281
MP3 Codec Enumerated Types .....	1286
MP3 Audio Encoder.....	1287
AnalysisPQMF_MP3.....	1288
AnalysisFilterInit_PQMF_MP3.....	1290
AnalysisFilterInitAlloc_PQMF_MP3.....	1290
AnalysisFilterGetSize_PQMF_MP3.....	1291
AnalysisFilterFree_PQMF_MP3.....	1291
AnalysisFilter_PQMF_MP3.....	1292
MDCTFwd_MP3.....	1293

PsychoacousticModelTwo_MP3.....	1295
JointStereoEncode_MP3.....	1300
Quantize_MP3.....	1303
PackScaleFactors_MP3.....	1308
HuffmanEncode_MP3.....	1310
PackFrameHeader_MP3.....	1313
PackSideInfo_MP3.....	1314
BitReservoirInit_MP3.....	1316
MP3 Audio Decoder.....	1317
UnpackFrameHeader_MP3.....	1318
UnpackSideInfo_MP3.....	1319
UnpackScaleFactors_MP3.....	1321
HuffmanDecode_MP3, HuffmanDecodeSfb_MP3, HuffmanDecodeSfbMbp_MP3.....	1323
ReQuantize_MP3, ReQuantizeSfb_MP3.....	1325
MDCTInv_MP3.....	1327
SynthPQMF_MP3.....	1329
SynthesisFilterInit_PQMF_MP3.....	1330
SynthesisFilterInitAlloc_PQMF_MP3.....	1331
SynthesisFilterGetSize_PQMF_MP3.....	1332
SynthesisFilterFree_PQMF_MP3.....	1332
SynthesisFilter_PQMF_MP3.....	1333
Advanced Audio Coding Functions.....	1334
Global Macros.....	1334
Data Types and Structures.....	1335
AAC Decoding Functions.....	1346
MPEG-2 AAC Functions .....	1348
MPEG-4 AAC Functions .....	1371
Spectral Band Replication Functions.....	1388
SBR Audio Encoder Functions.....	1388
DetectTransient_SBR.....	1390
EstimateTNR_SBR.....	1392

AnalysisFilterEncGetSize_SBR.....	1394
AnalysisFilterEncInit_SBR.....	1394
AnalysisFilterEnc_SBR.....	1395
AnalysisFilterEncInitAlloc_SBR.....	1396
AnalysisFilterEncFree_SBR.....	1397
SBR Audio Decoder Functions.....	1398
QMF Functions.....	1398
Prediction Functions.....	1429
Parametric Stereo Functions.....	1433
PS Audio Decoder Functions.....	1434
AnalysisFilter_PS.....	1434
DTS Audio Coding Functions.....	1436
DTS Audio Decoder.....	1436
SynthesisFilterInit_DTS.....	1436
SynthesisFilterInitAlloc_DTS.....	1437
SynthesisFilterGetSize_DTS.....	1437
SynthesisFilterFree_DTS.....	1438
SynthesisFilter_DTS.....	1439

## **Chapter 11: String Functions**

String Manipulation.....	1443
Find, FindRev.....	1443
FindC, FindRevC.....	1445
FindCAny, FindRevCAny.....	1447
Insert.....	1448
Remove.....	1450
Compare.....	1451
CompareIgnoreCase, CompareIgnoreCaseLatin.....	1452
Equal.....	1454
TrimC.....	1455
TrimCAny, TrimStartCAny, TrimEndCAny.....	1457
ReplaceC.....	1458

Uppercase, UppercaseLatin.....	1459
Lowercase, LowercaseLatin.....	1460
Hash.....	1462
Concat.....	1463
ConcatC.....	1465
SplitC.....	1466
Regular Expressions.....	1468
RegExpInitAlloc.....	1469
RegExpFree.....	1470
RegExpInit.....	1470
RegExpGetSize.....	1472
RegExpSetMatchLimit.....	1472
RegExpFind.....	1473
RegExpSetFormat.....	1475
ConvertUTF.....	1477
RegExpMultiGetSize.....	1478
RegExpMultiInit.....	1478
RegExpMultiInitAlloc.....	1479
RegExpMultiFree.....	1480
RegExpMultiAdd.....	1480
RegExpMultiDelete.....	1481
RegExpMultiModify.....	1482
RegExpMultiFind.....	1483
Example of Using Multipattern Functions.....	1485
RegExpReplaceGetSize.....	1488
RegExpReplaceInit.....	1489
RegExpReplace.....	1490

## **Chapter 12: Fixed-Accuracy Arithmetic Functions**

Arithmetic Functions.....	1496
Add.....	1496
Sub.....	1498

Sqr.....	1500
Mul.....	1502
MulByConj.....	1505
Conj.....	1507
Abs.....	1509
Arg.....	1511
Power and Root Functions.....	1513
Inv.....	1513
Div.....	1515
Sqrt.....	1518
InvSqrt.....	1521
Cbrt.....	1523
InvCbrt.....	1525
Pow2o3.....	1527
Pow3o2.....	1528
Pow.....	1530
Powx.....	1533
Hypot.....	1537
Exponential and Logarithmic Functions.....	1539
Exp.....	1539
Expm1.....	1542
Ln.....	1544
Log10.....	1546
Log1p.....	1549
Trigonometric Functions.....	1550
Cos.....	1550
Sin.....	1552
SinCos.....	1555
CIS.....	1557
Tan.....	1559
Acos.....	1561
Asin.....	1564

Atan.....	1566
Atan2.....	1569
Hyperbolic Functions.....	1571
Cosh.....	1571
Sinh.....	1574
Tanh.....	1576
Acosh.....	1578
Asinh.....	1581
Atanh.....	1584
Special Functions.....	1586
Erf.....	1586
Erfc.....	1588
CdfNorm.....	1591
ErfInv.....	1593
ErfcInv.....	1596
CdfNormInv.....	1599
Rounding Functions.....	1602
Floor.....	1602
Ceil.....	1604
Trunc.....	1605
Round.....	1607
NearbyInt.....	1608
Rint.....	1610
Modf.....	1612

## **Chapter 13: Data Compression Functions**

Application Notes.....	1620
VLC and Huffman Coding Functions.....	1621
Data Compression VLC functions.....	1621
VLCEncodeInitAlloc.....	1622
VLCEncodeFree.....	1622
VLCEncodeInit.....	1623

VLCEncodeGetSize.....	1624
VLCEncodeBlock.....	1625
VLCEncodeOne.....	1626
VLCCountBits.....	1627
VLCDecodeInitAlloc.....	1627
VLCDecodeFree.....	1628
VLCDecodeInit.....	1629
VLCDecodeGetSize.....	1630
VLCDecodeBlock.....	1631
VLCDecodeOne.....	1632
Example of Using VLC Coding Functions .....	1633
VLCDecodeUTupleInitAlloc.....	1634
VLCDecodeUTupleFree.....	1635
VLCDecodeUTupleInit.....	1636
VLCDecodeUTupleGetSize.....	1637
VLCDecodeUTupleBlock.....	1638
VLCDecodeUTupleOne.....	1639
Huffman Coding.....	1640
EncodeHuffInitAlloc.....	1641
HuffFree.....	1642
EncodeHuffInit.....	1643
HuffGetSize.....	1643
EncodeHuffOne.....	1644
EncodeHuff.....	1645
EncodeHuffFinal.....	1646
HuffGetLenCodeTable.....	1646
DecodeHuffInitAlloc.....	1647
DecodeHuffInit.....	1648
DecodeHuffOne.....	1648
DecodeHuff.....	1649
HuffGetDstBuffSize.....	1650
HuffLenCodeTablePack.....	1651

HuffLenCodeTableUnpack.....	1652
Dictionary-Based Compression Functions.....	1653
LZSS Compression Functions.....	1653
EncodeLZSSInitAlloc.....	1653
LZSSFree.....	1654
EncodeLZSSInit.....	1654
LZSSGetSize.....	1655
EncodeLZSS.....	1655
EncodeLZSSFlush.....	1656
DecodeLZSSInitAlloc.....	1660
DecodeLZSSInit.....	1661
DecodeLZSS.....	1662
ZLIB Coding Functions .....	1666
Special Parameters .....	1669
EncodeLZ77Init.....	1671
EncodeLZ77GetSize.....	1672
EncodeLZ77InitAlloc.....	1672
LZ77Free.....	1673
EncodeLZ77.....	1674
EncodeLZ77SelectMode.....	1675
EncodeLZ77FixedHuff.....	1676
EncodeLZ77DynamicHuff.....	1677
EncodeLZ77StoredBlock.....	1678
EncodeLZ77Flush.....	1679
EncodeLZ77GetPairs.....	1680
EncodeLZ77SetPairs.....	1681
EncodeLZ77GetStatus.....	1681
EncodeLZ77SetStatus.....	1682
EncodeLZ77Reset.....	1683
DecodeLZ77Init.....	1683
DecodeLZ77GetSize.....	1684
DecodeLZ77InitAlloc.....	1685



DecodeLZ77.....	1685
DecodeLZ77GetBlockType.....	1686
DecodeLZ77FixedHuff.....	1688
DecodeLZ77DynamicHuff.....	1689
DecodeLZ77StoredBlock.....	1690
DecodeLZ77GetPairs.....	1691
DecodeLZ77SetPairs.....	1692
DecodeLZ77GetStatus.....	1692
DecodeLZ77SetStatus.....	1693
DecodeLZ77Reset.....	1694
Adler32.....	1694
CRC32, CRC32C.....	1695
DeflateLZ77.....	1698
DeflateDictionarySet.....	1700
DeflateHuff.....	1701
InflateBuildHuffTable.....	1702
Inflate.....	1703
Example of Using Intel IPP ZLIB Coding Functions.....	1704
LZO Compression Functions.....	1707
Special Parameters.....	1707
EncodeLZOGetSize.....	1707
EncodeLZOInit.....	1708
EncodeLZO.....	1709
DecodeLZO.....	1710
DecodeLZOSafe.....	1711
BWT-Based Compression Functions.....	1715
Burrows-Wheeler Transform.....	1715
BWTFwdGetSize.....	1717
BWTFwd.....	1718
BWTInvGetSize.....	1719
BWTInv.....	1719
BWTGetSize_SmallBlock.....	1721

BWTFwd_SmallBlock.....	1722
BWTInv_SmallBlock.....	1723
Generalized Interval Transformation Coding .....	1723
Special Parameters.....	1724
EncodeGITInitAlloc.....	1724
GITFree.....	1725
EncodeGITInit.....	1726
EncodeGITGetSize.....	1727
EncodeGIT.....	1728
DecodeGITInitAlloc.....	1732
DecodeGITInit.....	1732
DecodeGITGetSize.....	1733
DecodeGIT.....	1734
Move To Front Functions .....	1738
MTFInitAlloc.....	1739
MTFFree.....	1739
MTFInit.....	1740
MTFGetSize.....	1740
MTFFwd.....	1741
MTFInv.....	1742
Run Length Encoding Functions .....	1743
EncodeRLE.....	1743
DecodeRLE.....	1744
bzip2 Coding Functions .....	1746
EncodeRLEInitAlloc_BZ2.....	1746
RLEFree_BZ2.....	1747
EncodeRLEInit_BZ2.....	1747
RLEGetSize_BZ2.....	1748
EncodeRLE_BZ2.....	1748
EncodeRLEFlush_BZ2.....	1749
RLEGetInUseTable.....	1750
DecodeRLE_BZ2.....	1750

EncodeZ1Z2_BZ2.....	1751
DecodeZ1Z2_BZ2.....	1752
ReduceDictionary.....	1753
ExpandDictionary.....	1754
CRC32_BZ2.....	1754
EncodeHuffGetSize_BZ2.....	1755
EncodeHuffInit_BZ2.....	1756
EncodeHuffInitAlloc_BZ2.....	1757
EncodeHuffFree_BZ2.....	1758
PackHuffContext_BZ2.....	1758
EncodeHuff_BZ2.....	1759
DecodeHuffGetSize_BZ2.....	1760
DecodeHuffInit_BZ2.....	1761
DecodeHuffInitAlloc_BZ2.....	1762
DecodeHuffFree_BZ2.....	1762
UnpackHuffContext_BZ2.....	1763
DecodeHuff_BZ2.....	1764

## **Chapter 14: Data Integrity Functions**

GF(2 <sup>m</sup> ) Arithmetic Functions.....	1767
GFGetSize.....	1768
GFInit.....	1769
GFAdd.....	1770
GFSub.....	1771
GFMul.....	1772
GFDiv.....	1773
GFPow.....	1774
GFInv.....	1775
GFNeg.....	1776
GFLogAlpha.....	1777
GFExpAlpha.....	1778
Arithmetic Functions for Polynomials over GF(2 <sup>m</sup> ).....	1778

PolyGFGetSize.....	1779
PolyGFInit.....	1780
PolyGFSetCoeffs.....	1781
PolyGFSetDegree.....	1782
PolyGFCopy.....	1783
PolyGFGetRef.....	1784
PolyGFAdd.....	1785
PolyGFSub.....	1785
PolyGFMod.....	1786
PolyGFMul.....	1787
PolyGFDiv.....	1788
PolyGFShlC.....	1789
PolyGFShrC.....	1789
PolyGFIrreducible.....	1790
PolyGFPrimitive.....	1791
PolyGFValue.....	1792
PolyGFDerive.....	1792
PolyGFRoots.....	1793
PolyGFGCD.....	1794
Reed-Solomon Code Functions.....	1795
RS Encoder Functions.....	1795
RSEncodeGetSize.....	1795
RSEncodeInit.....	1796
RSEncodeGetBufferSize.....	1797
RSEncode.....	1798
RS Decoder Functions.....	1799
RSDecodeGetSize.....	1799
RSDecodeInit.....	1800
RSDecodeBMGetBufferSize, RSDecodeEEGetBufferSize.....	1801
RSDecodeBM, RSDecodeEE.....	1802

**Appendix A: Handling of Special Cases**

**Appendix B: Removed Functions**

**Bibliography**

**Glossary**

**Index**



# Overview

---

This document describes the structure, operation and functions of the Intel® Integrated Performance Primitives (Intel® IPP) for Intel® architecture that operate on one-dimensional signals. This is the first volume of the Intel IPP Reference Manual, which also comprises descriptions of Intel IPP for image and video processing (volume 2), operations on small matrices, 3D data processing and rendering (volume 3), and cryptography functions (volume 4). The Intel IPP software package supports many functions whose performance can be significantly enhanced on Intel architecture, particularly using the MMX™ technology, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), as well as Intel® Itanium® architecture.

The Intel IPP for signal processing software is a collection of low-overhead, high-performance operations performed on one-dimensional (1D) data arrays.

This manual explains the concepts of Intel IPP as well as specific data type definitions and operation models used in the signal processing domain and provides detailed descriptions of the Intel IPP signal processing functions.

This chapter introduces the Intel IPP software and explains the organization of this manual.

## About This Software

The Intel IPP for Intel architecture software enables taking advantage of the parallelism of the single-instruction, multiple-data (SIMD) instructions that make up the core of the MMX technology and Streaming SIMD Extensions. These technologies improve the performance of computation-intensive signal, image, and video processing applications. Use of Intel IPP primitive functions can help to drastically reduce development costs and accelerate time-to-market by eliminating the need of writing processor-specific code for computation intensive routines.

## Hardware and Software Requirements

The Intel IPP for Intel architecture software runs on personal computers that are based on processors using IA-32, Intel® 64 or IA-64 architecture and running Microsoft Windows\* OS, Linux\* OS, or Apple Mac OS\* X. Intel IPP can be integrated into the customer's application or library written in C or C++.

## Platforms Supported

Intel IPP for Intel architecture software runs on Windows\* OS, Linux\* OS, and Mac OS\* X platforms. The code and syntax used in this manual for function and variable declarations are written in the ANSI C style. However, versions of Intel IPP for different processors or operating systems may, of necessity, vary slightly.

## Cross-Architecture Alignment

Intel IPP has been designed to support application development on various Intel® architectures.

By providing a single cross-architecture application programming interface (API), Intel IPP allows software application repurposing and enables developers to port to unique features across Intel® processor-based desktop, server, and mobile platforms. Developers can write their code once in order to realize application performance over many processor generations.

## API Changes in Version 5.0

Starting from the version 5.0 Intel IPP has a limited compatibility with the versions 4.x or lower. To improve the library in several aspects the following changes have been done:

- some functions have been replaced by new functions with extended functionality;
- some functions have been changed to make API more consistent and handy;
- odd and unusable functions have been discarded;
- several complicated functions have been discarded from the Intel IPP, but their functionalities have been moved to the codec/sample level.

All these changes affect existent applications. [Appendix B](#) lists such functions for the signal processing functional domains and specifies the corresponding Intel IPP 5.0 functions to substitute for them. If an application calls functions listed in this table, then the source code must be modified.

Several functions have been moved to the newly created functional domains. These changes do not affect existent applications.

Additionally, certain modifications breaking binary compatibility have been made in the Intel IPP 5.0, for example, directory tree structure has been modified, the `ipp20` folder has been discarded. These changes also affect existent applications that must be at least rebuilt.

The version 5.1 as well as the next following versions has full backward compatibility with the version 5.0.



## Technical Support

Intel IPP provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: <http://www.intel.com/software/products/>.

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, and more (visit <http://www.intel.com/software/products/support>).

Registering your product entitles you to one-year technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing the following services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, or contact Intel, or seek product support, please visit:  
<http://www.intel.com/software/products/support>

## Intel IPP Code Samples

An extensive library of code samples and codecs have been implemented using the Intel IPP functions to help demonstrate the use of Intel IPP and to help accelerate the development of your application, components and codecs. The samples can be downloaded from <http://www3.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## About This Manual

This manual provides a background for the signal processing concepts used in the Intel IPP software as well as detailed descriptions of the Intel IPP for Intel architecture signal processing functions.

The Intel IPP functions are combined in groups by their functionality. Each group of functions is described in a separate chapter (chapters 3 through 14).

## Manual Organization

This manual contains the following chapters:

Chapter 1	<a href="#">Overview</a> (this chapter). Introduces the Intel IPP software, provides information on manual organization, and explains notational conventions.
Chapter 2	<a href="#">Intel® Integrated Performance Primitives Concepts</a> . Explains the basic concepts underlying signal processing in Intel IPP and describes the supported data formats and operation modes.
Chapter 3	<a href="#">Support Functions</a> . Describes functions used to manipulate the signals, such as <code>Copy</code> and <code>Set</code> . Also describes data conversion and memory allocation functions.
Chapter 4	<a href="#">Vector Initialization Functions</a> . Describes initialization and sample-generating functions.
Chapter 5	<a href="#">Essential Functions</a> . Describes vector manipulation functions.
Chapter 6	<a href="#">Filtering Functions</a> . Details filtering operations that use linear and non-linear filters.
Chapter 7	<a href="#">Transform Functions</a> . Describes domain transform functions (Fourier, Wavelet, Cosine).
Chapter 8	<a href="#">Speech Recognition Functions</a> . Describes functions used in the speech recognition applications.
Chapter 9	<a href="#">Speech Coding Functions</a> . Describes functions used as building blocks for implementing speech codecs.
Chapter 10	<a href="#">Audio Coding Functions</a> . Includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio codec, MP3 encoder and decoder, for implementation of a portable optimized MPEG-4 AAC Main profile decoder and a portable optimized MPEG-1, 2 Layer III encoder. Also includes Spectral Band Replication (SBR) functions as an audio coding bandwidth extension technology.
Chapter 11	<a href="#">String Functions</a> . Describes functions that operate on strings of symbols.
Chapter 12	<a href="#">Fixed-Accuracy Arithmetic Functions</a> . Describes Intel IPP fixed-accuracy transcendental mathematical real and complex functions of vector arguments. The functions are suitable for multimedia and signal processing in real time applications.
Chapter 13	<a href="#">Data Compression Functions</a> . Describes Intel IPP data compression functions that support a number of different conversion methods.

Chapter 14	<a href="#">Data Integrity Functions</a> . Describes Intel IPP data glitterati functions implementing error-correcting.
Appendix A	<a href="#">Handling of Special Cases</a> . Briefly describes handling of special cases by the Intel IPP functions.
Appendix B	<a href="#">Removed Functions</a> . Lists functions removed starting from Intel IPP version 5.0.

The manual also contains a [Glossary](#) of terms, a [Bibliography](#), and an [Index](#).

## Function Descriptions

In Chapters 3 through 14, each function is introduced by its short name (without the `ipp` prefix and modifiers) and a brief description of its purpose. This is followed by the function call sequence, definition of its parameters (arguments), and more detailed explanation of the function's purpose. The following sections are included in function description:

<i>Syntax</i>	Lists function prototypes
<i>Parameters</i>	Describes all the function parameters (arguments).
<i>Description</i>	Defines the function and describes the operation performed by the function. This section may also include the code examples and descriptive equations.
<i>Application Notes</i>	If present, describes any special information which application programmers or other users of the function need to know.
<i>Return Values</i>	Explains the value returned by the function. Most commonly, it lists error codes that the function returns.
<i>See Also</i>	If present, lists the names of functions which perform related tasks.

## Enumerators

The `IppStatus` constant enumerates the status values returned by the Intel IPP functions, indicating whether the operation is error-free. See section [Error Reporting](#) in this chapter for more information on the set of valid status values and corresponding error messages for signal processing functions.

The `IppCmpOp` enumeration defines the type of relational operator to be used by threshold functions:

```
typedef enum {  
    ippCmpLess,  
    ippCmpLessEq,  
    ippCmpLessEq,
```

```

        ippCmpEq,
        ippCmpGreaterEq,
        ippCmpGreater
    } IppCmpOp;

```

The `IppRoundMode` enumeration defines the rounding mode to be used by conversion functions:

```

typedef enum {
    ippRndZero,
    ippRndNear,
    ippRndFinancial
} IppRoundMode;

```

The `IppHintAlgorithm` enumeration defines the type of code to be used in some operations: faster but less accurate, or vice-versa, more accurate but slower. For more information on using this enumeration, see ["Hint Arguments"](#).

```

typedef enum {
    ippAlgHintNone,
    ippAlgHintFast,
    ippAlgHintAccurate
} IppHintAlgorithm;

```

The `IppCpuType` enumerates processor types returned by the `ippGetCpuType` function:

```

typedef enum {
    /* Enumeration: Processor: */
    ippCpuUnknown = 0x0, /* Intel(R) Pentium(R) processor */
    ippCpuPP, /* Pentium(R) processor with MMX(TM) technology */
    ippCpuPMX, /* Pentium(R) Pro processor */
    ippCpuPPR, /* Pentium(R) II processor */
    ippCpuPII, /* Pentium(R) III processor and Pentium(R) III Xeon(R) processor */
    ippCpuPIII, /* Pentium(R) 4 processor and Intel(R) Xeon(R) processor */
    ippCpuP4, /* Pentium(R) 4 processor with HT Technology */
    ippCpuP4HT, /* Pentium(R) 4 processor with Streaming SIMD Extensions 3 */
    ippCpuP4HT2, /* Intel(R) Centrino(TM) mobile technology */
    ippCpuCentrino, /* Intel(R) Core(TM) Solo processor */
    ippCpuCoreSolo, /* Intel(R) Core(TM) Duo processor */
    ippCpuCoreDuo, /* Intel(R) Itanium(R) processor */
    ippCpuITP = 0x10, /* Intel(R) Itanium(R) 2 processor */
    ippCpuITP2, /* Intel(R) 64 Instruction Set Architecture(ISA) */
    ippCpuEM64T = 0x20, /* Intel(R) Core(TM) 2 Duo processor */
    ippCpuC2D, /* Intel(R) Core(TM) 2 Quad processor */
    ippCpuC2Q, /* Intel(R) Core(TM) 2 processor with Intel(R) SSE4.1 */
    ippCpuPenryn,
}

```

---

```

    ippCpuBonnell,      /* Intel(R) Atom (TM) processor */
    ippCpuNehalem,     /* Intel (R) Core(TM) i7 processor
    ippCpuNext,
    ippCpuSSE   = 0x40, /* Processor supports Pentium(R) III
                        processor instruction set          */
    ippCpuSSE2,        /* Processor supports Streaming SIMD
                        Extensions 2 instruction set        */
    ippCpuSSE3,        /* Processor supports Streaming SIMD
                        Extensions 3 instruction set        */
    ippCpuSSSE3,       /* Processor supports Supplemental Streaming
                        SIMD Extensions 3 instruction set   */
    ippCpuSSE41,       /* Processor supports Streaming SIMD
                        Extensions 4.1 instruction set      */
    ippCpuSSE42,       /* Processor supports Streaming SIMD
                        Extensions 4.2 instruction set      */
    ippCpuAVX,         /* Processor supports Advanced Vector
                        Extensions instruction set          */
    ippCpuX8664 = 0x60, /* Processor supports 64 bit extension      */
} IppCpuType;

```

The `IppWinType` enumeration defines the type of window to be used by the FIR filter coefficient generating functions :

```

typedef enum {
    ippWinBartlett,
    ippWinBlackman,
    ippWinHamming,
    ippWinHann,
    ippWinRect
} IppWinType;

```

The `IppLZ77ComprLevel` enumeration defines the compression level to be used by the ZLIB data compression functions :

```

typedef enum {

    IppLZ77FastCompr,

    IppLZ77AverageCompr,

    IppLZ77BestCompr

} IppLZ77ComprLevel;

```

The `IppLZ77Chcksm` enumeration defines what algorithm is used to compute the checksum by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77NoChcksm,  
    IppLZ77Adler32,  
    IppLZ77CRC32  
} IppLZ77Chcksm;
```

The `IppLZ77Flush` enumeration defines what encoding mode is used by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77NoFlush,  
    IppLZ77SyncFlush,  
    IppLZ77FullFlush,  
    IppLZ77FinishFlush  
} IppLZ77Flush;
```

The `IppLZ77DeflateStatus` enumeration defines the encoding status that is used by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77StatusInit,  
    IppLZ77StatusLZ77Process,  
    IppLZ77StatusHuffProcess,  
    IppLZ77StatusFinal  
} IppLZ77DeflateStatus;
```

The `IppLZ77InflateStatus` enumeration defines the decoding status that is used by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77InflateStatusInit,  
    IppLZ77InflateStatusHuffProcess  
    IppLZ77InflateStatusLZ77Process,  
    IppLZ77InflateStatusFinal  
} IppLZ77InflateStatus;
```

The `IppLZ77HuffMode` enumeration defines the encoding mode that is used by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77UseFixed,  
    IppLZ77UseDynamic,  
    IppLZ77UsedStored  
} IppLZ77HuffMode;
```

The `IppInflateState` enumeration defines the decoding parameters that are used by the ZLIB data compression functions :

```
typedef struct IppInflateState {  
    const Ipp8u* pWindow;           // pointer to the sliding window  
                                     // (the dictionary for the LZ77 algorithm)  
    unsigned int winSize;           // size of the sliding window  
    unsigned int tableType;         // type of Huffman code tables  
                                     // (for example, 0 - tables for Fixed  
                                     // Huffman codes)  
    unsigned int tableBufferSize;   //(ENOUGH = 2048) * (sizeof(code) = 4) -  
                                     // sizeof(IppInflateState)  
} IppInflateState;
```

The `IppInflateMode` enumeration defines the decode mode that is used by the ZLIB data compression functions :

```
typedef enum {  
    ippTYPE,  
    ippLEN,  
    ippLENEXT  
} IppInflateMode;
```

The `IppGITStrategyHint` enumeration defines which strategy of encoding is used in some operations by the GIT data compression functions :

```
typedef enum {  
    ippGITNoStrategy,  
    ippGITLeftReorder,  
    ippGITRightReorder,  
    ippGITFixedOrder  
} IppGITStrategyHint;
```

## Audience for This Manual

The manual is intended for the developers of signal processing applications and libraries, as well as cross-domain applications. The audience is expected to be experienced in using C and to have a working knowledge of the vocabulary and principles of signal processing.

## Related Publications

For more information about signal processing concepts and algorithms, refer to the books and papers listed in the [Bibliography](#).

## Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code;
- Signal name conventions;
- Naming conventions for different items.



## Font Conventions

The following font conventions are used:

<code>THIS TYPE STYLE</code>	Used in the text for the Intel IPP constant identifiers. For example, <code>IPPI_MAX_64S</code> .
<code>This type style</code>	Mixed with the uppercase in structure names as in <code>IppLibraryVersion</code> ; also used in function names, code examples and call statements; for example, <code>void ippsFree()</code> .
<i>This type style</i>	Parameters in function prototypes and parameters description; for example: <i>value, srcStep</i> .

## Signal Name Conventions

In this manual, vectors and arrays are commonly used to represent a discrete 1D signal. The notation  $x(n)$  refers to a conceptual signal, while the notation  $x[n]$  refers to an actual vector. Both of these are annotated to indicate a specific finite range of values:

$x[n]$ ,  $0 \leq n < len$ .

Typically, the number of elements in vectors is denoted by *len*. Vector names contain square brackets as distinct from vector elements with current index *n*.

For example, the expression  $pDst[n] = pSrc[n] + val$  implies that each element  $pDst[n]$  of the vector  $pDst$  is computed for each *n* in the range from 0 to *len*-1. Special cases are regarded and described separately.

## Naming Conventions

The following naming conventions for different items are used by the Intel IPP software:

- Constant identifiers are in uppercase; for example, `IPP_MIN_64S`.
- All structures and enumerators, specific for the signal processing domain have the `Ipps` prefix, while those common for entire Intel IPP software have the `Ipp` prefix; for example, `IppsROI`, `IppLibraryVersion`.
- All names of the functions used for signal processing have the `ipps` prefix. In code examples, you can distinguish the Intel IPP interface functions from the application functions by this prefix.



**NOTE.** In this manual, the `ipps` prefix in function names is always used in the code examples. In the text, this prefix is usually omitted when referring to the function group.

- Each new part of a function name starts with an uppercase character, without underscore; for example, `ippsAddAllRowSum`.

For the detailed description of function name structure in Intel IPP, see [Function Naming](#).

# Intel® Integrated Performance Primitives Concepts

## 2

This chapter explains the purpose and structure of the Intel® Integrated Performance Primitives (Intel® IPP) software and looks over some of the basic concepts used in the signal processing part of Intel IPP. It also describes the supported data formats and operation modes, and defines function naming conventions in the manual.

## Basic Features

The Intel Integrated Performance Primitives, like other members of the Intel® Performance Libraries, is a collection of high-performance code that performs domain-specific operations. It is distinguished by providing a low-level, stateless interface.

Based on experience in developing and using Intel Performance Libraries, Intel IPP has the following major distinctive features:

- Intel IPP provides basic low-level functions for creating applications in several different domains, such as signal processing, image and video processing, operations on small matrices, and cryptography applications;
- Intel IPP functions follow the same interface conventions including uniform naming rules and similar composition of prototypes for primitives that refer to different application domains;
- Intel IPP functions use abstraction level which is best suited to achieve superior performance figures by the application programs.

To speed up performance, the functions of Intel IPP are optimized to use all benefits of Intel® architecture processors. Besides that, most of these functions do not use complicated data structures - this reduces overall execution overhead.

Intel IPP is well-suited for cross-platform applications. For example, the functions developed for IA-32 platform can be readily ported to Intel® Itanium® -based platforms.

## Function Naming

Naming conventions for the Intel IPP functions are similar for all covered domains.

Function names in Intel IPP have the following general format:

```
ipp<data-domain><name>_<datatype>[_<descriptor>](<parameters>);
```

The elements of this format are explained in the sections that follow.

## Data-Domain

The *data-domain* is a single character that expresses the subset of functionality to which a given function belongs. The current version of Intel IPP supports the following data-domains:

s	for signals (expected data type is a 1D signal);
i	for images and video (expected data type is a 2D image);
m	for matrices (expected data type is a matrix);
r	for realistic rendering functionality and 3D data processing (expected data type depends on supported rendering techniques);
g	for signals of the fixed length.

For example, function names that begin with `ipps` signify that respective functions are used for signal processing.

## Name

The *name* is an abbreviation for the core operation that the function really does, for example *Set*, *Copy*, followed in some cases by a function-specific modifier:

`<name> = <operation>[_modifier]`

This modifier, if present, denotes a slight modification or variation of the given function. For example, the modifier `CToC` in the function name `ippsFFTInv_CToC_32fc` signifies that the inverse fast Fourier transform operates on complex data, performing Complex-To-Complex (CToC) transform.

## Data Types

The *datatype* field indicates data types used by the function, in the following format:

`<bit depth><bit interpretation>`,

where

`bit depth = <1|8|16|32|64>`

and

bit interpretation<u| s|f>[ c]

Here *u* indicates “unsigned integer”, *s* indicates “signed integer”, *f* indicates “floating point”, and *c* indicates “complex”.

The current version of Intel IPP supports the data types of the source and destination for signal processing functions listed in Table 2-1 below.



**NOTE.** In the lists of function parameters, the `Ipp` prefix is written in the data type. For example, the 8-bit signed data is denoted as `Ipp8s` type. These Intel IPP-specific data types are defined in the respective library header files.

**Table 2-1 Data Types Supported by Intel IPP for Signal Processing**

Type	Usual C Type	Intel IPP Type
8u	unsigned char	Ipp8u
8s	signed char	Ipp8s
16u	unsigned short	Ipp16u
16s	signed short	Ipp16s
16sc	complex short	Ipp16sc
32u	unsigned int	Ipp32u
32s	signed int	Ipp32s
32f	float	Ipp32f
32fc	complex float	Ipp32fc
64s	__int64 (Windows*) or long long (Linux*)	Ipp64s
64f	double	Ipp64f
64fc	complex double	Ipp64fc

For functions that operate on a single data type, the *datatype* field contains only one of the values listed above.

If a function operates on source and destination signals that have different data types, the respective data type identifiers are listed in the function name in order of source and destination as follows:

<datatype> = <src1Depth>[src2Depth][dstDepth]

For example, the function `ippsDotProd_16s16sc_Sfs` computes the dot product of 16-bit short and 16-bit complex short source vectors and stores the result in a 16-bit complex short destination vector. The *dstDepth* modifier is not present in the name because the second operand and the result are of the same type. The result is scaled and saturated.

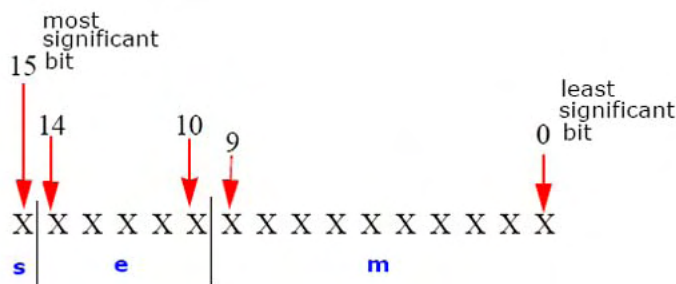
There are several data types, namely `24u`, `24s` and `16f`, that are not supported by Intel IPP but can be readily converted to the supported data types for further processing by the library functions.

For the unsigned `24u` data, each vector element consists of three consecutive bytes represented as `Ipp8u` data types. It has a little-endian byte order when a lower order byte is at the lower address. These data may be converted to and from `32u` or `32f` data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

For the signed `24s` data, each vector element consists of three consecutive bytes represented as `Ipp8u` data types. It has a little-endian byte order when a lower order byte is at the lower address. The sign is represented by the most significant bit of the highest order byte. These data may be converted to and from `32s` or `32f` data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

For the `16f` format, 16-bit floating point data (*half type*) can represent positive and negative numbers, whose magnitude is between roughly  $6.1e^{-5}$  and  $6.5e^4$ , with a relative error of  $9.8e^{-4}$ ; numbers smaller than  $6.1e^{-5}$  can be represented with an absolute error of  $6.0e^{-8}$ . All integers from  $-2048$  to  $+2048$  can be represented exactly.

The figure below illustrates the bit-layout for a half number:



**s** is the sign-bit, **e** is the exponent, and **m** is the significand.

These data may be converted to and from `16s` and `32f` data types by using the appropriate flavors of the Intel IPP function `ippsConvert`.

## Descriptor

The *descriptor* field further describes the data associated with the operation. It may contain implied parameters and/or indicate additional required parameters. To minimize the number of code branches in the function and thus reduce potentially unnecessary execution overhead, most of the general functions were split into separate primitive functions, with some of their parameters entering the primitive function name as descriptors.

However, some functions may still have parameters that determine internal operations (for example, `ippThreshold`).

The following descriptors are used in signal processing functions:

I	Operation is in-place (default is not-in-place).
Sfs	Saturation and fixed scaling mode (default is saturation and no scaling).
Dx	Signal is x-dimensional (default is D1).
L	One pointer is used for each row (in D2).
P	Operation is performed for the specified number of vectors.

The abbreviations of descriptors in function names are always present in alphabetized order.

Not all functions have every abbreviation listed above. For example, in-place mode makes no sense for a copy operation.

## Parameters

The *parameters* field specifies the function parameters (arguments).

The order of parameters is as follows:

- All source operands. Constants follow vectors.
- All destination operands. Constants follow vectors.
- Other, operation-specific parameters.

A parameter name has the following conventions:

- All parameters defined as pointers start with *p*, defined as double pointers start with *pp*, for example, *pPhase*, *pSrc*, *ppState*; and parameters defined as values start with a lowercase letter, for example, *val*, *src*, *srcLen*.
- Each new part of a parameter name starts with an uppercase character, without underscore; for example, *pSrc*, *lenSrc*, *pDlyLine*.
- Each parameter name specifies its functionality. Source parameters are named *pSrc* or *src*, sometimes followed by names or numbers, for example, *pSrc2*, *srcLen*. Output parameters are named *pDst* or *dst* followed by names or numbers, for example, *pDst2*, *dstLen*. For in-place operations, the input/output parameter contains the name *pSrcDst*.

## Structures and Enumerators

This section describes the structures and enumerators used by Intel IPP for signal processing.

### Library Version Structure

The `IppLibraryVersion` structure describes the current Intel IPP software version. The main fields of this structure are:

- integer fields *major* and *minor*, containing version numbers;
- string field *Name*, containing the Intel IPP version name, for example, "ippsa6";
- string field *Version*, containing the version description, for example, "v0.0 Alpha 0.0.3.3".

### Complex Data Structures

Complex numbers in Intel IPP are described by the structures that contains two numbers of the respective data type which are real and imaginary parts of the complex number. For example, a single precision complex number is described by the `Ipp32fc` structure as follows:

```
typedef struct {
    Ipp32f  re;
    Ipp32f  im;
} Ipp32fc;
```

The following complex data types are defined: `Ipp16sc`, `Ipp32fc`, `Ipp64fc`.



## Function Context Structures

Some Intel IPP functions use special structures to store function-specific (context) information. For example, the `IppsFFTSpec` structure stores twiddle factors and bit reverse indexes needed in the fast Fourier transform.

Two different kinds of structures are used: specification structures that are not modified during function operation - they have the suffix `Spec` in their names, and state structures that are modified during operation - they have the suffix `State` in their names.

These context-related structures are not defined in public headers, and their fields are not accessible. It was done because the function context interpretation is processor dependent. Thus, the user has no possibilities to modify these structures or to create a function context as an automatic variable.

## Enumerators

The `IppStatus` constant enumerates the status values returned by the Intel IPP functions, indicating whether the operation is error-free. See section [Error Reporting](#) in this chapter for more information on the set of valid status values and corresponding error messages for signal processing functions.

The `IppCmpOp` enumeration defines the type of relational operator to be used by threshold functions:

```
typedef enum {  
    ippCmpLess,  
    ippCmpLessEq,  
    ippCmpEq,  
    ippCmpGreaterEq,  
    ippCmpGreater  
} IppCmpOp;
```

The `IppRoundMode` enumeration defines the rounding mode to be used by conversion functions:

```
typedef enum {  
    ippRndZero,  
    ippRndNear,  
    ippRndFinancial  
} IppRoundMode;
```

The `IppHintAlgorithm` enumeration defines the type of code to be used in some operations: faster but less accurate, or vice-versa, more accurate but slower. For more information on using this enumeration, see ["Hint Arguments"](#).

```
typedef enum {
    ippAlgHintNone,
    ippAlgHintFast,
    ippAlgHintAccurate
} IppHintAlgorithm;
```

The `IppCpuType` enumerates processor types returned by the `ippGetCpuType` function:

```
typedef enum {
    /* Enumeration: Processor: */
    ippCpuUnknown = 0x0,
    ippCpuPP, /* Intel(R) Pentium(R) processor */
    ippCpuPMX, /* Pentium(R) processor with MMX(TM) technology */
    ippCpuPPR, /* Pentium(R) Pro processor */
    ippCpuPII, /* Pentium(R) II processor */
    ippCpuPIII, /* Pentium(R) III processor and Pentium(R)III Xeon(R) processor */
    ippCpuP4, /* Pentium(R) 4 processor and Intel(R) Xeon(R) processor */
    ippCpuP4HT, /* Pentium(R) 4 processor with HT Technology */
    ippCpuP4HT2, /* Pentium(R) 4 processor with Streaming SIMD Extensions 3 */
    ippCpuCentrino, /* Intel(R) Centrino(TM) mobile technology */
    ippCpuCoreSolo, /* Intel(R) Core(TM) Solo processor */
    ippCpuCoreDuo, /* Intel(R) Core(TM) Duo processor */
    ippCpuITP = 0x10, /* Intel(R) Itanium(R) processor */
    ippCpuITP2, /* Intel(R) Itanium(R) 2 processor */
    ippCpuEM64T = 0x20, /* Intel(R) 64 Instruction Set Architecture(ISA) */
    ippCpuC2D, /* Intel(R) Core(TM) 2 Duo processor */
    ippCpuC2Q, /* Intel(R) Core(TM) 2 Quad processor */
    ippCpuPenryn, /* Intel(R) Core(TM) 2 processor with Intel(R) SSE4.1 */
    ippCpuBonnell, /* Intel(R) Atom (TM) processor */
    ippCpuNehalem, /* Intel (R) Core(TM) i7 processor */
    ippCpuNext,
    ippCpuSSE = 0x40, /* Processor supports Pentium(R) III processor instruction set */
    ippCpuSSE2, /* Processor supports Streaming SIMD Extensions 2 instruction set */
    ippCpuSSE3, /* Processor supports Streaming SIMD Extensions 3 instruction set */
    ippCpuSSSE3, /* Processor supports Supplemental Streaming SIMD Extensions 3 instruction set */
    ippCpuSSE41, /* Processor supports Streaming SIMD Extensions 4.1 instruction set */
}
```

```
    ippCpuSSE42,          /* Processor supports Streaming SIMD
                           Extensions 4.2 instruction set          */
    ippCpuAVX,            /* Processor supports Advanced Vector
                           Extensions instruction set              */
    ippCpuX8664 = 0x60, /* Processor supports 64 bit extension      */
} IppCpuType;
```

The `IppWinType` enumeration defines the type of window to be used by the FIR filter coefficient generating functions :

```
typedef enum {
    ippWinBartlett,
    ippWinBlackman,
    ippWinHamming,
    ippWinHann,
    ippWinRect
} IppWinType;
```

The `IppLZ77ComprLevel` enumeration defines the compression level to be used by the ZLIB data compression functions :

```
typedef enum {
    IppLZ77FastCompr,
    IppLZ77AverageCompr,
    IppLZ77BestCompr
} IppLZ77ComprLevel;
```

The `IppLZ77Chcksm` enumeration defines what algorithm is used to compute the checksum by the ZLIB data compression functions :

```
typedef enum {
    IppLZ77NoChcksm,
    IppLZ77Adler32,
    IppLZ77CRC32
} IppLZ77Chcksm;
```

The `IppLZ77Flush` enumeration defines what encoding mode is used by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77NoFlush,  
    IppLZ77SyncFlush,  
    IppLZ77FullFlush,  
    IppLZ77FinishFlush  
} IppLZ77Flush;
```

The `IppLZ77DeflateStatus` enumeration defines the encoding status that is used by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77StatusInit,  
    IppLZ77StatusLZ77Process,  
    IppLZ77StatusHuffProcess,  
    IppLZ77StatusFinal  
} IppLZ77DeflateStatus;
```

The `IppLZ77InflateStatus` enumeration defines the decoding status that is used by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77InflateStatusInit,  
    IppLZ77InflateStatusHuffProcess  
    IppLZ77InflateStatusLZ77Process,  
    IppLZ77InflateStatusFinal  
} IppLZ77InflateStatus;
```

The `IppLZ77HuffMode` enumeration defines the encoding mode that is used by the ZLIB data compression functions :

```
typedef enum {  
    IppLZ77UseFixed,  
    IppLZ77UseDynamic,  
    IppLZ77UsedStored  
} IppLZ77HuffMode;
```

The `IppInflateState` enumeration defines the decoding parameters that are used by the ZLIB data compression functions :

```
typedef struct IppInflateState {  
    const Ipp8u* pWindow;          // pointer to the sliding window  
                                    // (the dictionary for the LZ77 algorithm)  
    unsigned int winSize;          // size of the sliding window  
    unsigned int tableType;        // type of Huffman code tables  
                                    // (for example, 0 - tables for Fixed  
                                    // Huffman codes)  
    unsigned int tableBufferSize; // (ENOUGH = 2048) * (sizeof(code) = 4) -  
                                    // sizeof(IppInflateState)  
} IppInflateState;
```

The `IppInflateMode` enumeration defines the decode mode that is used by the ZLIB data compression functions :

```
typedef enum {  
    ippTYPE,  
    ippLEN,  
    ippLENEXT  
} IppInflateMode;
```

The `IppGITStrategyHint` enumeration defines which strategy of encoding is used in some operations by the GIT data compression functions :

```
typedef enum {
    ippGITNoStrategy,
    ippGITLeftReorder,
    ippGITRightReorder,
    ippGITFixedOrder
} IppGITStrategyHint;
```

## Data Ranges

The range of values that can be represented by each data type lies between the lower and upper bounds. The following table lists data ranges and constant identifiers used in Intel IPP to denote the respective range bounds:

**Table 2-2 Data Types and Ranges**

Data Type	Lower Bound Identifier Value	Upper Bound Identifier Value		
8s	IPP_MIN_8S	-128	IPP_MAX_8S	127
8u		0	IPP_MAX_8U	255
16s	IPP_MIN_16S	-32768	IPP_MAX_16S	32767
16u		0	IPP_MAX_16U	65535
32s	IPP_MIN_32S	-2 <sup>31</sup>	IPP_MAX_32S	2 <sup>31</sup> -1
32u		0	IPP_MAX_32U	2 <sup>32</sup> -1
32f †	IPP_MINABS_32F	1.175494351e <sup>-38</sup>	IPP_MAXABS_32F	3.402823466e <sup>38</sup>
64s	IPP_MIN_64S	-2 <sup>63</sup>	IPP_MAX_64S	2 <sup>63</sup> -1
64f †	IPP_MINABS_64F	2.2250738585072014e <sup>-38</sup>	IPP_MAXABS_64F	1.7976931348623158e <sup>308</sup>

† The range for absolute values.

## Data Alignment

Intel IPP is built using the compiler option `/Zp16`, which aligns the structure fields on the field size or 16 bytes if the size is greater than 16.

Intel IPP allows also to align the allocated memory pointer on 32 bytes by means of using the function `ippsMalloc`.

## Integer Scaling

Some signal processing functions operating on integer data use scaling of the internally computed output results by the integer *scaleFactor*, which is specified as one of the function parameters. These functions have the *Sfs* descriptor in their names.

The scale factor can be negative, positive, or zero. Scaling is applied because internal computations are generally performed with a higher precision than the data types used for input and output signals.



**NOTE.** The result of integer operations is always saturated to the destination data type range even when scaling is used.

Scaling of an integer result is done by multiplying the output vector values by  $2^{-scaleFactor}$  before the function returns. This helps retain either the output data range or its precision. Usually the scaling with a positive factor is performed by the shift operation. The result is rounded off to the nearest integer number.

For example, the integer *Ipp16s* result of the square operation *ippsSqr* for the input value 200 is equal to 32767 instead of 40000, that is, the result is saturated and the exact value can not be restored.

The scaling of the output value with the factor *scaleFactor* = 1 yields the result 20000, which is not saturated, and the exact value can be restored as  $20000 * 2$ . Thus, the output data range is retained.

The following example shows how the precision can be partially retained by means of scaling.

The integer square root operation *ippsSqrt* (without scaling) for the input value 2 gives the result equal to 1 instead of 1.414. Scaling of the internally computed output value with the factor *scaleFactor* = -3 gives the result 11, and permits to restore the more precise value as  $11 * 2^{-3} = 1.375$ .

## Error Reporting

The Intel IPP functions return the status of the performed operation to report errors and warnings to the calling program. Thus, it is up to the application to perform error-related actions and/or recover from the error. The last value of the error status is not stored, and the user is to decide whether to check it or not as the function returns. The status values are of *IppStatus* type and are global constant integers.

Table 2-3 below lists status codes and corresponding messages reported by Intel IPP for signal processing.

**Table 2-3 Error Status Values and Messages**

Status	Message
ippStsCpuNotSupportedErr	The target cpu is not supported
ippStsPointAtInfinity	Point at infinity is detected
ippStsI18nUnsupportedErr	Internationalization (i18n) is not supported
ippStsI18nMsgCatalogOpenErr	Message Catalog cannot be opened, for extended information use errno in Linux and GetLastError in Windows
ippStsI18nMsgCatalogCloseErr	Message Catalog cannot be closed, for extended information use errno in Linux and GetLastError in Windows
ippStsUnknownStatusCodeErr	Unknown Status Code
ippStsOFBSizeErr	Wrong value for crypto OFB block size
ippStsLzoBrokenStreamErr	LZO safe decompression function cannot decode LZO stream
ippStsRoundModeNotSupportedErr	Unsupported round mode
ippStsMaxLenHuffCodeErr	Huff: Max length of Huffman code is more than expected one
ippStsCodeLenTableErr	Huff: Invalid codeLenTable
ippStsFreqTableErr	Huff: Invalid freqTable
ippStsRegExpOptionsErr	RegExp: Options for pattern are incorrect
ippStsRegExpErr	RegExp: The structure pRegExpState contains wrong data
ippStsRegExpMatchLimitErr	RegExp: The match limit has been exhausted
ippStsRegExpQuantifierErr	RegExp: wrong quantifier
ippStsRegExpGroupingErr	RegExp: wrong grouping
ippStsRegExpBackRefErr	RegExp: wrong back reference
ippStsRegExpChClassErr	RegExp: wrong character class
ippStsRegExpMetaChErr	RegExp: wrong metacharacter
ippStsMP3FrameHeaderErr	Error in fields IppMP3FrameHeader structure
ippStsMP3SideInfoErr	Error in fields IppMP3SideInfo structure
ippStsAacPrgNumErr	AAC: Invalid number of elements for one program



---

<code>ippStsAacSectCbErr</code>	AAC: Invalid section codebook
<code>ippStsAacSfValErr</code>	AAC: Invalid scalefactor value
<code>ippStsAacCoefValErr</code>	AAC: Invalid quantized coefficient value
<code>ippStsAacMaxSfbErr</code>	AAC: Invalid coefficient index
<code>ippStsAacPredSfbErr</code>	AAC: Invalid predicted coefficient index
<code>ippStsAacPlsDataErr</code>	AAC: Invalid pulse data attributes
<code>ippStsAacGainCtrErr</code>	AAC: Gain control not supported
<code>ippStsAacSectErr</code>	AAC: Invalid number of sections
<code>ippStsAacTnsNumFiltErr</code>	AAC: Invalid number of TNS filters
<code>ippStsAacTnsLenErr</code>	AAC: Invalid TNS region length
<code>ippStsAacTnsOrderErr</code>	AAC: Invalid order of TNS filter
<code>ippStsAacTnsCoefResErr</code>	AAC: Invalid bit-resolution for TNS filter coefficients
<code>ippStsAacTnsCoefErr</code>	AAC: Invalid TNS filter coefficients
<code>ippStsAacTnsDirectErr</code>	AAC: Invalid TNS filter direction
<code>ippStsAacTnsProfileErr</code>	AAC: Invalid TNS profile
<code>ippStsAacErr</code>	AAC: Internal error
<code>ippStsAacBitOffsetErr</code>	AAC: Invalid current bit offset in bitstream
<code>ippStsAacAdtsSyncWordErr</code>	AAC: Invalid ADTS syncword
<code>ippStsAacSmplRateIdxErr</code>	AAC: Invalid sample rate index
<code>ippStsAacWinLenErr</code>	AAC: Invalid window length (not short or long)
<code>ippStsAacWinGrpErr</code>	AAC: Invalid number of groups for current window length
<code>ippStsAacWinSeqErr</code>	AAC: Invalid window sequence range
<code>ippStsAacComWinErr</code>	AAC: Invalid common window flag
<code>ippStsAacStereoMaskErr</code>	AAC: Invalid stereo mask
<code>ippStsAacChanErr</code>	AAC: Invalid channel number
<code>ippStsAacMonoStereoErr</code>	AAC: Invalid mono-stereo flag
<code>ippStsAacStereoLayerErr</code>	AAC: Invalid this Stereo Layer flag
<code>ippStsAacMonoLayerErr</code>	AAC: Invalid this Mono Layer flag
<code>ippStsAacScalableErr</code>	AAC: Invalid scalable object flag

<code>ippStsAacObjTypeErr</code>	AAC: Invalid audio object type
<code>ippStsAacWinShapeErr</code>	AAC: Invalid window shape
<code>ippStsAacPcmModeErr</code>	AAC: Invalid PCM output interleaving indicator
<code>ippStsVLCUsrTblHeaderErr</code>	VLC: Invalid header inside table
<code>ippStsVLCUsrTblUnsupportedFmtErr</code>	VLC: Unsupported table format
<code>ippStsVLCUsrTblEscAlgTypeErr</code>	VLC: Unsupported Ecs-algorithm
<code>ippStsVLCUsrTblEscCodeLengthErr</code>	VLC: Incorrect Esc-code length inside table header
<code>ippStsVLCUsrTblCodeLengthErr</code>	VLC: Unsupported code length inside table
<code>ippStsVLCInternalTblErr</code>	VLC: Invalid internal table
<code>ippStsVLCInputDataErr</code>	VLC: Invalid input data
<code>ippStsVLCAACEscCodeLengthErr</code>	VLC: Invalid AAC-Esc code length
<code>ippStsIncorrectLSPErr</code>	Incorrect Linear Spectral Pair values
<code>ippStsNoRootFoundErr</code>	No roots are found for equation
<code>ippStsLengthErr</code>	Wrong value of string length
<code>ippStsFBankFreqErr</code>	Incorrect value of the filter bank frequency parameter
<code>ippStsFBankFlagErr</code>	Incorrect value of the filter bank parameter
<code>ippStsFBankErr</code>	Filter bank is not correctly initialized
<code>ippStsNegOccErr</code>	Negative occupation count
<code>ippStsCdbkFlagErr</code>	Incorrect value of the codebook flag parameter
<code>ippStsSVDConvErr</code>	No convergence of SVD algorithm
<code>ippStsToneMagnErr</code>	Tone magnitude is less than or equal to zero
<code>ippStsToneFreqErr</code>	Tone frequency is negative, or greater than or equal to 0.5
<code>ippStsTonePhaseErr</code>	Tone phase is negative, or greater than or equal to $2\pi$
<code>ippStsTrnglMagnErr</code>	Triangle magnitude is less than or equal to zero
<code>ippStsTrnglFreqErr</code>	Triangle frequency is negative, or greater than or equal to 0.5
<code>ippStsTrnglPhaseErr</code>	Triangle phase is negative, or greater than or equal to $2\pi$

---

<code>ippStsTrnglAsymErr</code>	Triangle asymmetry is less than -PI, or greater than or equal to PI
<code>ippStsHugeWinErr</code>	Incorrect size of the Kaiser window
<code>ippStsJaehneErr</code>	Magnitude value is negative
<code>ippStsStepErr</code>	Step value is not valid
<code>ippStsDlyLineIndexErr</code>	Invalid value of the delay line sample index
<code>ippStsStrideErr</code>	Stride value is less than the row length
<code>ippStsEpsValErr</code>	Negative epsilon value error
<code>ippStsScaleRangeErr</code>	Scale bounds are out of range
<code>ippStsThresholdErr</code>	Invalid threshold bounds
<code>ippStsWtOffsetErr</code>	Invalid offset value of the wavelet filter
<code>ippStsAnchorErr</code>	Anchor point is outside the mask
<code>ippStsMaskSizeErr</code>	Invalid mask size
<code>ippStsShiftErr</code>	Shift value is less than zero
<code>ippStsSampleFactorErr</code>	Sampling factor is less than or equal to zero
<code>ippStsSamplePhaseErr</code>	Phase value is out of range, $0 \leq phase < factor$
<code>ippStsFIRMRFactorErr</code>	MR FIR sampling factor is less than or equal to zero
<code>ippStsFIRMRPhaseErr</code>	MR FIR sampling phase parameter is negative, or greater than or equal to the sampling factor
<code>ippStsRelFreqErr</code>	Relative frequency value is out of range
<code>ippStsFIRLenErr</code>	Length of a FIR filter is less than or equal to zero
<code>ippStsIIROrderErr</code>	Order of an IIR filter is less than or equal to zero
<code>ippStsResizeFactorErr</code>	Resize factor(s) is less than or equal to zero
<code>ippStsDivByZeroErr</code>	An attempt to divide by zero
<code>ippStsInterpolationErr</code>	Invalid interpolation mode
<code>ippStsMirrorFlipErr</code>	Invalid flip mode
<code>ippStsMoment00ZeroErr</code>	Moment value M(0,0) is too small to continue calculations

<code>ippStsThreshNegLevelErr</code>	Negative value of the level in the threshold operation
<code>ippStsContextMatchErr</code>	Context parameter doesn't match the operation
<code>ippStsFftFlagErr</code>	Invalid value of the FFT flag parameter
<code>ippStsFftOrderErr</code>	Invalid value of the FFT order parameter
<code>ippStsMemAllocErr</code>	Not enough memory allocated for the operation
<code>ippStsNullPtrErr</code>	Null pointer error
<code>ippStsSizeErr</code>	Wrong value of the data size
<code>ippStsBadArgErr</code>	Function argument/parameter is bad
<code>ippStsErr</code>	Unknown/unspecified error
<code>ippStsNoErr</code>	No error, it's OK
<code>ippStsNoOperation</code>	No operation has been executed
<code>ippStsMisalignedBuf</code>	Misaligned pointer in operation in which it must be aligned
<code>ippStsSqrtNegArg</code>	Negative value(s) of the argument in the function Sqrt
<code>ippStsInvByZero</code>	Inf result. Zero value was met by InvThresh with zero level
<code>ippStsEvenMedianMaskSize</code>	Even size of the Median Filter mask was replaced by the odd one
<code>ippStsDivByZero</code>	Zero value(s) of the divisor in the function Div
<code>ippStsLnZeroArg</code>	Zero value(s) of the argument in the function Ln
<code>ippStsLnNegArg</code>	Negative value(s) of the argument in the function Ln
<code>ippStsNanArg</code>	Not a Number (NaN) argument value warning
<code>ippStsResFloor</code>	All result values are floored
<code>ippStsOverflow</code>	Overflow occurred in the operation
<code>ippStsZeroOcc</code>	Zero occupation count
<code>ippStsUnderflow</code>	Underflow occurred in the operation
<code>ippStsSingularity</code>	Singularity occurred in the operation

---

<code>ippStsDomain</code>	Argument is out of the function domain
<code>ippStsNotIntelCpu</code>	The target cpu is not Genuine Intel
<code>ippStsCpuMismatch</code>	The library for given cpu cannot be set
<code>ippStsNotIppFunctionFound</code>	Application does not contain IPP functions calls
<code>ippStsDllNotFoundBestUsed</code>	The newest version of IPP DLL's not found by dispatcher
<code>ippStsNoOperationInDll</code>	The function does nothing in the dynamic version of the library
<code>ippStsOvermuchStrings</code>	Number of destination strings is more than expected
<code>ippStsOverlongString</code>	Length of one of the destination strings is more than expected
<code>ippStsSrcSizeLessExpected</code>	DC: The size of source buffer is less than expected one
<code>ippStsDstSizeLessExpected</code>	DC: The size of destination buffer is less than expected one
<code>ippStsNotSupportedCpu</code>	The CPU is not supported
<code>ippStsUnkhownCacheSize</code>	The CPU is supported, but the size of the cache is unknown.

---

The status codes ending with `Err` (except for the `ippStsNoErr` status) indicate an error; the integer values of these codes are negative. When an error occurs, the function execution is interrupted. All other status codes indicate warnings. When a specific case is encountered, the function execution is completed and the corresponding warning status is returned.

For example, if the integer function `ippsDiv_8u` meets an attempt to divide a positive value by zero, the function execution is not aborted. The result of the operation is set to the maximum value that can be represented by the source data type, and the function returns the warning status `ippStsDivByZero`. This is the case for the vector-vector operation `ippsDiv`. For the vector-scalar division operation `ippsDivC`, the function behavior is different: if the constant divisor is zero, then the function stops execution and returns immediately with the error status `ippStsDivByZeroErr`.

## Code Examples

The manual contains a number of code examples that use the Intel IPP functions and serve to demonstrate both some particular features of the primitives and how the primitives can be called. Many of these code examples output result data together with status code and associated messages in case when error or warning condition was met.

To keep the example code simpler, special definitions of print statements are used that get output strings look exactly the way is needed for better representation of both real and complex results of different format, as well as print status codes and messages.

The code definitions given below make it possible to build the examples contained in the manual by straightforward copying and pasting the example code fragments.

```

/// the functions providing simple output of the result
/// they are for real and complex data

#define genPRINT(TYPE,FMT) \

void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus st ) { \

    int n; \

    if( st > ippStsNoErr ) \

        printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \

    else if( st < ippStsNoErr ) \

        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \

    printf( " %s ", msg ); \

    for( n=0; n<len; ++n ) printf( FMT, buf[n] ); \

    printf("\n" ); \

}

#define genPRINTcplx(TYPE,FMT) \

void printf_##TYPE(const char* msg, Ipp##TYPE* buf, int len, IppStatus st ) { \

    int n; \

    if( st > ippStsNoErr ) \

        printf( "-- warning %d, %s\n", st, ippGetStatusString( st )); \

    else if( st < ippStsNoErr ) \

        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \

    printf( " %s ", msg ); \

    for( n=0; n<len; ++n ) printf( FMT, buf[n].re, buf[n].im ); \

    printf("\n" ); \

}

genPRINT( 64f, " %f" )
genPRINT( 32f, " %f" )
genPRINT( 16s, " %d" )

```

```
genPRINTcplx( 64fc, " {%f,%f}" )  
genPRINTcplx( 32fc, " {%f,%f}" )  
genPRINTcplx( 16sc, " {%d,%d}"  
)
```



# Support Functions

This chapter describes Intel® IPP support functions that are used to:

- retrieve information about the current Intel IPP software version
- allocate and free memory that is needed for the operation of other Intel IPP functions
- retrieve information about the processor and perform specific auxiliary operations
- functions for internationalization.

The full list of these functions is given in Table 3-1 below.

**Table 3-1 Intel IPP Support Functions**

Function Base Name	Operation
Version Information Functions	
<code>ippsGetLibVersion</code>	Returns information about the active library version.
Memory Allocation Functions	
<code>ippsMalloc</code>	Allocates memory aligned to 32-byte boundary.
<code>ippsFree</code>	Frees memory allocated by the function <code>ippsMalloc</code> .
Common Functions	
<code>ippGetStatusString</code>	Translates a status code into a message.
<code>ippGetCpuType</code>	Returns a processor type.
<code>ippGetCpuClocks</code>	Returns a current value of the time stamp counter (TSC) register.
<code>ippGetCpuFreqMhz</code>	Estimates the processor frequency.
<code>ippGetCpuFeatures</code>	Retrieves the processor features.
<code>ippGetNumCoresOnDie</code>	Returns the number of cores.
<code>ippGetMaxCacheSizeB</code>	Returns maximum size of the L2 and L3 caches of the processor.
<code>ippSetFlushToZero</code>	Enables or disables flush-to-zero mode.
<code>ippSetDenormAreZeros</code>	Enables or disables denormals-are-zero mode.

Function Base Name	Operation
<code>ippAlignPtr</code>	Aligns a pointer to the specified number of bytes.
<code>ippSetNumThreads</code>	Sets the number of threads in the multithread environment.
<code>ippGetNumThreads</code>	Returns the number of existing threads in the multithread environment.
<code>ippMalloc</code>	Allocates memory aligned to 32-byte boundary.
<code>ippFree</code>	Frees memory allocated by the function <code>ippMalloc</code> .
Dispatcher Control Functions	
<code>ippStaticInit</code>	Automatically selects the most appropriate static code.
<code>ippInit</code>	Automatically initializes the version of the library code most appropriate for the current processor type.
<code>ippStaticInitCpu</code>	<b>THIS FUNCTION IS DEPRECATED.</b> Initializes the specified version of the static library code.
<code>ippInitCpu</code>	Initializes the specified version of the library code.
<code>ippEnableCpu</code>	Allows dispatching the processor-specific library
Internationalization Functions	
<code>ippMessageCatalogOpenI18n</code>	Opens an i18n message catalog.
<code>ippMessageCatalogCloseI18n</code>	Closes the opened i18n message catalog.
<code>ippGetMessageStatusI18n</code>	Returns the translation of the status message.
<code>ippStatusToMessageIdI18n</code>	Transforms an Intel IPP status code to the message ID for the i18n message catalog.

## Version Information Functions

These functions return the version number and other information about the active Intel IPP software.

### ippsGetLibVersion

*Returns information about the active version of Intel IPP signal processing software.*

---

#### Syntax

```
const IppLibraryVersion* ippsGetLibVersion(void);
```

#### Description

The function `ippsGetLibVersion` is declared in the `ipps.h` file. This function returns a pointer to a static data structure `IppLibraryVersion` that contains information about the current version of the Intel IPP software for signal processing. There is no need for you to release memory referenced by the returned pointer, as it points to a static variable. The following fields of the `IppLibraryVersion` structure are available:

*major* - the major number of the current library version.

*minor* - the minor number of the current library version.

*majorBuild* - the number of builds of the major version.

*build* current build number.

*Name* the name of the current library version.

*Version* is the library version string.

*BuildDate* is the library version actual build date.

For example, if the library version is "v1.2 Beta", library name is "ippsm6", and build date is "Jul 20 99", then the fields in this structure are set as:

*major* = 1, *minor* = 2, *Name* = "ippsm6", *Version* = "v1.2 Beta", *BuildDate* = "Jul 20 99"

Example 3-1 shows how to use the function `ippsGetLibVersion`.

## Example 3-1 Using the Function `ippsGetLibVersion`

```
void libinfo(void) {
    const IppLibraryVersion*
    lib = ippsGetLibVersion();

    printf("%s %s %d.%d.%d.%d\n",
        lib->Name, lib->Version,

        lib->major,
        lib->minor, lib->majorBuild, lib->build);
}
```

Output:

```
    ippsa6 v0.0 Alpha
0.0.5.5
```



**NOTE.** Each sub-library that is used in the signal processing domain has its own similar function to retrieve information about the active library version. These functions are: `ippGetLibVersion`, `ippacGetLibVersion`, `ippchGetLibVersion`, `ipppdcGetLibVersion`, `ippdiGetLibVersion`, `ipppdgenGetLibVersion`, `ipppscGetLibVersion`, `ipppsrGetLibVersion`, `ipppvmGetLibVersion`. They are declared in the following header files: `ippcore.h`, `ippac.h`, `ippch.h`, `ipppdc.h`, `ippdi.h`, `ippps.h`, `ipppsc.h`, `ipppsr.h`, `ipppvm.h`, respectively, and have the same interface as the function `ippsGetLibVersion`.

---

## Memory Allocation Functions

This section describes the Intel IPP signal processing functions that allocate aligned memory blocks for data of required type or free the previously allocated memory. The size of allocated memory is specified by the number of allocated elements *len*.



**NOTE.** The only function to free the memory allocated by any of these functions is `ippsFree()`.

---

## ippMalloc

*Allocates memory aligned to 32-byte boundary.*

---

### Syntax

```
Ipp8u* ippMalloc_8u(int len);  
Ipp16u* ippMalloc_16u(int len);  
Ipp32u* ippMalloc_32u(int len);  
Ipp8s* ippMalloc_8s(int len);  
Ipp16s* ippMalloc_16s(int len);  
Ipp32s* ippMalloc_32s(int len);  
Ipp64s* ippMalloc_64s(int len);  
Ipp32f* ippMalloc_32f(int len);  
Ipp64f* ippMalloc_64f(int len);  
Ipp8sc* ippMalloc_8sc(int len);  
Ipp16sc* ippMalloc_16sc(int len);  
Ipp32sc* ippMalloc_32sc(int len);  
Ipp64sc* ippMalloc_64sc(int len);  
Ipp32fc* ippMalloc_32fc(int len);  
Ipp64fc* ippMalloc_64fc(int len);
```

### Parameters

*len*                                      Number of elements to allocate.

### Description

The function `ippMalloc` is declared in the `ipp.h` file. This function allocates memory block aligned to a 32-byte boundary for elements of different data types.

Example 3-2 shows how to use the function `ippMalloc_8u`.

## Example 3-2 Using the Function `ippsMalloc`

```
void func_malloc(void)
{
    Ipp8u* pBuf = ippsMalloc_8u(8*sizeof(Ipp8u));
    if(NULL == pBuf)
        // not enough memory
        ippsFree(pBuf);
}
```

### Return Values

The return value of `ippsMalloc` is a pointer to an aligned memory block. If no memory is available in the system, then the `NULL` value is returned. To free this block, use the function `ippsFree`.

## `ippsFree`

*Frees memory allocated by the function `ippsMalloc`.*

---

### Syntax

```
void ippsFree(void* ptr);
```

### Parameters

<i>ptr</i>	Pointer to a memory block to be freed. The memory block pointed to with <i>ptr</i> has been allocated by the function <code>ippsMalloc</code> .
------------	---

### Description

The function `ippsFree` is declared in the `ipps.h` file. This function frees the aligned memory block allocated by the function `ippsMalloc`.



---

**CAUTION.** You can not use the function `ippsFree` to free memory allocated by standard functions like `malloc` or `calloc`, nor can the memory allocated by the function `ippsMalloc` be freed by `free`.

---

## Common Functions

This section describes the Intel IPP functions that perform special operations common for all domains. For example, the function `ippGetStatusString` gets a brief description of the status code returned by the Intel IPP software. The function `ippGetCpuType` retrieves the type of the processor in the system running the software. The function `ippGetCpuClocks` gets the current number of processor clocks, which is widely used for operations timing. The functions `ippSetFlushToZero` and `ippSetDenormAreZeros` enable or disable special processor modes. The function `ippAlignPtr` performs pointer alignment. The functions `ippMalloc` and `ippFree` allow to allocate and free the memory block aligned to 32-byte boundary. The specific subset contains functions to control the dispatching of the merged static libraries. All these functions are grouped in the separate sub-library called `ippcore`.

### ippGetStatusString

*Translates a status code into a message.*

---

#### Syntax

```
const char* ippGetStatusString(IppStatus stsCode);
```

#### Parameters

<i>stsCode</i>	Code that indicates the status type (see Table 2-3 "Error Status Values and Messages").
----------------	---

#### Description

The function `ippGetStatusString` is declared in the `ippcore.h` file. This function returns a pointer to the text string associated with a status code of type `IppStatus`. Use this function to produce error and warning messages for users. The returned pointer is a pointer to an internal static buffer and need not be released.

A code example 3-3 shows how to use the function `ippGetStatusString`. If you call an Intel IPP function, in this case, `ippsAddC_16s_I` with a `NULL` pointer, it returns an error code `-8`. The status information function translates this code into the corresponding message "Null Pointer Error".

### Example 3-3 Using the Function `ippGetStatusString`

```
void statusinfo(void) {
    IppStatus st = ippAddC_16s_I (3, 0, 0);

    printf("%d : %s\n", st, ippGetStatusString(st));
}
```

Output:

```
-8, Null Pointer Error
```

## ippGetCpuType

Returns a processor type.

### Syntax

```
IppCpuType ippGetCpuType (void);
```

### Description

The function `ippGetCpuType` is declared in the `ippcore.h` file. This function detects the processor type used in your computer system and returns an appropriate `IppCpuType` variable value. Table 3-2 below lists possible values and their meaning.



**CAUTION.** This function returns only type of the processor. If you need more information about processor features please use the function [ippGetCpuFeatures](#) instead.

**Table 7: Table 3-2. Processor Type Detected in Intel IPP**

Returned Variable Value	Processor Type
<code>ippCpuPP</code>	Intel® Pentium® processor
<code>ippCpuPMX</code>	Intel® Pentium® processor with MMX™ technology
<code>ippCpuPPR</code>	Intel® Pentium® Pro processor
<code>ippCpuPII</code>	Intel® Pentium® II processor



Returned Variable Value	Processor Type
<code>ippCpuPIII</code>	Intel® Pentium® III processor or Intel® Pentium® III Xeon® processor
<code>ippCpuP4</code>	Intel® Pentium® 4 processor or Intel® Xeon® processor
<code>ippCpuP4HT</code>	Intel® Pentium® 4 processor with Hyper-Threading Technology
<code>ippCpuP4HT2</code>	Intel® Pentium® 4 processor with Intel® Streaming SIMD Extensions 3
<code>ippCpuCentrino</code>	Intel® Centrino™ mobile technology
<code>ippCpuCoreSolo</code>	Intel® Core™ Solo processor
<code>ippCpuCoreDuo</code>	Intel® Core™ Duo processor
<code>ippCpuITP</code>	Intel® Itanium® processor
<code>ippCpuITP2</code>	Intel® Itanium® 2 processor
<code>ippCpuEM64T</code>	Intel® 64 instruction set architecture
<code>ippCpuC2D</code>	Intel® Core™2 Duo processor
<code>ippCpuC2Q</code>	Intel® Core™2 Quad processor
<code>ippCpuPenryn</code>	Intel® Core™2 processor with Intel® Streaming SIMD Extensions 4.1
<code>ippCpuBonnell</code>	Intel® Atom™ processor
<code>ippCpuNehalem</code>	Intel® Core™ i7 processor
<code>ippCpuSSE</code>	Processor with Intel® Streaming SIMD Extensions instruction set
<code>ippCpuSSE2</code>	Processor with Intel® Streaming SIMD Extensions 2 instruction set

Returned Variable Value	Processor Type
ippCpuSSE3	Processor with Intel® Streaming SIMD Extensions 3 instruction set
ippCpuSSSE3	Processor with Supplemental Intel® Streaming SIMD Extensions 3 instruction set
ippCpuSSE41	Processor with Intel® Streaming SIMD Extensions 4.1instruction set
ippCpuSSE42	Processor with Intel® Streaming SIMD Extensions 4.2 instruction set
ippCpuAVX	Processor supports Intel® Advanced Vector Extensions instruction set
ippCpuX8664	Processor supports 64 bit extension
ippCpuUnknown	Unknown processor

## ippGetCpuClocks

Returns a current value of the time stamp counter (TSC) register.

### Syntax

```
Ipp64u ippGetCpuClocks (void);
```

### Description

The function `ippGetCpuClocks` is declared in the `ippcore.h` file. This function reads the current state of the time stamp counter (TSC) register and returns its value. A hardware exception is possible if TSC reading is not supported by the current chipset.

## ippGetCpuFreqMhz

*Estimates the processor operating frequency.*

---

### Syntax

```
IppStatus ippGetCpuFreqMhz(int* pMhz);
```

### Parameters

*pMhz* pMhz Pointer to the result.

### Description

The function `ippGetCpuFreqMhz` is declared in the `ippcore.h` file. This function estimates the processor operating frequency and returns its value in MHz as an integer stored in *pMhz*. Note that no exact value of frequency is guaranteed. So the estimated value can vary depending on the processor workload.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition when the <i>pMhz</i> pointer is NULL.

## ippGetCpuFeatures

*Retrieves the processor features.*

---

### Syntax

```
IppStatus ippGetCpuFeatures(Ipp64u* pFeaturesMask, Ipp32u pCpuidInfoRegs[4]);
```

### Parameters

<i>pFeaturesMask</i>	Pointer to the features mask. It can has the value <code>ippCPUID_GETINFO_A</code> .
<i>pCpuidInfoRegs</i>	Pointer to the 4-element vector for data from the registers <code>eax</code> , <code>ebx</code> , <code>ecx</code> , <code>edx</code> of the function <code>CPUID.1</code> .

## Description

The function `ippGetCpuFeatures` is declared in the `ippcore.h` file. This function retrieves some of the CPU features returned by the function `CPUID.1` and stores them consecutively in the mask `pFeaturesMask`. Table 3-3 lists the features stored in the mask.

If `pFeaturesMask` hasn't any value on input, then the function retrieves the features in accordance with `eax=1` and `ecx=0`. If `pFeaturesMask` is set to `ippCPUID_GETINFO_A`, then the function retrieves the features in accordance with the input values of the registers `eax` and `ecx` that are specified in this case by the `pCpuIdInfoRegs[0]` and `pCpuIdInfoRegs[2]` respectively.

**Table 8: Table 3-3. CPU Features Mask**

Mask Value	Bit Name	Feature	Mask Bit Number
1	<code>ippCPUID_MMX</code>	MMX™ technology	0
2	<code>ippCPUID_SSE</code>	Intel® Streaming SIMD Extensions	1
4	<code>ippCPUID_SSE2</code>	Intel® Streaming SIMD Extensions 2	2
8	<code>ippCPUID_SSE3</code>	Intel® Streaming SIMD Extensions 3	3
16	<code>ippCPUID_SSSE3</code>	Supplemental Intel® Streaming SIMD Extensions	4
32	<code>ippCPUID_MOVBE</code>	MOVBE instruction is supported	5
64	<code>ippCPUID_SSE41</code>	Intel® Streaming SIMD Extensions 4.1	6
128	<code>ippCPUID_SSE42</code>	Intel® Streaming SIMD Extensions 4.2	7

Mask Value	Bit Name	Feature	Mask Bit Number
256	ippCpuID_AVX	The processor supports Intel® Advanced Vector Extensions (Intel® AVX) instruction set	8
512	ippAVX_ENABLEDBYOS	The operating system supports Intel® AVX	9
1024	ippCpuID_AES	AES instruction is supported	10
2048	ippCpuID_CLMUL	PCLMULQDQ instruction is supported	11

Additionally all features returned by the function `CPUID.1` can be stored in the 4-element vector `pCpuIDInfoRegs` where each next element contains data from one of the registers `eax`, `ebx`, `ecx`, `edx` respectively. If these data are not required, the pointer `pCpuIDInfoRegs` must be set to `NULL`.



**CAUTION.** Intel® Itanium® processors are not supported.

## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error condition when the `pFeaturesMask` pointer is `NULL`.

`ippStsNotSupportedCpu` Indicates that processor is not supported.

## ippGetNumCoresOnDie

Returns the number of cores for the multi-core processors.

---

### Syntax

```
int ippGetNumCoresOnDie (void);
```

### Description

The function `ippGetNumCoresOnDie` is declared in the `ippcore.h` file. This function allows to distinguish the multi-core processors returning the number of cores.

## ippGetMaxCacheSizeB

Returns maximum size of the L2 and L3 caches of the processor.

---

### Syntax

```
IppStatus ippGetMaxCacheSizeB(int* pSizeByte);
```

### Parameters

*pSizeByte*                      Pointer to the output result.

### Description

The function `ippGetMaxCacheSizeB` is declared in the `ippcore.h` file. This function finds the maximum size (in bytes) of the L2 and L3 caches of the processor used in your computer system. The result is stored it in the *pSizeByte*.



**CAUTION.** Intel® Itanium® processors are not supported.

---

If the processor is not supported, or size of cache is unknown, the result is 0, and the function returns corresponding warning message.

### Return Values

`ippStsNoErr`                      Indicates no error.

`ippStsNullPtrErr` Indicates an error condition when the `pSizeByte` pointer is `NULL`.  
`ippStsNotSupportedCpu` Indicates that processor is not supported.  
`ippStsUnknownCacheSize` Indicates that size of the cache is unknown.

## ippSetFlushToZero

*Enables or disables flush-to-zero mode.*

---

### Syntax

```
IppStatus ippSetFlushToZero(int value, unsigned int* pUMask);
```

### Parameters

<code>value</code>	Switch to set or clear the corresponding bit of the MXCSR register.
<code>pUMask</code>	Pointer to the current underflow exception mask; may be set to <code>NULL</code> .

### Description

The function `ippSetFlushToZero` is declared in the `ippcore.h` file. This function enables (when the `value` is not equal to 0) or disables (when the `value` is equal to 0) a flush-to-zero (FTZ) mode of processors that support Streaming SIMD Extensions (SSE) instructions. The FTZ mode controls the masked response to a SIMD floating-point underflow condition. The FTZ mode is provided primarily for performance reasons. At the cost of a slight precision loss, FTZ mode enables faster execution of applications where underflows are common and rounding the underflow result to zero can be tolerated.

FTZ mode is possible only when the mask register is in a certain state. The `ippSetFlushToZero` function checks and changes this state if necessary. After disabling the FTZ mode, you can restore the initial mask register state. To do this, you must declare a variable of `unsigned integer` type in your application and point to it the parameter `pUMask` of the `ippSetFlushToZero` function. The initial state of mask register is saved in this location and can be restored later. If you do not need to restore the initial mask state, then the pointer `pUMask` may be set to `NULL`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsCpuNotSupportedErr` Indicates an error condition when the FTZ mode is not supported by the processor.

## ippSetDenormAreZeros

*Enables or disables denormals-are-zero mode.*

---

### Syntax

```
IppStatus ippSetDenormAreZeros(int value);
```

### Parameters

`value` Switch to set or clear the corresponding bit of the MXCSR register.

### Description

The function `ippSetDenormAreZeros` is declared in the `ippcore.h` file. This function enables (when the `value` is not equal to 0) or disables (when the `value` is equal to 0) the denormals-are-zero (DAZ) mode of processors that support Streaming SIMD Extensions (SSE) instructions. The DAZ mode controls the processor response to a SIMD floating-point denormal operand condition. When the DAZ flag is set, the processor converts all denormal source operands to zero with the sign of the original operand before performing any computations on source data. The DAZ mode is provided to improve processor performance for applications such as streaming media processing, where rounding a denormal operand to zero does not noticeably affect the quality of the processed data.

### Return Values

`ippStsNoErr` Indicates no error.

`ippStsCpuNotSupportedErr` Indicates an error condition when the DAZ mode is not supported by the processor.

## ippAlignPtr

*Aligns a pointer to the specified number of bytes.*

---

### Syntax

```
void* ippAlignPtr(void* ptr, int alignBytes);
```



## Parameters

<i>ptr</i>	Aligned pointer.
<i>alignBytes</i>	Number of bytes to align. Possible values are the powers of 2, that is, 2, 4, 8, 16 and so on.

## Description

The function `ippAlignPtr` is declared in the `ippcore.h` file. This function returns a pointer *ptr* aligned to the specified number of bytes *alignBytes*. Possible values of *alignBytes* are powers of two. The function does not check the validity of this parameter.



**CAUTION.** Do not free the pointer returned by the function, but free the original pointer.

## ippSetNumThreads

*Sets the number of threads in the multithreading environment.*

### Syntax

```
IppStatus ippSetNumThreads(int numThr);
```

### Parameters

<i>numThr</i>	Number of threads, should be greater than 0.
---------------	--

### Description

The function `ippSetNumThreads` is declared in the `ippcore.h` file. This function sets the number of OMP threads. A number of established threads may be less than specified *numThr*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsSizeErr</code>	Indicates an error condition if <i>numThr</i> is less than or equal to 0.
<code>ippStsNoOperation</code>	Indicates that there is no such operation in the static version of the library.

## ippGetNumThreads

Returns the number of existing threads in the multithreading environment.

---

### Syntax

```
ippStatus ippGetNumThreads(int* pNumThr);
```

### Parameters

*pNumThr*                      Pointer to the number of threads.

### Description

The function `ippGetNumThreads` is declared in the `ippcore.h` file. This function returns the number of OMP threads specified by the user previously. If it has not been specified, the function returns the initial number of threads that depends on the number of logical processors.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition when the <i>pMhz</i> pointer is <code>NULL</code> .
<code>ippStsNoOperation</code>	Indicates that there is no such operation in the static version of the library.

## ippMalloc

Allocates memory aligned to 32-byte boundary.

---

### Syntax

```
void* ippMalloc(int length);
```

### Parameters

*len*                          Size (in bytes) of the allocated block.

### Description

The function `ippMalloc` is declared in the `ippcore.h` file. This function allocates memory block aligned to a 32-byte boundary.

## Return Values

The return value of `ippMalloc` is a pointer to an aligned memory block. To free this block, use only the function `ippFree`.

## ippFree

*Frees memory allocated by the function `ippMalloc`.*

---

### Syntax

```
void ippFree(void* ptr);
```

### Parameters

`ptr`                                      Pointer to a memory block to be freed.

### Description

The function `ippFree` is declared in the `ippcore.h` file. This function frees the aligned memory block allocated by the function `ippMalloc`.



---

**CAUTION.** You can not use the function `ippFree` to free memory allocated by any other functions like `malloc`, nor can the memory allocated by the function `ippMalloc` be freed by the function `free`.

---

## Dispatcher Control Functions

This section describes Intel IPP functions that control the dispatchers of the merged static libraries.

## ippStaticInit

*Automatically initializes static code that is the most appropriate for the current processor type.*

---

### Syntax

```
IppStatus ippStaticInit(void);
```

## Description

The function `ippStaticInit` is declared in the `ippcore.h` file. This function detects the processor type used in the user computer system and sets the most appropriate processor-specific static code of the Intel IPP software.



---

**NOTE.** This function operates only in the static version of the library.

---



---

**CAUTION.** You can not use any other Intel IPP function while the function `ippStaticInit` continues execution.

---

## Return Values

<code>ippStsNoErr</code>	Indicates that the most appropriate static code of the Intel IPP software is successfully set.
<code>ippStsNonIntelCpu</code>	Indicates that the static version of generic code for Intel Architecture is set.
<code>ippStsNoOperationInDll</code>	Indicates that there is no such operation in the dynamic version of the library.

## ippInit

*Automatically initializes the library code that is most appropriate for the current processor type.*

---

### Syntax

```
IppStatus ippInit(void);
```

### Description

The function `ippInit` is declared in the `ippcore.h` file. This function detects the processor type used in the user computer system and sets the processor-specific code of the Intel IPP library most appropriate for the current processor type.



---

**CAUTION.** You can not use any other Intel IPP function while the function `ippInit` continues execution.

---

## Return Values

<code>ippStsNoErr</code>	Indicates that the required processor-specific code is successfully set.
--------------------------	--

## ippStaticInitCpu

**THIS FUNCTION IS DEPRECATED.** *Initializes the specified version of the static code.*

---

### Syntax

```
IppStatus ippStaticInitCpu(IppCpuType cpu);
```

### Parameters

`cpu` Processor type.

### Description



**CAUTION. THIS FUNCTION IS DEPRECATED.** Please use the function `ippInitCpu` instead.

---

The function `ippStaticInitCpu` is declared in the `ippcore.h` file. This function sets the processor-specific version of the static code of the Intel IPP library in accordance with the specified processor type `cpu`.



**CAUTION.** This function operates only in the static version of the library.

---

## Return Values

<code>ippStsNoErr</code>	Indicates that the required processor-specific static code is successfully set.
<code>ippStsCpuMismatch</code>	Indicates that the specified processor type is not valid and the previously set code version is used.
<code>ippStsNoOperationInDll</code>	Indicates that there is no such operation in the dynamic version of the library.

## ippInitCpu

*Initializes the version of the library code for the specified processor type.*

---

### Syntax

```
IppStatus ippInitCpu(IppCpuType cpu);
```

### Parameters

*cpu* Processor type.

### Description

The function `ippInitCpu` is declared in the `ippcore.h` file. This function sets the processor-specific code of the Intel IPP library in accordance with the specified processor type *cpu*.



---

**CAUTION.** You can not use any other Intel IPP function while the function `ippInitCpu` continues execution.

---

### Return Values

<code>ippStsNoErr</code>	Indicates that the required processor-specific code is successfully set.
<code>ippStsCpuMismatch</code>	Indicates that the specified processor type is not valid; the previously set code is used.

## ippEnableCpu

*Enables automatic dispatching for the specified processor type.*

---

### Syntax

```
IppStatus ippEnableCpu(IppCpuType cpu);
```

### Parameters

*cpu* Processor type.

### Description

The function `ippEnableCpu` is declared in the `ippcore.h` file.

This function enables automatic dispatching of the library code for processors with Intel® Advanced Vector Extensions (AVX) on Intel® AVX enabled hardware or simulators. To do this the function `ippEnableCpu` must be called with the parameter `cpu=ippCpuAVX` before the function `ippInit`.

### Return Values

<code>ippStsNoErr</code>	Indicates that the required processor-specific code is successfully set.
--------------------------	--

## Internationalization Functions

This section describes auxiliary functions for adapting Intel IPP software to different languages and regional differences.

### ippMessageCatalogOpenI18n

*Opens an i18n message catalog.*

#### Syntax

```
IppStatus ippMessageCatalogOpenI18n(IppMsgCatalog** ppMsgCatalog);
```

#### Parameters

<code>ppMsgCatalog</code>	Double pointer to the message catalog
---------------------------	---------------------------------------

#### Description

The function `ippMessageCatalogOpenI18n` is declared in the `ippcore.h` file. This function opens the i18n [message catalog](#) `ppMsgCatalog` containing the translated strings.



**CAUTION.** This function allocates memory that can be freed only by the function [ippMessageCatalogCloseI18n](#). Always use the function `ippMessageCatalogCloseI18n` after opening the message catalog regardless of the returned status value.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition when the <code>pMsgCatalog</code> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error condition when no memory allocated.
<code>ippStsI18nUnsupportedErr</code>	Indicates an error condition when the internationalization i18n is not supported.
<code>ippStsI18nMsgCatalogOpenErr</code>	Indicates an error condition when the message catalog cannot be opened; to get extended information use <code>errno</code> on Linux* OS and <code>GetLastError</code> on Windows* OS.

## ippMessageCatalogCloseI18n

*Closes the opened i18n message catalog.*

---

### Syntax

```
IppStatus ippMessageCatalogCloseI18n(IppMsgCatalog* pMsgCatalog);
```

### Parameters

`pMsgCatalog`                      Pointer to the message catalog.

### Description

The function `ippMessageCatalogCloseI18n` is declared in the `ippcore.h` file. This function closes the message catalog `pMsgCatalog`, which is opened by the function [ippMessageCatalogOpenI18n](#). Always use the function `ippMessageCatalogCloseI18n` after opening the message catalog to free memory allocated by the function `ippMessageCatalogOpenI18n`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition when the <code>pMsgCatalog</code> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition when the catalog identifier is not correct.



`ippStsI18nMsgCatalogCloseErr` Indicates an error condition when the message catalog cannot be closed; to get extended information use `errno` on Linux\* OS and `GetLastError` on Windows\* OS.

`ippStsI18nUnsupportedErr` Indicates an error condition when the internationalization i18n is not supported.

## ippGetMessageStatusI18n

*Returns the translation of the status message.*

### Syntax

```
IppStatus ippGetMessageStatusI18n(const IppMsgCatalog* pMsgCatalog, IppStatus
stsCode, IppMsg* pMsg);
```

### Parameters

<code>pMsgCatalog</code>	Pointer to the message catalog.
<code>stsCode</code>	Code of an Intel IPP status.
<code>pMsg</code>	Pointer to the string containing the translated status message.

### Description

The function `ippGetMessageStatusI18n` is declared in the `ippcore.h` file. This function returns the string `pMsg` containing the translation of the status message corresponding to `stsCode`. The translated message is encoded in UTF16 on Windows\* OS, and UTF8 on Linux\* OS. The messages are stored in the *message catalog* `pMsgCatalog`. On Linux\* OS the binary message catalog is generated using the `gencat` utility (for more details see *X/Open Message Catalog Handling* [OMC]). On Windows\* OS the binary message catalog is generated using the message compiler (`mc`) and resource compiler (`rc`) utilities of the Microsoft SDK Tools.

The message catalog `pMsgCatalog` must be opened by the function `ippMessageCatalogOpenI18n` beforehand. The returned `pMsg` is freed by the function `ippMessageCatalogCloseI18n`.

If the translation is not available, the function returns the not translated status message.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error condition when one of the specified pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error condition when no memory allocated.
<code>ippStsContextMatchErr</code>	Indicates an error condition when the catalog identifier is not correct.
<code>ippStsUnknownStatusCodeErr</code>	Indicates an error condition when <code>stsCode</code> has unknown value.
<code>ippStsI18nMsgCatalogInvalid</code>	Indicates a warning when the message catalog is not valid; the status message in English is returned.
<code>ippStsI18nGetMessageFail</code>	Indicates a warning when the function cannot return the translated message; the status message in English is returned. To get extended information use <code>errno</code> on Linux* OS and <code>GetLastError</code> on Windows* OS.

## ippStatusToMessageIdI18n

*Transforms an Intel IPP status code to the message ID for the message catalog.*

---

### Syntax

```
Ipp32u ippStatusToMessageIdI18n(IppStatus stsCode);
```

### Parameters

*stsCode*                      Code of an Intel IPP status.

### Description

The function `ippStatusToMessageIdI18n` is declared in the `ippcore.h` file. This function returns the message ID in the i18n [message catalog](#) for the Intel IPP status specified by the code `stsCode`. This function is useful for the direct access to this message catalog.

# Vector Initialization Functions

This chapter describes the Intel® IPP functions that initialize vectors with either constants, the contents of other vectors, or the generated signals.

The full list of the functions in this group is given in Table 4-1 below.

**Table 4-1 Intel IPP Vector Initialization Functions**

Function Base Name	Operation
Vector Initialization Functions	
<a href="#">Copy</a>	Copies the contents of one vector into another.
<a href="#">PackBits</a>	Packs part of data bits from the vector to the another vector.
<a href="#">Move</a>	Moves the contents of one vector to another vector.
<a href="#">Set</a>	Initializes vector elements to a specified common value.
<a href="#">Zero</a>	Initializes a vector to zero.
Tone and Triangle Generation Functions	
<a href="#">ToneInitAllocQ15</a>	Allocates memory and initializes the tone generator state
<a href="#">ToneFree</a>	Frees memory allocated by the function <code>ipp-sToneInitAllocQ15</code> .
<a href="#">ToneGetStateSizeQ15</a>	Computes the length of the tone generator structure.
<a href="#">ToneInitQ15</a>	Initializes the tone generator specification structure.
<a href="#">ToneQ15</a>	Generates a tone in accordance with tone generator specification
<a href="#">Tone_Direct</a>	Generates a tone with a given frequency, phase, and magnitude.
<a href="#">ToneQ15_Direct</a>	Generates a tone with a given frequency, phase, and magnitude.

Function Base Name	Operation
TriangleInitAllocQ15	Allocates memory and initializes the triangle generator state.
TriangleFree	Frees memory allocated by the function ippsTriangleInitAlloc.
TriangleGetStateSizeQ15	Computes the length of the triangle generator structure.
TriangleInitQ15	Initializes the triangle generator specification structure.
TriangleQ15	Generates a triangle in accordance with triangle generator specification
Triangle_Direct	Generates a triangle with a given frequency, phase, and magnitude.
TriangleQ15_Direct	Generates a triangle with a given frequency, phase, and magnitude
Uniform Distribution Functions	
RandUniformInitAlloc	Allocates memory and initializes a noise generator with uniform distribution.
RandUniformFree	Closes the uniform distribution generator state.
RandUniformGetSize	Computes the length of the uniform distribution generator structure
RandUniformInit	Initializes a noise generator with uniform distribution.
RandUniform	Generates the pseudo-random samples with a uniform distribution.
RandUniform_Direct	Generates the pseudo-random samples with a uniform distribution in direct mode.
Gaussian Distribution Functions	

Function Base Name	Operation
<a href="#">RandGaussInitAlloc</a>	Allocates memory and initializes a noise generator with Gaussian distribution.
<a href="#">RandGaussFree</a>	Closes the Gaussian distribution generator state.
<a href="#">RandGaussGetSize</a>	Computes the length of the Gaussian distribution generator state
<a href="#">RandGaussInit</a>	Initializes a noise generator with Gaussian distribution.
<a href="#">RandGauss</a>	Generates the pseudo-random samples with a Gaussian distribution.
<a href="#">RandGauss_Direct</a>	Generates pseudo-random samples with a Gaussian distribution in the direct mode.
Special Vector Functions	
<a href="#">VectorJaehne</a>	Creates a Jaehne vector.
<a href="#">VectorSlope</a>	Creates a slope vector.
<a href="#">VectorRamp</a>	Creates a ramp vector.

## Vector Initialization Functions

This section describes functions that initialize the values of vector elements. All vector elements can be initialized to a common zero or another specified value. They can also be initialized to respective values of a second vector elements.

### Copy

*Copies the contents of one vector into another.*

#### Syntax

```
ippStatus ippCopy_1u(const Ipp8u* pSrc, int srcBitOffset, Ipp8u* pDst, int
dstBitOffset, int len);
```

```

IppStatus ippsCopy_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsCopy_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsCopy_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsCopy_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCopy_64s(const Ipp64s* pSrc, Ipp64s* pDst, int len);
IppStatus ippsCopy_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsCopy_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsCopy_32sc(const Ipp32sc* pSrc, Ipp32sc* pDst, int len);
IppStatus ippsCopy_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsCopy_64sc(const Ipp64sc* pSrc, Ipp64sc* pDst, int len);
IppStatus ippsCopy_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements to copy.
<i>srcBitOffset</i>	Offset (in bits) in the first byte of the source vector.
<i>dstBitOffset</i>	Offset (in bits) in the first byte of the destination vector.

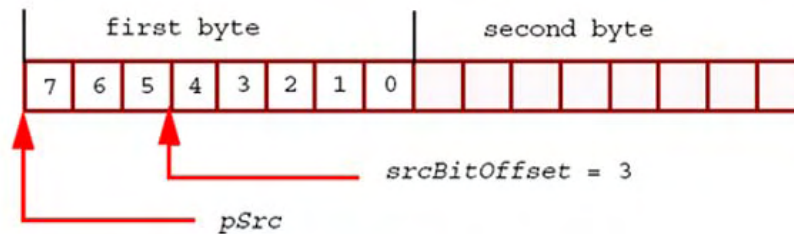
## Description

The function `ippsCopy` is declared in the `ipps.h` file. This function copies the first *len* elements from a source vector *pSrc* into a destination vector *pDst*.

**ippsCopy\_1u.** This function flavor performs copying operation on a vector that has a `8u` data type where each byte represents eight consecutive elements. In this case additional parameters *srcBitOffset* and *dstBitOffset* are required to specify start positions in the source and

destination vectors respectively. Note that the bit order in each byte is inverse relative to the element order, that is, the first element in a vector is represented by the last (7th) bit in the first byte of the vector (see Figure 4-1 below).

**Figure 4-1 Bit Layout for the Function `ippsCopy_1u`.**



**CAUTION.** These functions perform only copying operations described above and are not intended to move data. Their behavior is unpredictable if source and destination buffers are overlapping. To move data, the functions [ippsMove](#) must be used.

Example 4-1 shows how to use the function `ippsCopy`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 4-1 Using the `ippsCopy` Function

```
IppStatus copy(void) {
    char src[] = "to be copied\0";
    char dst[256];

    return ippsCopy_8u(src, dst,
        strlen(src)+1);
}
```

## PackBits

*Packs part of data bits from the vector to the bitstream.*

---

### Syntax

```
IppStatus ippsPackBits_32u8u(const Ipp32u* pSrcBit, const int* pSrcBits, int
srcLen, Ipp8u* pDst, int dstBitOffset, int* pDstLenBit);
```

### Parameters

<i>pSrcBit</i>	Pointer to the source vector with data.
<i>pSrcBits</i>	Pointer to the source vector that specifies the number of data bits.
<i>srcLen</i>	Number of elements in each source vector.
<i>pDst</i>	Pointer to the destination vector (bitstream).
<i>dstBitOffset</i>	Offset (in bits) in the first byte of the destination vector.
<i>pDstLenBit</i>	Pointer to the length in bits of the destination vector.

### Description

The function `ippsPackBits` is declared in the `ipps.h` file. This function copies the certain number of bits from each element of the data source vector *pSrcBit* and stores them contiguously in the destination vector *pDst* starting from the bit position *dstBitOffset*. Each element of the source vector *pSrcBits* contains the number of bits that are copied from the corresponding element of the vector *pSrcBit*. If this number is less than 0, or greater than



32, then its value is set to 0 or 32 respectively. If such cases occur, the function returns the warning message after completing the operation. After completing the operation the function returns the length of the destination vector in bits *pDstLenBit*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>srcLen</i> is less than or equal to 0, or <i>dstBitOffset</i> is less than 0.
<code>ippStsOverlongString</code>	Indicates a warning when <i>pSrcBits[i]</i> is greater than 32 or less than 0.

## Move

Moves the contents of one vector to another vector.

### Syntax

```
IppStatus ippMove_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippMove_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippMove_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippMove_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippMove_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippMove_64s(const Ipp64s* pSrc, Ipp64s* pDst, int len);
IppStatus ippMove_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippMove_32sc(const Ipp32sc* pSrc, Ipp32sc* pDst, int len);
IppStatus ippMove_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippMove_64sc(const Ipp64sc* pSrc, Ipp64sc* pDst, int len);
IppStatus ippMove_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector used to initialize <i>pDst</i> .
<i>pDst</i>	Pointer to the destination vector to be initialized.

*len*                                      Number of elements to move.

## Description

The function `ippMove` is declared in the `ipp.h` file. This function moves the first *len* elements from a source vector *pSrc* into the destination vector *pDst*. If some parts of the source and destination vectors are overlapping, then the function ensures that the original source bytes in the overlapping parts are moved (it means that they are copied before being overwritten) to the appropriate parts of the destination vector.

Code example 4-2 below shows how to use the function `ippMove`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

**Example 4-2 Using the `ippsMove` function**

```

Ipp8u pSrc[10] = { "123456789" };
Ipp8u pDst[6];
int len = 6;
IppStatus status;
status = ippsMove_8u ( pSrc, pDst, len );
if(ippStsNoErr != status)
    printf("IPP Error: %s",ippGetStatusString(status));
result:
pSrc = 123456789
pDst = 123456

```

**Set**

*Initializes vector elements to a specified common value.*

---

**Syntax**

```

IppStatus ippsSet_8u(Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsSet_16s(Ipp16s val, Ipp16s* pDst, int len);
IppStatus ippsSet_16sc(Ipp16sc val, Ipp16sc* pDst, int len);
IppStatus ippsSet_32s(Ipp32s val, Ipp32s* pDst, int len);
IppStatus ippsSet_32f(Ipp32f val, Ipp32f* pDst, int len);
IppStatus ippsSet_32sc(Ipp32sc val, Ipp32sc* pDst, int len);
IppStatus ippsSet_32fc(Ipp32fc val, Ipp32fc* pDst, int len);
IppStatus ippsSet_64s(Ipp64s val, Ipp64s* pDst, int len);
IppStatus ippsSet_64f(Ipp64f val, Ipp64f* pDst, int len);
IppStatus ippsSet_64sc(Ipp64sc val, Ipp64sc* pDst, int len);
IppStatus ippsSet_64fc(Ipp64fc val, Ipp64fc* pDst, int len);

```

## Parameters

<i>pDst</i>	Pointer to the vector to be initialized.
<i>len</i>	Number of elements to initialize.
<i>val</i>	Value used to initialize the vector <i>pDst</i> .

## Description

The function `ippsSet` is declared in the `ipps.h` file. This function initializes the first *len* elements of the real or complex vector *pDst* to contain the same value *val*.

Code example 4-3 below shows how to use the function `ippsSet`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 4-3 Using the `ippsSet` Function

```

IppStatus set(void) {
    char src[] = "set";
    return ippsSet_8u('0', src, strlen(src));
}

```

## Zero

*Initializes a vector to zero.*

---

### Syntax

```

IppStatus ippsZero_8u(Ipp8u* pDst, int len);
IppStatus ippsZero_16s(Ipp16s* pDst, int len);
IppStatus ippsZero_32s(Ipp32s* pDst, int len);
IppStatus ippsZero_32f(Ipp32f* pDst, int len);
IppStatus ippsZero_64s(Ipp64s* pDst, int len);
IppStatus ippsZero_64f(Ipp64f* pDst, int len);

```

```

IppStatus ippsZero_16sc(Ipp16sc* pDst, int len);
IppStatus ippsZero_32sc(Ipp32sc* pDst, int len);
IppStatus ippsZero_32fc(Ipp32fc* pDst, int len);
IppStatus ippsZero_64sc(Ipp64sc* pDst, int len);
IppStatus ippsZero_64fc(Ipp64fc* pDst, int len);

```

### Parameters

<i>pDst</i>	Pointer to the vector to be initialized to zero.
<i>len</i>	Number of elements to initialize.

### Description

The function `ippsZero` is declared in the `ipps.h` file. This function initializes the first *len* elements of the vector *pDst* to 0. If *pDst* is a complex vector, both real and imaginary parts are zeroed.

Code example 4-4 shows how to use the function `ippsZero`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is <code>NULL</code>
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0

### Example 4-4 Using the `ippsZero` Function

```

IppStatus zero(void) {
    char src[] = "zero";
    return ippsZero_8u(src, strlen(src));
}

```

## Sample-Generating Functions

This section describes Intel IPP functions which generate tone samples, triangle samples, pseudo-random samples with uniform distribution, and pseudo-random samples with Gaussian distribution, as well as special test samples.

Some sample-generating functions operate with data in the fixed point format. These functions have Q15 suffix in their name. This means that integer data are used in calculations inside the function as real numbers equal to the integer value multiplied by  $2^{-15}$  (where “15” is called a *scale factor*).

## Tone-Generating Functions

The functions described below generate a tone (or “sinusoid”) of a given frequency, phase, and magnitude. Tones are fundamental building blocks for analog signals. Thus, sampled tones are extremely useful in signal processing systems as test signals and as building blocks for more complex signals.

The use of tone functions is preferable against the analogous C math library’s `sin()` function for many applications, because Intel IPP functions can use information retained from the computation of the previous sample to compute the next sample much faster than standard `sin()` or `cos()`.

## ToneInitAllocQ15

*Allocates memory and initializes the tone generator specification structure for fixed point data.*

---

### Syntax

```
IppStatus ippsToneInitAllocQ15_16s(IppToneState_16s** ppToneState, Ipp16s
magn, Ipp16s rFreqQ15, Ipp32s phaseQ15);
```

### Parameters

<i>ppToneState</i>	Double pointer to the tone generator specification structure.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].

## Description

The function `ippsToneInitAllocQ15` is declared in the `ipps.h` file. This function allocates memory and initializes the tone generator structure `pToneState` with the specified frequency `rFreqQ15`, phase `phaseQ15`, and magnitude `magn`. Input data in the fixed point format Q15 are converted to the corresponding float data type that lay in the range  $[0, 0.5)$  for relative frequency and  $[0, 2\pi)$  for phase. Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format. Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

Code [example 4-5](#) demonstrates how to use the function `ippsToneInitAllocQ15`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pToneState</code> pointer is NULL.
<code>ippStsToneMagnErr</code>	Indicates an error when <code>magn</code> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <code>rFreqQ15</code> is negative, or greater than 16383.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <code>phaseQ15</code> value is negative, or greater than 205886.

## ToneFree

*Frees memory allocated by the function `ippsToneInitAllocQ15`.*

---

### Syntax

```
IppStatus ippsToneFree(IppToneState_16s* pToneState);
```

### Parameters

`pToneState`                      Pointer to the tone generator specification structure.

### Description

The function `ippsToneFree` is declared in the `ipps.h` file. This function closes the tone generator state by freeing all memory associated with the structure created by `ippsToneInitAllocQ15`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointer <i>pToneState</i> is NULL.

## ToneGetStateSizeQ15

*Computes the length of the tone generator structure.*

---

### Syntax

```
IppStatus ippstToneGetStateSizeQ15_16s(int* pToneStateSize);
```

### Parameters

<i>pToneStateSize</i>	Pointer to the computed value of size in bytes of the generator specification structure.
-----------------------	--

### Description

The function `ippstToneGetStateSizeQ15` is declared in the `ipps.h` file. This function computes the length (in bytes) *pToneStateSize* of the tone generator structure that is used by the function `ippstToneInitQ15`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointer <i>pToneStateSize</i> is NULL.

## ToneInitQ15

*Initializes the tone generator specification structure for fixed point data.*

---

### Syntax

```
IppStatus ippstToneInitQ15_16s(IppToneState_16s* pToneState, Ipp16s magn,  
Ipp16s rFreqQ15, Ipp32s phaseQ15);
```



## Parameters

<i>pToneState</i>	Pointer to the tone generator specification structure.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].

## Description

The function `ippsToneInitQ15` is declared in the `ipps.h` file. This function initializes the tone generator structure *pToneState* with the specified frequency *rFreqQ15*, phase *phaseQ15*, and magnitude *magn*. The structure is allocated in the external buffer, the size of which must be computed by the function `ippsToneGetStateSizeQ15`. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0,  $2\pi$ ) for phase. Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format. Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pToneState</i> pointer is NULL.
<code>ippStsToneMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <i>phaseQ15</i> value is negative, or greater than 205886.

## ToneQ15

*Generates a tone with a frequency, phase, and magnitude specified in the tone generator structure.*

---

### Syntax

```
IppStatus ippsToneQ15_16s(Ipp16s* pDst, int len, IppToneState_16s* pToneState);
```

### Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.
<i>pToneState</i>	Pointer to the tone generator specification structure.

### Description

The function `ippsToneQ15` is declared in the `ipps.h` file. This function generates the tone with the frequency, phase, and magnitude parameters that are specified in the previously created structure *pToneState*. The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values  $x[n]$  are computed using the same formulas as in the function [ippsTone\\_Direct](#) for computing real tones.

Example 4-5 below demonstrates how to use the function `ippsToneQ15`.

### Return Values

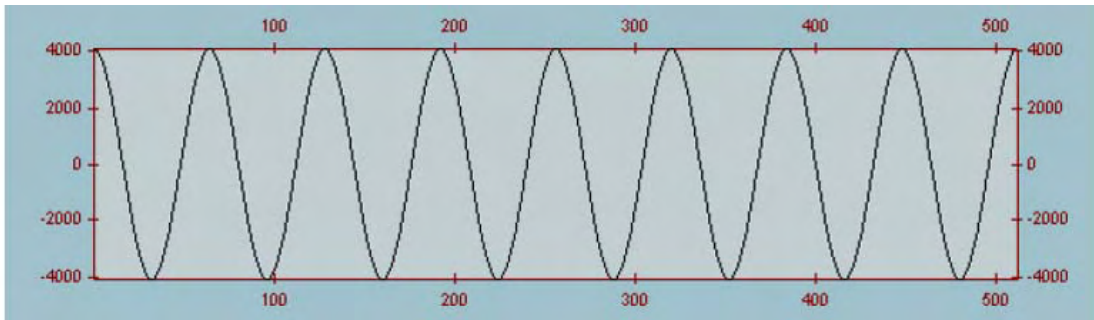
<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pToneState</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

### Example 4-5 Using Tone Generating Functions

```
void func_toneq15()
{
    IppToneState_16s *TS;
```

```
Ipp16s magn = 4095;
Ipp16s rFreqQ15 = 512;
Ipp16s phaseQ15 = 0;
Ipp16s* pDst;
IppStatus status;

status = ippsToneInitAllocQ15_16s(&TS,magn,rFreqQ15,phaseQ15);
if(ippStsNoErr != status)
    printf("IPP Error: %s",ippGetStatusString(status));
status = ippsToneQ15_16s(pDst,rFreqQ15,TS);
if(ippStsNoErr != status)
    printf("IPP Error: %s",ippGetStatusString(status));
}
result:
```



## Tone\_Direct

*Generates a tone with a given frequency, phase, and magnitude.*

---

### Syntax

```

IppStatus ippsTone_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, float
rFreq, float* pPhase, IppHintAlgorithmhint);

IppStatus ippsTone_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn, float
rFreq, float* pPhase, IppHintAlgorithmhint);

IppStatus ippsTone_Direct_32f(Ipp32f* pDst, int len, float magn, float rFreq,
float* pPhase, IppHintAlgorithmhint);

IppStatus ippsTone_Direct_32fc(Ipp32fc* pDst, int len, float magn, float
rFreq, float* pPhase, IppHintAlgorithmhint);

IppStatus ippsTone_Direct_64f(Ipp64f* pDst, int len, double magn, double
rFreq, double* pPhase, IppHintAlgorithmhint);

IppStatus ippsTone_Direct_64fc(Ipp64fc* pDst, int len, double magn, double
rFreq, double* pPhase, IppHintAlgorithmhint);

```

### Parameters

<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>pPhase</i>	Pointer to the phase of the tone relative to a cosine wave. It must be in range $[0.0, 2\pi)$ . The returned value may be used to compute the next continuous data block.
<i>rFreq</i>	Frequency of the tone relative to the sampling frequency. It must be in the interval $[0.0, 0.5)$ for real tone and in $[0.0, 1.0)$ for complex tone.
<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.
<i>hint</i>	Suggests using specific code. The possible values for the <i>hint</i> argument are described in <a href="#">Hint Arguments</a> .

## Description

The function `ippsTone_Direct` is declared in the `ipps.h` file. This function generates the tone with the specified frequency *rFreq*, phase *pPhase*, and magnitude *magn*. The function computes *len* samples of the tone, and stores them in the array *pDst*. For real tones, each generated value  $x[n]$  is defined as:

$$x[n] = \text{magn} * \cos(2\pi n * r\text{Freq} + \text{phase})$$

For complex tones,  $x[n]$  is defined as:

$$x[n] = \text{magn} * (\cos(2\pi n * r\text{Freq} + \text{phase}) + j * \sin(2\pi n * r\text{Freq} + \text{phase}))$$

The parameter *hint* suggests using specific code, which provides for either fast but less accurate calculation, or more accurate but slower execution.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pPhase</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.
<code>ippStsToneMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <i>rFreq</i> is negative, or greater than or equal to 0.5 for real tone and to 1.0 for complex tone.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <i>pPhase</i> value is negative, or greater than or equal to <code>IPP_2PI</code> .

## ToneQ15\_Direct

*Generates a tone with a given frequency, phase, and magnitude.*

---

### Syntax

```
Ippl6s* ippsToneQ15_Direct_16s(Ippl6s* pDst, int len, Ippl6s magn, Ippl6s rFreqQ15, Ippl6s phaseQ15);
```

### Parameters

*pDst*                      Pointer to the array which stores the samples.

<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].

## Description

The function `ippsToneQ15_Direct` is declared in the `ipps.h` file. This function generates the tone with the specified frequency *rFreqQ15*, phase *pPhaseQ15*, and magnitude *magn*. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for relative frequency and [0, 2 $\pi$ ) for phase.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format. Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values *x[n]* are computed using the same formulas as in the function `ippsTone_Direct` for computing real tones.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.
<code>ippStsToneMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <i>phaseQ15</i> value is negative, or greater than 205886.

## Triangle-Generating Functions

This section describes the functions that generate a periodic signal with a triangular wave form (referred to as "triangle") of a given frequency, phase, magnitude, and asymmetry.

A real periodic signal with triangular wave form  $x[n]$  (referred to as a real triangle) of a given frequency  $rFreq$ , phase value  $phase$ , magnitude  $magn$ , and asymmetry  $h$  is defined as follows:

$$x[n] = magn * \text{ct}_h(2\pi * rFreq * n + phase), n = 0, 1, 2, \dots$$

A complex periodic signal with triangular wave form  $x[n]$  (referred to as a complex triangle) of a given frequency  $rFreq$ , phase value  $phase$ , magnitude  $magn$ , and asymmetry  $h$  is defined as follows:

$$x[n] = magn * [\text{ct}_h(2\pi * rFreq * n + phase) + j * \text{st}_h(2\pi * rFreq * n + phase)], n = 0, 1, 2, \dots$$

The  $\text{ct}_h()$  function is determined as follows:

$$H = \pi + h$$

$$\text{ct}_h(\alpha) = \begin{cases} -\frac{2}{H} \cdot \left(\alpha - \frac{H}{2}\right), & 0 \leq \alpha \leq H \\ \frac{2}{2\pi - H} \cdot \left(\alpha - \frac{2\pi + H}{2}\right), & H \leq \alpha \leq \pi \end{cases}$$

$$\text{ct}_h(\alpha + k \cdot 2\pi) = \text{ct}_h(\alpha), k = 0, \pm 1, \pm 2, \dots$$

$$\text{ct}_h(\alpha + k \cdot 2\pi) = \text{ct}_h(\alpha), k = 0, \pm 1, \pm 2, \dots$$

When  $H = \pi$ , asymmetry  $h = 0$ , and function  $\text{ct}_h()$  is symmetric and a triangular analog of the **cos()** function. Note the following equations:

$$\text{ct}_h(H/2 + k \cdot \pi) = 0, k = 0, \pm 1, \pm 2, \dots$$

$$\text{ct}_h(k \cdot 2\pi) = 1, k = 0, \pm 1, \pm 2, \dots$$

$$\text{ct}_h(H + k \cdot 2\pi) = -1, k = 0, \pm 1, \pm 2, \dots$$

The  $\text{st}_h()$  function is determined as follows:

$$\mathbf{st}_h(\alpha) = \begin{cases} \frac{2}{2\pi-H} \cdot \alpha, & 0 \leq \alpha \leq \frac{2\pi-H}{2} \\ -\frac{2}{H} \cdot (\alpha - \pi), & \frac{2\pi-H}{2} \leq \alpha \leq \frac{2\pi+H}{2} \\ \frac{2}{2\pi-H} \cdot (\alpha - 2\pi), & \frac{2\pi+H}{2} \leq \alpha \leq \pi \end{cases}$$

$$\mathbf{st}_h(\alpha + k \cdot 2\pi) = \mathbf{st}_h(\alpha), k = 0, \pm 1, \pm 2, \dots$$

When  $H = \pi$ , asymmetry  $h = 0$ , and function  $\mathbf{st}_h()$  is symmetric and a triangular analog of the sine function. Note the following equations:

$$\mathbf{st}_h(\alpha) = \mathbf{ct}_h(\alpha + (3\pi + h)/2), k = 0, \pm 1, \pm 2, \dots$$

$$\mathbf{st}_h(k \cdot \pi) = 0, k = 0, \pm 1, \pm 2, \dots$$

$$\mathbf{st}_h((\pi - h)/2 + k \cdot 2\pi) = 1, k = 0, \pm 1, \pm 2, \dots$$

$$\mathbf{st}_h((3\pi + h)/2 + k \cdot 2\pi) = -1, k = 0, \pm 1, \pm 2, \dots$$

## TriangleInitAllocQ15

*Allocates memory and initializes the triangle generator specification structure for fixed point data.*

### Syntax

```
IpStatus ippsTriangleInitAllocQ15_16s(IppTriangleState_16s** pTriangleState,
Ippl6s magn, Ippl6s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

### Parameters

<i>pTriangleState</i>	Pointer to the pointer to the triangle generator specification structure.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].



<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>asymQ15</i>	Asymmetry <i>h</i> of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If <i>h</i> =0, then the triangle is symmetric and a direct analog of a tone.

## Description

The function `ippsTriangleInitAllocQ15` is declared in the `ipps.h` file. This function allocates memory and initializes the triangle generator structure `pTriangleState` with the specified frequency *rFreqQ15*, phase *phaseQ15*, asymmetry *asymQ15* and magnitude *magn*. Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2 $\pi$ ) for the phase, and (- $\pi$ ,  $\pi$ ) for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format. Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pTriangleState</code> pointer is NULL.
<code>ippStsTriangleMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsTriangleFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.
<code>ippStsTrianglePhaseErr</code>	Indicates an error when the <i>phaseQ15</i> value is negative, or greater than 205886.
<code>ippStsTriangleAsymErr</code>	Indicates an error when the <i>asymQ15</i> value is less than -102943 or greater than 102943.

## TriangleFree

*Frees memory allocated by the function `ippsTriangleInitAlloc`.*

## Syntax

```
IppStatus ippsTriangleFree(IppTriangleState_16s* pTriangleState);
```

## Parameters

*pTriangleState*      Pointer to the triangle generator specification structure.

## Description

The function `ippsTriangleFree` is declared in the `ipps.h` file. This function closes the triangle generator state by freeing all memory associated with the structure created by `ippsTriangleInitAllocQ15`.

## Return Values

`ippStsNoErr`            Indicates no error.  
`ippStsNullPtrErr`      Indicates an error when the pointer *pTriangleState* is NULL.

## TriangleGetStateSizeQ15

*Computes the length of the triangle generator structure.*

---

## Syntax

```
IppStatus ippsTriangleGetStateSizeQ15_16s(int* pTriangleStateSize);
```

## Parameters

*pTriangleStateSize*    Pointer to the computed value of size in bytes of the generator specification structure.

## Description

The function `ippsTriangleGetStateSizeQ15` is declared in the `ipps.h` file. This function computes the length (in bytes) *pTriangleStateSize* of the triangle generator structure that is used by the function `ippsTriangleInitQ15`.

## Return Values

`ippStsNoErr`            Indicates no error.  
`ippStsNullPtrErr`      Indicates an error when the pointer *pTriangleStateSize* is NULL.

## TriangleInitQ15

*Initializes the triangle generator specification structure for fixed point data.*

---

### Syntax

```
IppStatus ippsTriangleInitQ15_16s(IppTriangleState_16s* pTriangleState,
Ipp16s magn, Ipp16s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

### Parameters

<i>pTriangleState</i>	Pointer to the triangle generator specification structure.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>asymQ15</i>	Asymmetry $h$ of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If $h=0$ , then the triangle is symmetric and a direct analog of a tone.

### Description

The function `ippsTriangleInitQ15` is declared in the `ipps.h` file. This function initializes the triangle generator structure *pTriangleState* with the specified magnitude *magn*, frequency *rFreqQ15*, phase *phaseQ15*, and asymmetry *asymQ15*. The structure is allocated in the external buffer, the size of which must be computed by the function `ippsTriangleGetStateSizeQ15`. Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2 $\pi$ ) for the phase, and (- $\pi$ ,  $\pi$ ) for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format. Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTriangleState</i> pointer is NULL.
<code>ippStsTriangleMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippStsTriangleFreqErr</code>	Indicates an error when <i>rFreqQ15</i> is negative, or greater than 16383.
<code>ippStsTrianglePhaseErr</code>	Indicates an error when the <i>phaseQ15</i> value is negative, or greater than 205886.
<code>ippStsTriangleAsymErr</code>	Indicates an error when the <i>asymQ15</i> value is less than -102943 or greater than 102943.

## TriangleQ15

*Generates a triangle with a frequency, phase, and magnitude specified in the triangle generator structure.*

---

### Syntax

```
IppStatus ippstTriangleQ15_16s(Ipp16s* pDst, int len, IppTriangleState_16s* pTriangleState);
```

### Parameters

<i>pDst</i>	Pointer to the array which stores the generated samples.
<i>len</i>	Number of samples to be computed.
<i>pTriangleState</i>	Pointer to the triangle generator specification structure.

### Description

The function `ippstTriangleQ15` is declared in the `ipps.h` file. This function generates the triangle with the frequency, phase, magnitude, and asymmetry parameters that are specified in the previously created structure *pTriangleState*. The function computes *len* samples of the triangle, and stores them in the array *pDst*. Generated values *x[n]* are computed using the same formulas as in `ippstTriangle_Direct` function for computing real triangles.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pToneState</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## Triangle\_Direct

*Generates a triangle with a given frequency, phase, and magnitude.*

---

### Syntax

```
IppStatus ippsTriangle_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn, float rFreq, float asym, float* pPhase);
```

```
IppStatus ippsTriangle_Direct_16sc(Ipp16sc* pDst, int len, Ipp16s magn, float rFreq, float asym, float* pPhase);
```

```
IppStatus ippsTriangle_Direct_32f(Ipp32f* pDst, int len, float magn, float rFreq, float asym, float* pPhase);
```

```
IppStatus ippsTriangle_Direct_32fc(Ipp32fc* pDst, int len, float magn, float rFreq, float asym, float* pPhase);
```

```
IppStatus ippsTriangle_Direct_64f(Ipp64f* pDst, int len, double magn, double rFreq, double asym, double* pPhase);
```

```
IppStatus ippsTriangle_Direct_64fc(Ipp64fc* pDst, int len, double magn, double rFreq, double asym, double* pPhase);
```

### Parameters

<code>rFreq</code>	Frequency of the triangle relative to the sampling frequency. It must be in range $[0.0, 0.5]$ .
<code>pPhase</code>	Pointer to the phase of the triangle relative to a cosine triangular analog wave. It must be in range $[0.0, 2\pi]$ . The returned value may be used to compute the next continuous data block.
<code>magn</code>	Magnitude of the triangle, that is, the maximum value attained by the wave.

<i>asym</i>	Asymmetry $h$ of a triangle. It must be in range $[-\pi, \pi)$ . If $h=0$ , then the triangle is symmetric and a direct analog of a tone.
<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.

## Description

The function `ippsTriangle_Direct` is declared in the `ipps.h` file. This function generates the triangle with the specified frequency *rFreq*, phase pointed by *pPhase*, and magnitude *magn*. The function computes *len* samples of the triangle, and stores them in the array *pDst*. For real triangle,  $x[n]$  is defined as:

$$x[n] = \text{magn} * \text{ct}_h(2\pi * r\text{Freq} * n + \text{phase}), n = 0, 1, 2, \dots$$

For complex triangles,  $x[n]$  is defined as:

$$x[n] = \text{magn} * [\text{ct}_h(2\pi * r\text{Freq} * n + \text{phase}) + j * \text{st}_h(2\pi * r\text{Freq} * n + \text{phase})], n = 0, 1, 2, \dots$$

See [Triangle Generating Functions](#) for the definition of functions `cth` and `sth`.

Example 4-6 below demonstrates how to use the function `ippsTriangle`.

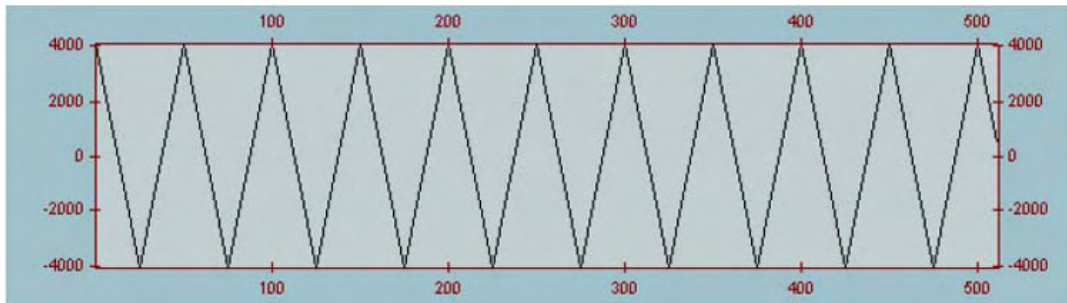
## Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pPhase</i> pointer is NULL.
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.
<code>ippsStsTrnglMagnErr</code>	Indicates an error when <i>magn</i> is less than or equal to zero.
<code>ippsStsTrnglFreqErr</code>	Indicates an error when <i>rFreq</i> is negative, or greater than or equal to 0.5.
<code>ippsStsTrnglPhaseErr</code>	Indicates an error when the <i>pPhase</i> value is negative, or greater than or equal to <code>IPP_2PI</code> .
<code>ippsStsTrnglAsymErr</code>	Indicates an error when <i>asym</i> is less than <code>-IPP_PI</code> , or greater than or equal to <code>IPP_PI</code> .

### Example 4-6 Using the Triangle Generating Functions

```
void func_triangle_direct()
{
    Ipp16s* pDst;
    int len = 512;
    Ipp16s magn = 4095;
    float rFreq = 0.02;
    float asym = 0.0;;
    float Phase = 0.0;
    IppStatus status;
    status = ippsTriangle_Direct_16s(pDst, len, magn, rFreq, asym, &Phase);
    if(ippStsNoErr != status)
        printf("IPP Error: %s",ippGetStatusString(status));
}

result:
```



## TriangleQ15\_Direct

*Generates a triangle with a given frequency, phase, and magnitude for fixed point data.*

---

### Syntax

```
IppStatus ippsTriangleQ15_Direct_16s(Ipp16s* pDst, int len, Ipp16s magn,
Ipp16s rFreqQ15, Ipp32s phaseQ15, Ipp32s asymQ15);
```

### Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>magn</i>	Magnitude of the tone, that is, the maximum value attained by the wave.
<i>rFreqQ15</i>	Frequency of the tone relative to the sampling frequency in Q0.15 format. It must be in the range [0, 16383].
<i>phaseQ15</i>	Phase of the tone relative to a cosine wave in Q16.15 format. It must be in the range [0, 205886].
<i>asymQ15</i>	Asymmetry $h$ of a triangle in Q16.15 format. It must be in the range [-102943, 102943]. If $h=0$ , then the triangle is symmetric and a direct analog of a tone.

### Description

The function `ippsTriangleQ15_Direct` is declared in the `ipps.h` file. This function generates the triangle with the specified magnitude *magn*, frequency *rFreqQ15*, phase *pPhaseQ15*, and asymmetry *asymQ15*. Input data in the fixed point format Q15 format are converted to the corresponding float data type that lay in the range [0, 0.5) for the relative frequency, [0, 2 $\pi$ ) for the phase, and [- $\pi$ ,  $\pi$ ) for the asymmetry.

Q16.15 designates that 16 bits before and 15 bits after fixed point position are used to present a 32-bit value in the fixed point format. Q0.15 designates that 0 bits before and 15 bits after fixed point position are used to present a 16-bit value in the fixed point format.

The function computes *len* samples of the tone, and stores them in the array *pDst*. Generated values *x[n]* are computed using the same formulas as in the function `ippsTriangle_Direct` for computing real triangles.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.
<code>ippStsToneMagnErr</code>	Indicates an error when <code>magn</code> is less than or equal to zero.
<code>ippStsToneFreqErr</code>	Indicates an error when <code>rFreqQ15</code> is negative, or greater than 16383.
<code>ippStsTonePhaseErr</code>	Indicates an error when the <code>phaseQ15</code> value is negative, or greater than 205886.
<code>ippStsTriangleAsymErr</code>	Indicates an error when the <code>asymQ15</code> value is less than -102943 or greater than 102943.

## Uniform Distribution Functions

This section describes the functions that generate pseudo-random samples with uniform distribution.

### RandUniformInitAlloc

*Allocates memory and initializes a noise generator with uniform distribution.*

#### Syntax

```

IppStatus ippsRandUniformInitAlloc_8u(IppsRandUniState_8u** pRandUniState,
Ipp8u low, Ipp8u high, unsigned int seed);

IppStatus ippsRandUniformInitAlloc_16s(IppsRandUniState_16s** pRandUniState,
Ipp16s low, Ipp16s high, unsigned int seed);

IppStatus ippsRandUniformInitAlloc_32f(IppsRandUniState_32f** pRandUniState,
Ipp32f low, Ipp32f high, unsigned int seed);

```

#### Parameters

<code>pRandUniState</code>	Pointer to the structure containing parameters for the generator of noise.
<code>low</code>	Lower bound of the uniform distribution range.

<i>high</i>	Upper bound of the uniform distribution range.
<i>seed</i>	Seed value used by the pseudo-random number generation algorithm.

## Description

The function `ippsRandUniformInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the pseudo-random generator state `pRandUniState`. The uniform distribution range is specified by the lower and upper bounds `low` and `high`, respectively.

Code [example 4-7](#) demonstrates how to use the function `ippsRandUniformInitAlloc`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRandUniState</code> pointer is <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for the operation.

## RandUniformFree

Closes the uniform distribution generator state.

### Syntax

```
IppStatus ippsRandUniformFree_8u(IppsRandUniState_8u* pRandUniState);
IppStatus ippsRandUniformFree_16s(IppsRandUniState_16s* pRandUniState);
IppStatus ippsRandUniformFree_32f(IppsRandUniState_32f* pRandUniState);
```

### Parameters

<i>pRandUniState</i>	Pointer to the structure containing parameters for the generator of noise.
----------------------	--

### Description

The function `ippsRandUniformFree` is declared in the `ipps.h` file. This function closes the noise generator state `pRandUniState` by freeing all memory allocated by the function [ippsRandUniformInitAlloc](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRandUniState</code> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## RandUniformInit

*Initializes a noise generator with uniform distribution.*

---

### Syntax

```
IppStatus ippRandUniformInit_16s(IppsRandUniState_16s* pRandUniState, Ipp16s low, Ipp16s high, unsigned int seed);
```

### Parameters

<code>pRandUniState</code>	Pointer to the structure containing parameters for the generator of noise.
<code>low</code>	Lower bound of the uniform distribution range.
<code>high</code>	Upper bound of the uniform distribution range.
<code>seed</code>	Seed value used by the pseudo-random number generation algorithm.

### Description

The function `ippRandUniformInit` is declared in the `ipps.h` file. This function initializes the pseudo-random generator state `pRandUniState` in the external buffer. The size of this buffer must be computed previously by calling the function [ippRandUniformGetSize](#). The uniform distribution range is specified by the lower and upper bounds `low` and `high`, respectively.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRandUniState</code> pointer is <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for the operation.

## RandUniformGetSize

*Computes the length of the uniform distribution generator structure.*

---

### Syntax

```
IppStatus ippsRandUniformGetSize_16s(int* pRandUniStateSize);
```

### Parameters

*pRandUniStateSize*      Pointer to the computed value of size in bytes of the generator specification structure.

### Description

The function `ippsRandUniformGetSize` is declared in the `ipps.h` file. This function computes the length (in bytes) *pRandUniStateSize* of the uniform distribution generator structure that is used by the function `ippsRandUniformInit`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointer <i>pRandUniStateSize</i> is NULL.

## RandUniform

*Generates the pseudo-random samples with a uniform distribution.*

---

### Syntax

```
IppStatus ippsRandUniform_8u(Ipp8u* pDst, int len, IppsRandUniState_8u* pRandUniState);
```

```
IppStatus ippsRandUniform_16s(Ipp16s* pDst, int len, IppsRandUniState_16s* pRandUniState);
```

```
IppStatus ippsRandUniform_32f(Ipp32f* pDst, int len, IppsRandUniState_32f* pRandUniState);
```

## Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.
<i>pRandUniState</i>	Pointer to the structure containing parameters for the generator of noise.

## Description

The function `ippsRandUniform` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a uniform distribution and stores them in the array *pDst*. Initial parameters of the generator are set in the generator state structure *pRandUniState*. Before calling `ippsRandUniform`, you must initialize the generator state by calling the function `ippsRandUniformInitAlloc`.

Example 4-7 below demonstrates how to use the function `ippsRandUniform`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pRandUniState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## Example 4-7 Using the Function to Generate Pseudo-Random Signal

```
void func_randuniform()
{
    IppsRandUniState_16s*
pRUS;

    Ipp16s low, high;
    low = -4096;
    high = 4095;

    unsigned int seed
= 0;

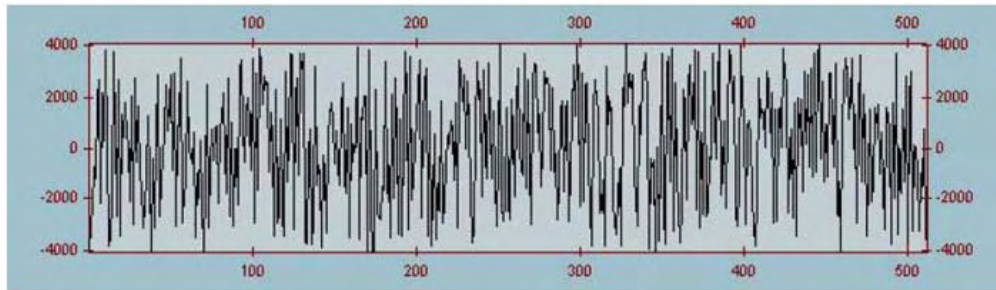
    Ipp16s* pDst;
    int len = 512;

    IppStatus status;

    status = ippsRandUniformInitAlloc_16s(&pRUS,low,high,seed);
    if(ippStsNoErr !=
status)
        printf("IPP Error: %s",ippGetStatusString(status));

    status = ippsRandUniform_16s(pDst,len,
pRUS);
    if(ippStsNoErr !=
status)
        printf("IPP Error: %s",ippGetStatusString(status));
}

result:
```



## RandUniform\_Direct

*Generates the pseudo-random samples with a uniform distribution in direct mode.*

### Syntax

```
IppStatus ippsRandUniform_Direct_16s(Ipp16s* pDst, int len, Ipp16s low,
Ipp16s high, unsigned int* pSeed);
```

```
IppStatus ippsRandUniform_Direct_32f(Ipp32f* pDst, int len, Ipp32f low,
Ipp32f high, unsigned int* pSeed);
```

```
IppStatus ippsRandUniform_Direct_64f(Ipp64f* pDst, int len, Ipp64f low,
Ipp64f high, unsigned int* pSeed);
```

### Parameters

<i>pSeed</i>	Pointer to the seed value used by the pseudo-random number generation algorithm.
<i>low</i>	Lower bound of the uniform distribution range.
<i>high</i>	Upper bound of the uniform distribution range.
<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.

## Description

The function `ippsRandUniform_Direct` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a uniform distribution and stores them in the array *pDst*. This function does not require to initialize the generator state structure in advance. All parameters of the pseudo-random number generator are set directly in the function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSeed</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Gaussian Distribution Functions

This section describes the function that generates pseudo-random samples with Gaussian distribution.

## RandGaussInitAlloc

*Allocates memory and initializes a noise generator with Gaussian distribution.*

---

## Syntax

```
IppStatus ippsRandGaussInitAlloc_8u(IppsRandGaussState_8u** pRandGaussState,
Ipp8u mean, Ipp8u stdDev, unsigned int seed);

IppStatus ippsRandGaussInitAlloc_16s(IppsRandGaussState_16s** pRandGaussState,
Ipp16s mean, Ipp16s stdDev, unsigned int seed);

IppStatus ippsRandGaussInitAlloc_32f(IppsRandGaussState_32f** pRandGaussState,
Ipp32f mean, Ipp32f stdDev, unsigned int seed);
```

## Parameters

<i>pRandGaussState</i>	Pointer to the structure containing parameters for the generator of noise.
<i>mean</i>	Mean of the Gaussian distribution.
<i>stdDev</i>	Standard deviation of the Gaussian distribution.



*seed* Seed value used by the pseudo-random number generator algorithm.

### Description

The function `ippsRandGaussInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the pseudo-random generator state structure `pRandGaussState`. This structure contains parameters of the required noise generator that are specified by the *mean*, *stdDev* and *seed* values.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRandGaussState</code> pointer is <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for the operation.

## RandGaussFree

Closes the Gaussian distribution generator state.

### Syntax

```
IppStatus ippsRandGaussFree_8u(IppsRandGaussState_8u* pRandGaussState);
IppStatus ippsRandGaussFree_16s(IppsRandGaussState_16s* pRandGaussState);
IppStatus ippsRandGaussFree_32f(IppsRandGaussState_32f* pRandGaussState);
```

### Parameters

*pRandGaussState* Pointer to the structure containing parameters for the generator of noise.

### Description

The function `ippsRandGaussFree` is declared in the `ipps.h` file. This function closes the noise generator state `pRandGaussState` by freeing all memory allocated by the function `ippsRandGaussInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when the `pRandGaussState` pointer is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## RandGaussInit

*Initializes a noise generator with Gaussian distribution.*

---

### Syntax

```
IppStatus ippRandGaussInit_16s(IppsRandGaussState_16s* pRandGaussState,
Ipp16s mean, Ipp16s stdDev, unsigned int seed);
```

### Parameters

<code>pRandGaussState</code>	Pointer to the structure containing parameters for the generator of noise.
<code>mean</code>	Mean of the Gaussian distribution.
<code>stdDev</code>	Standard deviation of the Gaussian distribution.
<code>seed</code>	Seed value used by the pseudo-random number generator algorithm.

### Description

The function `ippRandGaussInit` is declared in the `ipps.h` file. This function initializes the pseudo-random generator state structure `pRandGaussState` in the external buffer. The size of this buffer must be computed previously by calling the function [ippRandGaussGetSize](#). This structure contains parameters of the required noise generator that are specified by the `mean`, `stdDev` and `seed` values.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRandGaussState</code> pointer is <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for the operation.

## RandGaussGetSize

*Computes the length of the Gaussian distribution generator structure.*

---

### Syntax

```
IppStatus ippsRandGaussGetSize_16s(int* pRandGaussStateSize);
```

### Parameters

*pRandGaussStateSize* Pointer to the computed value of size in bytes of the generator specification structure.

### Description

The function `ippsRandGaussGetSize` is declared in the `ipps.h` file. This function computes the length (in bytes) *pRandGaussStateSize* of the uniform distribution generator structure that is used by the function `ippsRandGaussInit`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointer <i>pRandGaussStateSize</i> is NULL.

## RandGauss

*Generates the pseudo-random samples with a Gaussian distribution.*

---

### Syntax

```
IppStatus ippsRandGauss_8u(Ipp8u* pDst, int len, IppsRandGaussState_8u* pRandGaussState);
```

```
IppStatus ippsRandGauss_16s(Ipp16s* pDst, int len, IppsRandGaussState_16s* pRandGaussState);
```

```
IppStatus ippsRandGauss_32f(Ipp32f* pDst, int len, IppsRandGaussState_32f* pRandGaussState);
```

## Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
<i>len</i>	Number of samples to be computed.
<i>pRandGaussState</i>	Pointer to the structure containing parameters of the noise generator.

## Description

The function `ippsRandGauss` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a Gaussian distribution and stores them in the array *pDst*. The initial parameters of the generator are set in the generator state structure *pRandGaussState*. Before calling `ippsRandGauss`, you must initialize the generator state by calling the `ippsRandGaussInitAlloc` function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pRandGaussState</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## RandGauss\_Direct

*Generates pseudo-random samples with a Gaussian distribution in the direct mode.*

---

### Syntax

```

IppStatus ippsRandGauss_Direct_16s(Ipp16s* pDst, int len, Ipp16s mean, Ipp16s
stdev, unsigned int* pSeed);

IppStatus ippsRandGauss_Direct_32f(Ipp32f* pDst, int len, Ipp32f mean, Ipp32f
stdev, unsigned int* pSeed);

IppStatus ippsRandGauss_Direct_64f(Ipp64f* pDst, int len, Ipp64f mean, Ipp64f
stdev, unsigned int* pSeed);

```

### Parameters

<i>pDst</i>	Pointer to the array which stores the samples.
-------------	--

<i>pSeed</i>	Pointer to the seed value used by the pseudo-random number generation algorithm.
<i>len</i>	Number of samples to be computed.
<i>mean</i>	Mean of the Gaussian distribution.
<i>stdev</i>	Standard deviation of the Gaussian distribution.

## Description

The function `ippsRandGauss_Direct` is declared in the `ipps.h` file. This function generates *len* pseudo-random samples with a Gaussian distribution, and stores them in the array *pDst*. This function does not require to initialize the generator state structure in advance. All parameters of the pseudo-random number generator are set directly in the function.

Example 4-8 below demonstrates how to use the function `ippsRandGauss_Direct`.

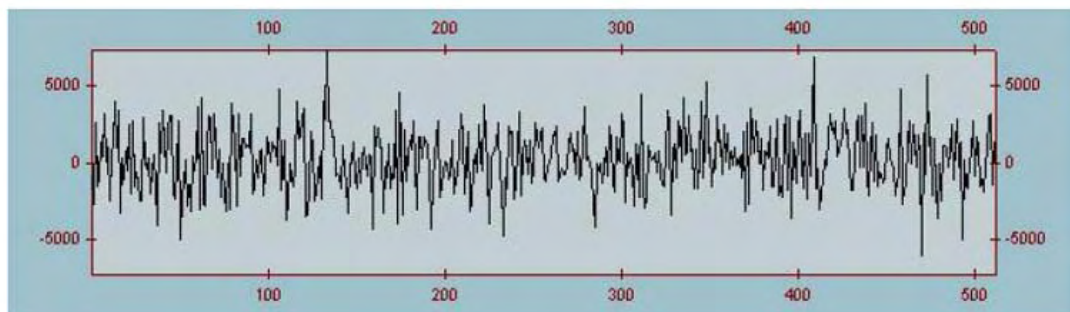
## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSeed</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 4-8 Using the Function `ippsaRandGauss_Direct`

```
void func_gauss_direct()
{
    Ipp16s* pDst;
    int len = 512;
    Ipp16s mean = 0;
    Ipp16s stdev = 2047;
    unsigned int Seed = 0;
    IppStatus status;
    status = ippsRandGauss_Direct_16s(pDst, len, mean, stdev, &Seed);
    if(ippStsNoErr != status)
        printf("IPP Error: %s",ippGetStatusString(status));
}

result:
```



## Special Vector Functions

The functions described in this section create special vectors that can be used as a test signals to examine the effect of applying different signal processing functions.

## VectorJaehne

*Creates a Jaehne vector.*

---

### Syntax

```

IppStatus ippsVectorJaehne_8u(Ipp8u* pDst, int len, Ipp8u magn);
IppStatus ippsVectorJaehne_8s(Ipp8s* pDst, int len, Ipp8s magn);
IppStatus ippsVectorJaehne_16u(Ipp16u* pDst, int len, Ipp16u magn);
IppStatus ippsVectorJaehne_16s(Ipp16s* pDst, int len, Ipp16s magn);
IppStatus ippsVectorJaehne_32u(Ipp32u* pDst, int len, Ipp32u magn);
IppStatus ippsVectorJaehne_32s(Ipp32s* pDst, int len, Ipp32s magn);
IppStatus ippsVectorJaehne_32f(Ipp32f* pDst, int len, Ipp32f magn);
IppStatus ippsVectorJaehne_64f(Ipp64f* pDst, int len, Ipp64f magn);
    
```

### Parameters

*pDst*                                      Pointer to the destination vector.

*len*                      Number of elements in the vector.  
*magn*                    Magnitude of the signal to be generated.

### Description

The function `ippsVectorJaehne` is declared in the `ipps.h` file. This function creates a Jaehne vector and stores the result in *pDst*. The magnitude *magn* must be positive. The function generates the sinusoid with a variable frequency. The computation is performed as follows:

$$pDst[n] = magn * \sin((0.5\pi n^2)/len), 0 \leq n < len$$

Example 4-9 below shows how to use the function `ippsVectorJaehne`.

### Return Values

`ippStsNoErr`            Indicates no error.  
`ippStsNullPtrErr`    Indicates an error when the *pSrcDst* pointer is NULL.  
`ippStsSizeErr`        Indicates an error when *len* is less than or equal to 0.  
`ippStsJaehneErr`      Indicates an error when *magn* is negative.

### Example 4-9 Using the `ippsVectorJaehne` Function

```
IppStatus Jaehne (void)
{
    Ipp16s buf[100] ;
    return ippsVectorJaehne_16s ( buf, 100, 255 );
}
```

## VectorSlope

*Creates a slope vector.*

---

### Syntax

```
IppStatus ippsVectorSlope_8u(Ipp8u* pDst, int len, Ipp32f offset, Ipp32f
slope);

IppStatus ippsVectorSlope_8s(Ipp8s* pDst, int len, Ipp32f offset, Ipp32f
slope);

IppStatus ippsVectorSlope_16u(Ipp16u* pDst, int len, Ipp32f offset, Ipp32f
slope);
```

```

IppStatus ippsVectorSlope_16s(Ipp16s* pDst, int len, Ipp32f offset, Ipp32f
slope);

IppStatus ippsVectorSlope_32u(Ipp32u* pDst, int len, Ipp64f offset, Ipp64f
slope);

IppStatus ippsVectorSlope_32s(Ipp32s* pDst, int len, Ipp64f offset, Ipp64f
slope);

IppStatus ippsVectorSlope_32f(Ipp32f* pDst, int len, Ipp32f offset, Ipp32f
slope);

IppStatus ippsVectorSlope_64f(Ipp64f* pDst, int len, Ipp64f offset, Ipp64f
slope);

```

## Parameters

<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.
<i>offset</i>	Offset value.
<i>slope</i>	Slope coefficient.

## Description

The function `ippsVectorSlope` is declared in the `ipps.h` file. This function creates a slope vector and stores the result in *pDst*. The destination vector elements are computed according to the following formula:

$$pDst[n] = offset + slope * n, 0 \leq n < len.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.



## VectorRamp

*Creates a ramp vector.*

---

### Syntax

```

IppStatus ippsVectorRamp_8u(Ipp8u* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_8s(Ipp8s* pDst, int len, float offset, float slope);
IppStatus ippsVectorRamp_16u(Ipp16u* pDst, int len, float offset, float
slope);
IppStatus ippsVectorRamp_16s(Ipp16s* pDst, int len, float offset, float
slope);
IppStatus ippsVectorRamp_32u(Ipp32u* pDst, int len, float offset, float
slope);
IppStatus ippsVectorRamp_32s(Ipp32s* pDst, int len, float offset, float
slope);
IppStatus ippsVectorRamp_32f(Ipp32f* pDst, int len, float offset, float
slope);
IppStatus ippsVectorRamp_64f(Ipp64f* pDst, int len, float offset, float
slope);

```

### Parameters

<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.
<i>offset</i>	Offset value.
<i>slope</i>	Slope coefficient.

### Description

The function `ippsVectorRamp` is declared in the `ipps.h` file. This function creates a ramp vector and stores the result in `pDst`. The destination vector elements are computed according to the following formula:

$$pDst[n] = offset + slope * n, 0 \leq n < len.$$

Note that this function is similar to the function `ippsVectorSlope`, but the linear transform coefficients *offset* and *slope* have floating-point values for all flavors of the function `ippsVectorRamp`. In most cases the use of the function `ippsVectorSlope` is more preferable.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

# Essential Functions

This chapter describes the Intel® IPP functions that perform logical and shift, arithmetic, conversion, windowing, and statistical operations.

The full list of functions in this group is given in Table 5-1 below.

**Table 5-1 Intel IPP Essential Vector Functions**

Function Base Name	Operation
Logical and Shift Functions	
<a href="#">AndC</a>	Computes the bitwise AND of a scalar value and each element of a vector.
<a href="#">And</a>	Computes the bitwise AND of two vectors.
<a href="#">OrC</a>	Computes the bitwise OR of a scalar value and each element of a vector.
<a href="#">Or</a>	Computes the bitwise OR of two vectors.
<a href="#">XorC</a>	Computes the bitwise XOR of a scalar value and each element of a vector.
<a href="#">Xor</a>	Computes the bitwise XOR of two vectors.
<a href="#">Not</a>	Computes the bitwise NOT of the vector elements.
<a href="#">LShiftC</a>	Shifts bits in vector elements to the left.
<a href="#">RShiftC</a>	Shifts bits in vector elements to the right.
Arithmetic Functions	
<a href="#">AddC</a>	Adds a constant value to each element of a vector.
<a href="#">Add</a>	Adds the elements of two vectors.
<a href="#">AddProduct</a>	Adds product of two vectors to the accumulator vector.
<a href="#">AddProductC</a>	Adds product of a vector and a constant to the accumulator vector.

Function Base Name	Operation
MulC	Multiplies each elements of a vector by a constant value.
Mul	Multiplies the elements of two vectors.
SubC	Subtracts a constant value from each element of a vector.
SubCRev	Subtracts each element of a vector from a constant value.
Sub	Subtracts the elements of two vectors.
DivC	Divides each element of a vector by a constant value.
DivCRev	Divides a constant value by each element of a vector.
Div	Divides the elements of two vectors.
Div_Round	Divides the elements of two vectors with rounding.
Abs	Computes absolute values of vector elements.
Sqr	Computes a square of each element of a vector.
Sqrt	Computes a square root of each element of a vector.
Cubrt	Computes cube root of each element of a vector.
Exp	Computes to the power of each element of a vector.
Ln	Computes the natural logarithm of each element of a vector.
10Log10	Computes the decimal logarithm of each element of a vector and multiplies it by 10.

Function Base Name	Operation
SumLn	Sums natural logarithms of each element of a vector.
Arctan	Computes the inverse tangent of each element of a vector.
Normalize	Normalizes elements of a real or complex vector using offset and division operations.
Cauchy, CauchyD, CauchyDD2	Computes Cauchy robust error function, and its first and second derivatives
Conversion Functions	
SortAscend, SortDescend	Rearranges all elements of a vector.
SortIndexAscend, SortIndexDescend	Rearranges all elements of the vector and their indexes.
SortRadixAscend, SortRadixDescend	Sorts all elements of a vector using radix sorting algorithm.
SortRadixIndexAscend, SortRadixIndexDescend	Indirectly sorts all elements of a vector using radix sorting algorithm.
SwapBytes	Reverses byte order of a vector.
Convert	Converts the data type of a vector and stores the results in a second vector.
Join	Converts the floating-point data of several vectors to integer data, and stores the results in a single vector.
JoinScaled	Converts with scaling the floating-point data of several vectors to integer data, and stores the results in a single vector.

Function Base Name	Operation
<a href="#">SplitScaled</a>	Converts with scaling the integer data of a vector to floating-point data, and stores the result in several vectors.
<a href="#">Conj</a>	Stores the complex conjugate values of a vector in a second vector or in-place.
<a href="#">ConjFlip</a>	Computes the complex conjugate of a vector and stores the result in reverse order.
<a href="#">Magnitude</a>	Computes the magnitudes of the elements of a complex vector.
<a href="#">MagSquared</a>	Computes the squared magnitudes of the elements of a complex vector.
<a href="#">Phase</a>	Computes the phase angles of elements of a complex vector.
<a href="#">PowerSpectr</a>	Computes the power spectrum of a complex vector.
<a href="#">Real</a>	Returns the real part of a complex vector in a second vector.
<a href="#">Imag</a>	Returns the imaginary part of a complex vector in a second vector.
<a href="#">RealToCplx</a>	Returns a complex vector constructed from the real and imaginary parts of two real vectors.
<a href="#">CplxToReal</a>	Returns the real and imaginary parts of a complex vector in two respective vectors.
<a href="#">DemodulateFM</a>	Converts frequency modulated signal into the initial demodulated form.

Function Base Name	Operation
Threshold, Threshold_LT, Threshold_GT, Threshold_LT- Val, Threshold_GTVal, Threshold_LTValGT- Val	Performs the threshold operation on the elements of a vector by limiting the element values by the specified value.
Threshold_LTAbs, Threshold_GTAbs	Performs the threshold operation on the absolute values of elements of a vector.
Threshold_LTInv	Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.
CartToPolar	Converts the elements of a complex vector to polar coordinate form.
PolarToCart	Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.
MaxOrder	Computes the maximum order of a vector.
Preemphasize	Computes preemphasis of a single precision real signal in-place.
Flip	Reverses the order of elements in a vector.
FindNearestOne	Finds an element of the table which is closest to the specified value.
FindNearest	Finds table elements that are closest to the elements of the specified vector.
Viterbi Decoder Functions	
GetVarPointDV	Fills the array with the information about points that are closest to the received point.
CalcStatesDV	Calculates the states of the Viterbi decoder.
BuildSymblTableDV4D	Fills the array with the information about possible 4D symbols.

Function Base Name	Operation
<a href="#">UpdatePathMetricsDV</a>	Searches for the state with the minimum path metric.
Windowing Functions	
<a href="#">WinBartlett</a>	Multiplies a vector by a Bartlett windowing function.
<a href="#">WinBlackman</a>	Multiplies a vector by a Blackman windowing function.
<a href="#">WinHamming</a>	Multiplies a vector by a Hamming windowing function.
<a href="#">WinHann</a>	Multiplies a vector by a Hann windowing function.
<a href="#">WinKaiser</a>	Multiplies a vector by a Kaiser windowing function.
Statistical Functions	
<a href="#">Sum</a>	Computes the sum of the elements of a vector.
<a href="#">Max</a>	Returns the maximum value of a vector.
<a href="#">MaxIndx</a>	Returns the maximum value of a vector and the index of the maximum element.
<a href="#">MaxAbs</a>	Returns the maximum absolute value of a vector.
<a href="#">MaxAbsIndx</a>	Returns the maximum absolute value of a vector and the index of the corresponding element
<a href="#">Min</a>	Returns the minimum value of a vector.
<a href="#">MinIndx</a>	Returns the minimum value of a vector and the index of the minimum element.
<a href="#">MinAbs</a>	Returns the minimum absolute value of a vector.



Function Base Name	Operation
MinAbsIndx	Returns the minimum absolute value of a vector and the index of the corresponding element
MinMax	Returns the maximum and minimum values of a vector.
MinMaxIndx	Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.
Mean	Computes the mean value of a vector.
StdDev	Computes the standard deviation value of a vector.
MeanStdDev	Computes the mean value and the standard deviation value of a vector.
Norm	Computes the C, L1, L2, or L2Sqr norm of a vector.
NormDiff	Computes the C, L1, L2, or L2Sqr norm of two vectors' difference.
DotProd	Computes the dot product of two vectors.
MaxEvery, MinEvery	Computes maximum or minimum value for each pair of elements of two vectors.
ZeroCrossing	Computes specific zero crossing measure.
CountInRange	Computes the number of elements of the vector whose values are in the specified range.
Sampling Functions SampleUp	Up-samples a signal, conceptually increasing its sampling rate by an integer factor.

Function Base Name	Operation
<a href="#">SampleDown</a>	Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.

## Logical and Shift Functions

This section describes the Intel IPP signal processing functions that perform logical and shift operations on vectors. Logical and shift functions are only defined for integer arguments.

For binary logical operations AND, OR and XOR, the following functions are provided:

AndC, OrC, XorC for vector-scalar operations;

And, Or, Xor for vector-vector operations.

### AndC

*Computes the bitwise AND of a scalar value and each element of a vector.*

#### Syntax

```

IppStatus ippsAndC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);

IppStatus ippsAndC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);

IppStatus ippsAndC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);

IppStatus ippsAndC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);

IppStatus ippsAndC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);

IppStatus ippsAndC_32u_I(Ipp32u val, Ipp32u* pSrcDst, int len);

```

#### Parameters

<i>val</i>	Input scalar value.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.

<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsAndC` is declared in the `ipps.h` file. This function computes the bitwise AND of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsAndC` compute the bitwise AND of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## And

*Computes the bitwise AND of two vectors.*

### Syntax

```

IppStatus ippsAnd_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
int len);

IppStatus ippsAnd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst,
int len);

IppStatus ippsAnd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst,
int len);

IppStatus ippsAnd_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);

IppStatus ippsAnd_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);

IppStatus ippsAnd_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);

```

### Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the two source vectors.
-----------------------------	-------------------------------------

<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector for the in-place operation.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsAnd` is declared in the `ipps.h` file. This function computes the bitwise AND of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsAnd` compute the bitwise AND of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## OrC

*Computes the bitwise OR of a scalar value and each element of a vector.*

---

### Syntax

```

IppStatus ippsOrC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);
IppStatus ippsOrC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);
IppStatus ippsOrC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);
IppStatus ippsOrC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);
IppStatus ippsOrC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);
IppStatus ippsOrC_32u_I(Ipp16u val, Ipp32u* pSrcDst, int len);

```

### Parameters

<i>val</i>	Input scalar value.
------------	---------------------

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsOrC` is declared in the `ipps.h` file. This function computes the bitwise OR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsOrC` compute the bitwise OR of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Or

Computes the bitwise OR of two vectors.

## Syntax

```
IppStatus ippsOr_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len);
```

```
IppStatus ippsOr_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len);
```

```
IppStatus ippsOr_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst, int len);
```

```
IppStatus ippsOr_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
```

```
IppStatus ippsOr_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
```

```
IppStatus ippsOr_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

## Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the two source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector for the in-place operation.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsOr` is declared in the `ipps.h` file. This function computes the bitwise OR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsOr` compute the bitwise OR of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

## Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## XorC

*Computes the bitwise XOR of a scalar value and each element of a vector.*

---

### Syntax

```

IppStatus ippsXorC_8u(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len);

IppStatus ippsXorC_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);

IppStatus ippsXorC_32u(const Ipp32u* pSrc, Ipp32u val, Ipp32u* pDst, int len);

IppStatus ippsXorC_8u_I(Ipp8u val, Ipp8u* pSrcDst, int len);

IppStatus ippsXorC_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);

```

```
IppStatus ippsXorC_32u_I(Ipp32u val, Ipp32u* pSrcDst, int len);
```

### Parameters

<i>val</i>	Input scalar value.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsXorC` is declared in the `ipps.h` file. This function computes the bitwise XOR of a scalar value *val* and each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsXorC` compute the bitwise XOR of a scalar value *val* and each element of the vector *pSrcDst* and store the result in *pSrcDst*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Xor

*Computes the bitwise XOR of two vectors.*

---

### Syntax

```
IppStatus ippsXor_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
int len);
```

```
IppStatus ippsXor_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst,
int len);
```

```
IppStatus ippsXor_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst,
int len);
```

```
IppStatus ippsXor_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
```

```
IppStatus ippsXor_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsXor_32u_I(const Ipp32u* pSrc, Ipp32u* pSrcDst, int len);
```

## Parameters

<i>pSrc1</i> , <i>pSrc2</i>	Pointers to the two source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector for the in-place operation.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsXor` is declared in the `ipps.h` file. This function computes the bitwise XOR of the corresponding elements of the vectors *pSrc1* and *pSrc2*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsXor` compute the bitwise XOR of the corresponding elements of the vectors *pSrc* and *pSrcDst* and store the result in the vector *pSrcDst*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Not

Computes the bitwise NOT of the vector elements.

## Syntax

```
IppStatus ippsNot_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsNot_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsNot_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsNot_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsNot_16u_I(Ipp16u* pSrcDst, int len);
```



```
IppStatus ippsNot_32u_I(Ipp32u* pSrcDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsNot` is declared in the `ipps.h` file. This function computes the bitwise NOT of the corresponding elements of the vectors *pSrc*, and stores the result in the vector *pDst*.

The in-place flavors of `ippsNot` compute the bitwise NOT of the corresponding elements of the vector *pSrcDst* and store the result in the vector *pSrcDst*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## LShiftC

Shifts bits in vector elements to the left.

### Syntax

```
IppStatus ippsLShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int len);
```

```
IppStatus ippsLShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst, int len);
```

```
IppStatus ippsLShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst, int len);
```

```
IppStatus ippsLShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst, int len);
```

```
IppStatus ippsLShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
```

```
IppStatus ippsLShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsLShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsLShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);
```

## Parameters

<i>val</i>	Number of bits by which the function shifts each element of the vector <i>pSrc</i> or <i>pSrcDst</i> .
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsLShiftC` is declared in the `ipps.h` file. This function shifts each element of the vector *pSrc* by *val* bits to the left, and stores the result in *pDst*.

The in-place flavors of `ippsLShiftC` shift each element of the vector *pSrcDst* by *val* bits to the left and store the result in *pSrcDst*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## RShiftC

*Shifts bits in vector elements to the right.*

---

### Syntax

```
IppStatus ippsRShiftC_8u(const Ipp8u* pSrc, int val, Ipp8u* pDst, int len);
IppStatus ippsRShiftC_16s(const Ipp16s* pSrc, int val, Ipp16s* pDst, int len);
```

```

IppStatus ippsRShiftC_16u(const Ipp16u* pSrc, int val, Ipp16u* pDst, int
len);

IppStatus ippsRShiftC_32s(const Ipp32s* pSrc, int val, Ipp32s* pDst, int
len);

IppStatus ippsRShiftC_8u_I(int val, Ipp8u* pSrcDst, int len);
IppStatus ippsRShiftC_16u_I(int val, Ipp16u* pSrcDst, int len);
IppStatus ippsRShiftC_16s_I(int val, Ipp16s* pSrcDst, int len);
IppStatus ippsRShiftC_32s_I(int val, Ipp32s* pSrcDst, int len);

```

## Parameters

<i>val</i>	Number of bits by which the function shifts each element of the vector <i>pSrc</i> or <i>pSrcDst</i> .
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsRShiftC` is declared in the `ipps.h` file. This function shifts each element of the vector *pSrc* by *val* bits to the right, and stores the result in *pDst*.

The in-place flavors of `ippsRShiftC` shift each element of the vector *pSrcDst* by *val* bits to the right and store the result in *pSrcDst*.

Note that the arithmetic shift is realized for signed data, and the logical shift for unsigned data.

Example 5-1 below shows how the logical and shift functions can be used in the saturate operation. The data are converted to the `unsigned char` range [0...255].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

**Example 5-1 Using the Logical and Shift Functions**

```
void saturate(void) {
    Ipp16s x[8] = {1000, -257, 127, 4, 5, 0, 7, 8}, lo[8], hi[8];
    IppStatus status = ippsNot_16u((Ipp16u*)x, (Ipp16u*)lo, 8);
    ippsRShiftC_16s_I(15, lo, 8);
    ippsCopy_16s(x, hi, 8);
    ippsSubCRev_16s_ISfs(255, hi, 8, 0);
    ippsRShiftC_16s_I(15, hi, 8);
    ippsAnd_16u_I((Ipp16u*)lo, (Ipp16u*)x, 8);
    ippsOr_16u_I((Ipp16u*)hi, (Ipp16u*)x, 8);
    ippsAndC_16u_I(255, (Ipp16u*)x, 8);
    printf_16s("saturate =", x, 8, status);
}
```

Output:

```
saturate = 255 0 127 4 5 0 7 8
```

## Arithmetic Functions

This section describes the Intel IPP signal processing functions that perform vector arithmetic operations on vectors. The arithmetic functions include basic element-wise arithmetic operations between vectors, as well as more complex calculations such as computing absolute values, square and square root, natural logarithm and exponential of vector elements.

Intel IPP software provides two versions of each function. One version performs the operation in-place, while the other stores the results of the operation in a different destination vector, that is, executes an out-of-place operation.

## AddC

Adds a constant value to each element of a vector.

### Syntax

#### Case 1. Not-in-place operations on floating point data.

```
IppStatus ippsAddC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
```

```
IppStatus ippsAddC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
```

```
IppStatus ippsAddC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
```

```
IppStatus ippsAddC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
```

#### Case 2. Not-in-place operations on integer data.

```
IppStatus ippsAddC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int scaleFactor);
```

```
IppStatus ippsAddC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsAddC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len, int scaleFactor);
```

```
IppStatus ippsAddC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsAddC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int len, int scaleFactor);
```

```
IppStatus ippsAddC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst, int len, int scaleFactor);
```

#### Case 3. In-place operations on floating point data.

```
IppStatus ippsAddC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
```

```
IppStatus ippsAddC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsAddC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsAddC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsAddC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

#### Case 4. In-place operations on integer data.

```
IppStatus ippsAddC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAddC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAddC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAddC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAddC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsAddC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
scaleFactor);
```

#### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>val</i>	Scalar value used to increment each element of the source vector
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

#### Description

The function `ippsAddC` is declared in the `ipps.h` file. This function adds a value *val* to each element of the source vector *pSrc*, and stores the result in the destination vector *pDst*.

The in-place flavors of `ippsAddC` add a value *val* to each element of the vector *pSrcDst*, and store the result in *pSrcDst*.

Functions with *sfs* suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Add

*Adds the elements of two vectors.*

---

### Syntax

**Case 1. Not-in-place operations on floating point data, and integer data without scaling.**

```

IppStatus ippsAdd_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
int len);

IppStatus ippsAdd_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst,
int len);

IppStatus ippsAdd_32u(const Ipp32u* pSrc1, const Ipp32u* pSrc2, Ipp32u* pDst,
int len);

IppStatus ippsAdd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
int len);

IppStatus ippsAdd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
int len);

IppStatus ippsAdd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
pDst, int len);

IppStatus ippsAdd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
pDst, int len);

IppStatus ippsAdd_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp16u* pDst,
int len);

IppStatus ippsAdd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f*
pDst, int len);

```

**Case 2. Not-in-place operations on integer data with scaling.**

```

IppStatus ippsAdd_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
int len, int scaleFactor);

IppStatus ippsAdd_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
pDst, int len, int scaleFactor);

IppStatus ippsAdd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
pDst, int len, int scaleFactor);

IppStatus ippsAdd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
pDst, int len, int scaleFactor);

IppStatus ippsAdd_64s_Sfs(const Ipp64s* pSrc1, const Ipp64s* pSrc2, Ipp64s*
pDst, Ipp32u len, int scaleFactor);

IppStatus ippsAdd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
Ipp16sc* pDst, int len, int scaleFactor);

IppStatus ippsAdd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
Ipp32sc* pDst, int len, int scaleFactor);

```

**Case 3. In-place operations on floating point data, and integer data without scaling.**

```

IppStatus ippsAdd_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsAdd_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsAdd_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsAdd_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
IppStatus ippsAdd_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
IppStatus ippsAdd_16s32s_I(const Ipp16s* pSrc, Ipp32s* pSrcDst, int len);

```

**Case 4. In-place operations on integer data with scaling.**

```

IppStatus ippsAdd_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int
scaleFactor);

IppStatus ippsAdd_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
scaleFactor);

IppStatus ippsAdd_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
scaleFactor);

IppStatus ippsAdd_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
scaleFactor);

```



```
IppStatus ippsAdd_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len,
int scaleFactor);
```

```
IppStatus ippsAdd_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len,
int scaleFactor);
```

## Parameters

<i>pSrc1, pSrc2</i>	Pointers to the source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector for in-place operations.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsAdd` is declared in the `ipps.h` file. This function adds the elements of the vector *pSrc1* to the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of `ippsAdd` add the elements of the vector *pSrc* to the elements of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Example 5-2 below shows that overflow does not occur while adding big numbers due to the scaling operation.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 5-2 Using the ippsAdd Function

```
IppStatus add(void) {
    Ipp16s x[4] = {-1, 32767, 2, 30000};
    IppStatus st = ippsAdd_16s_ISfs(x, x, 4, 1);
    printf_16s("add =", x, 4, st);
    return st;
}
```

Output:  
add = -1 32767 2 30000

## AddProductC

*Adds product of a vector and a constant to the accumulator vector.*

---

### Syntax

```
IppStatus ippsAddProductC_32f(const Ipp32f* pSrc, const Ipp32f val, Ipp32f* pSrcDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>val</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsAddProductC` is declared in the `ipps.h` file. This function multiplies each element of the source vector *pSrc* by a value *val* and adds the result to the corresponding element of the accumulator vector *pSrcDst* as given by:

$$pSrcDst[n] = pSrcDst[n] + pSrc[n]*val, 0 \leq n < len$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

## AddProduct

Adds product of two vectors to the accumulator vector.

---

### Syntax

#### Case 1. Operations on floating point data.

```
IppStatus ippsAddProduct_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsAddProduct_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsAddProduct_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsAddProduct_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pSrcDst, int len);
```

#### Case 2. Operations on integer data with scaling.

```
IppStatus ippsAddProduct_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsAddProduct_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s* pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsAddProduct_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32s* pSrcDst, int len, int scaleFactor);
```

### Parameters

<i>pSrc1, pSrc2</i>	Pointers to the source vectors.
<i>pSrcDst</i>	Pointer to the destination accumulator vector.
<i>len</i>	The number of elements in the vectors.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsAddProduct` is declared in the `ipps.h` file. This function multiplies each element of the source vector *pSrc1* by the corresponding element of the vector *pSrc2*, and adds the result to the corresponding element of the accumulator vector *pSrcDst* as given by:

$pSrcDst[n] = pSrcDst[n] + pSrc1[n] * pSrc2[n], 0 \leq n < len.$

Functions with *Sfs* suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## MulC

*Multiplies each elements of a vector by a constant value.*

---

### Syntax

#### Case 1. Not-in-place operations without scaling.

```
IppStatus ippMulC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
```

```
IppStatus ippMulC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
```

```
IppStatus ippMulC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
```

```
IppStatus ippMulC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
```

```
IppStatus ippMulC_Low_32f16s(const Ipp32f* pSrc, Ipp32f val, Ipp16s* pDst, int len);
```

#### Case 2. Not-in-place operations with scaling.

```
IppStatus ippMulC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int scaleFactor);
```

```
IppStatus ippMulC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippMulC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len, int scaleFactor);
```

```
IppStatus ippsMulC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int
len, int scaleFactor);
```

```
IppStatus ippsMulC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst,
int len, int scaleFactor);
```

```
IppStatus ippsMulC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst,
int len, int scaleFactor);
```

```
IppStatus ippsMulC_32f16s_Sfs(const Ipp32f* pSrc, Ipp32f val, Ipp16s* pDst,
int len, int scaleFactor);
```

### Case 3. In-place operations without scaling.

```
IppStatus ippsMulC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
```

```
IppStatus ippsMulC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsMulC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsMulC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsMulC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

### Case 4. In-place operations with scaling.

```
IppStatus ippsMulC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsMulC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsMulC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsMulC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsMulC_64s_ISfs(Ipp64s val, Ipp64s* pSrcDst, Ipp32f len, int
scaleFactor);
```

```
IppStatus ippsMulC_64f64s_ISfs(Ipp64f val, Ipp64s* pSrcDst, Ipp32f len, int
scaleFactor);
```

```
IppStatus ippsMulC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsMulC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
scaleFactor);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>val</i>	The scalar value used to multiply each element of the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operation.
<i>len</i>	The number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsMulC` is declared in the `ipps.h` file. This function multiplies each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsMulC` multiply each element of the vector *pSrcDst* by a value *val* and store the result in *pSrcDst*.

Function flavor with `Low` suffix in its name requires that each value of the product *pSrc\*val* does not exceed the `Ipp32s` data type range.

Function flavors with `Sfs` suffix perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Mul

*Multiplies the elements of two vectors.*

---

### Syntax

**Case 1. Not-in-place operations on floating point data, and integer data without scaling.**

```
IppStatus ippsMul_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
int len);
```

```

IppStatus ippsMul_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
int len);

IppStatus ippsMul_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
int len);

IppStatus ippsMul_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
pDst, int len);

IppStatus ippsMul_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
pDst, int len);

IppStatus ippsMul_8u16u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp16u* pDst,
int len);

IppStatus ippsMul_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f*
pDst, int len);

IppStatus ippsMul_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
pDst, int len);

```

### Case 2. Not-in-place operations on integer data with scaling.

```

IppStatus ippsMul_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
int len, int scaleFactor);

IppStatus ippsMul_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
pDst, int len, int scaleFactor);

IppStatus ippsMul_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
pDst, int len, int scaleFactor);

IppStatus ippsMul_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
pDst, int len, int scaleFactor);

IppStatus ippsMul_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
Ipp16sc* pDst, int len, int scaleFactor);

IppStatus ippsMul_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
Ipp32sc* pDst, int len, int scaleFactor);

IppStatus ippsMul_16u16s_Sfs(const Ipp16u* pSrc1, const Ipp16s* pSrc2, Ipp16s*
pDst, int len, int scaleFactor);

IppStatus ippsMul_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32s*
pDst, int len, int scaleFactor);

```

```
IppStatus ippsMul_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc* pSrc2,
Ipp32sc* pDst, int len, int scaleFactor);
```

```
IppStatus ippsMul_Low_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2,
Ipp32s* pDst, int len, int scaleFactor);
```

### **Case 3. In-place operations on floating point data and integer data without scaling.**

```
IppStatus ippsMul_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
```

```
IppStatus ippsMul_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsMul_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsMul_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsMul_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
```

```
IppStatus ippsMul_32f32fc_I(const Ipp32f* pSrc, Ipp32fc* pSrcDst, int len);
```

### **Case 4. In-place operations on integer data with scaling.**

```
IppStatus ippsMul_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus, ippsMul_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len,
int scaleFactor);
```

```
IppStatus ippsMul_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsMul_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsMul_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len,
int scaleFactor);
```

```
IppStatus ippsMul_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len,
int scaleFactor);
```

```
IppStatus ippsMul_32s32sc_ISfs(const Ipp32s* pSrc, Ipp32sc* pSrcDst, int
len, int scaleFactor);
```

## **Parameters**

<i>pSrc1, pSrc2</i>	Pointers to the source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector for in-place operation.



<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsMul` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2* and stores the result in *pDst*.

The in-place flavors of `ippsMul` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst* and store the result in *pSrcDst*.

Function flavors with `Sfs` suffix perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

Function flavor with `Low` suffix in its name requires that each value of the product does not exceed the `Ipp32s` data type range.

## Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0

## SubC

*Subtracts a constant value from each element of a vector.*

---

### Syntax

#### Case 1. Not-in-place operations on floating point data.

```
IppStatus ippsSubC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
```

```
IppStatus ippsSubC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
```

```
IppStatus ippsSubC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
```

```
IppStatus ippsSubC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
```

### **Case 2. Not-in-place operations on integer data.**

```
IppStatus ippsSubC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst, int len, int scaleFactor);
```

### **Case 3. In-place operations on floating point data.**

```
IppStatus ippsSubC_16s_I(Ipp16s val, Ipp16s* pSrcDst, int len);
```

```
IppStatus ippsSubC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsSubC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsSubC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsSubC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

### **Case 4. In-place operations on integer data.**

```
IppStatus ippsSubC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsSubC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSubC_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
scaleFactor);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>val</i>	Scalar value used to decrement each element of the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsSubC` is declared in the `ipps.h` file. This function subtracts a value *val* from each element of the vector *pSrc*, and stores the result in *pDst*.

The in-place flavors of `ippsSubC` subtract a value *val* from each element of the vector *pSrcDst* and store the result in *pSrcDst*.

Functions with *sfs* suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## SubCRev

*Subtracts each element of a vector from a constant value.*

---

### Syntax

#### Case 1. Not-in-place operations on floating point data.

```
IppStatus ippsSubCRev_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
```

```
IppStatus ippsSubCRev_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
```

```
IppStatus ippsSubCRev_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
```

```
IppStatus ippsSubCRev_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
```

#### Case 2. Not-in-place operations on integer data.

```
IppStatus ippsSubCRev_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubCRev_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubCRev_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubCRev_32s_Sfs(const Ipp32s* pSrc, Ipp32s val, Ipp32s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubCRev_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSubCRev_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc val, Ipp32sc* pDst, int len, int scaleFactor);
```

#### Case 3. In-place operations on floating point data.

```
IppStatus ippsSubCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsSubCRev_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsSubCRev_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsSubCRev_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

#### Case 4. In-place operations on integer data.

```
IppStatus ippsSubCRev_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSubCRev_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSubCRev_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSubCRev_32s_ISfs(Ipp32s val, Ipp32s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSubCRev_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSubCRev_32sc_ISfs(Ipp32sc val, Ipp32sc* pSrcDst, int len, int
scaleFactor);
```

### Parameters

<i>val</i>	Scalar value from which vector elements are subtracted.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the vector whose elements are to be subtracted from the value <i>val</i> in case of the in-place operation. The destination vector which stores the result of the subtraction $val - pSrcDst[n]$ .
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsSubCRev` is declared in the `ipps.h` file. This function subtracts each element of the vector *pSrc* from a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsSubCRev` subtract each element of the vector *pSrcDst* from a value *val* and store the result in *pSrcDst*.

Functions with *Sfs* suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Sub

*Subtracts the elements of two vectors.*

---

### Syntax

#### Case 1. Not-in-place operations on floating point data, and integer data without scaling.

```
IppStatus ippsSub_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst,
int len);
```

```
IppStatus ippsSub_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
int len);
```

```
IppStatus ippsSub_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
int len);
```

```
IppStatus ippsSub_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
pDst, int len);
```

```
IppStatus ippsSub_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
pDst, int len);
```

```
IppStatus ippsSub_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp32f*
pDst, int len);
```

#### Case 2. Not-in-place operations on integer data with scaling.

```
IppStatus ippsSub_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
int len, int scaleFactor);
```

```
IppStatus ippsSub_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u*
pDst, int len, int scaleFactor);
```

```
IppStatus ippsSub_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
pDst, int len, int scaleFactor);
```

```
IppStatus ippsSub_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
pDst, int len, int scaleFactor);
```

```
IppStatus ippsSub_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
Ipp16sc* pDst, int len, int scaleFactor);
```

```
IppStatus ippsSub_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
Ipp32sc* pDst, int len, int scaleFactor);
```

### Case 3. In-place operations on floating point data and integer data without scaling.

```
IppStatus ippsSub_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
```

```
IppStatus ippsSub_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsSub_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsSub_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsSub_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
```

### Case 4. In-place operations on integer data with scaling.

```
IppStatus ippsSub_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSub_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSub_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSub_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsSub_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len,
int scaleFactor);
```

```
IppStatus ippsSub_32sc_ISfs(const Ipp32sc* pSrc, Ipp32sc* pSrcDst, int len,
int scaleFactor);
```

## Parameters

*pSrc1*                      Pointer to the source vector-subtrahend, whose elements are to be subtracted.

<i>pSrc2</i>	Pointer to the source vector-minuend from whose elements the elements of <i>pSrc1</i> are to be subtracted.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector-subtrahend for in-place operation.
<i>pSrcDst</i>	Pointer to the source vector-minuend and destination vector for in-place operation.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsSub` is declared in the `ipps.h` file. This function subtracts the elements of the vector *pSrc1* from the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of `ippsSub` subtract the elements of the vector *pSrc* from the elements of a vector *pSrcDst* and store the result in *pSrcDst*.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## DivC

*Divides each element of a vector by a constant value.*

---

### Syntax

#### Case 1. Not-in-place operations on floating point data.

```
IppStatus ippsDivC_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);
```

```
IppStatus ippsDivC_64f(const Ipp64f* pSrc, Ipp64f val, Ipp64f* pDst, int len);
```



```
IppStatus ippsDivC_32fc(const Ipp32fc* pSrc, Ipp32fc val, Ipp32fc* pDst, int len);
```

```
IppStatus ippsDivC_64fc(const Ipp64fc* pSrc, Ipp64fc val, Ipp64fc* pDst, int len);
```

### Case 2. Not-in-place operations on integer data with scaling.

```
IppStatus ippsDivC_8u_Sfs(const Ipp8u* pSrc, Ipp8u val, Ipp8u* pDst, int len, int ScaleFactor);
```

```
IppStatus ippsDivC_16u_Sfs(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len, int scaleFactor);
```

```
IppStatus ippsDivC_16s_Sfs(const Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len, int ScaleFactor);
```

```
IppStatus ippsDivC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc val, Ipp16sc* pDst, int len, int ScaleFactor);
```

### Case 3. In-place operations on floating point data.

```
IppStatus ippsDivC_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsDivC_64f_I(Ipp64f val, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsDivC_32fc_I(Ipp32fc val, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsDivC_64fc_I(Ipp64fc val, Ipp64fc* pSrcDst, int len);
```

### Case 4. In-place operations on integer data with scaling.

```
IppStatus ippsDivC_8u_ISfs(Ipp8u val, Ipp8u* pSrcDst, int len, int ScaleFactor);
```

```
IppStatus ippsDivC_16u_ISfs(Ipp16u val, Ipp16u* pSrcDst, int len, int scaleFactor);
```

```
IppStatus ippsDivC_16s_ISfs(Ipp16s val, Ipp16s* pSrcDst, int len, int ScaleFactor);
```

```
IppStatus ippsDivC_64s_ISfs(Ipp64s val, Ipp64s* pSrcDst, Ipp32u len, int ScaleFactor);
```

```
IppStatus ippsDivC_16sc_ISfs(Ipp16sc val, Ipp16sc* pSrcDst, int len, int ScaleFactor);
```

## Parameters

*val*                      Scalar value used as a divisor.

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsDivC` is declared in the `ipps.h` file. This function divides each element of the vector *pSrc* by a value *val* and stores the result in *pDst*.

The in-place flavors of `ippsDivC` divide each element of the vector *pSrcDst* by a value *val* and store the result in *pSrcDst*.

Functions with *Sfs* suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when <i>val</i> is equal to 0.

## DivCRev

*Divides a constant value by each element of a vector.*

---

### Syntax

```

IppStatus ippsDivCRev_16u(const Ipp16u* pSrc, Ipp16u val, Ipp16u* pDst, int len);

IppStatus ippsDivCRev_32f(const Ipp32f* pSrc, Ipp32f val, Ipp32f* pDst, int len);

IppStatus ippsDivCRev_16u_I(Ipp16u val, Ipp16u* pSrcDst, int len);

IppStatus ippsDivCRev_32f_I(Ipp32f val, Ipp32f* pSrcDst, int len);

```

## Parameters

<i>val</i>	Constant value used as a dividend in the operation.
<i>pSrc</i>	Pointer to the source vector whose elements are used as divisors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operation.
<i>len</i>	Number of elements in the vector

## Description

The function `ippsDivCRev` is declared in the `ipps.h` file. This function divides the constant value *val* by each element of the vector *pSrc* and stores the results in *pDst*.

The in-place flavors of `ippsDivC` divide the constant value *val* by each element of the vector *pSrcDst* and store the results in *pSrcDst*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when <i>val</i> is equal to 0.

## Div

*Divides the elements of two vectors.*

---

## Syntax

### Case 1. Not-in-place operations on integer data.

```
IpplStatus ippsDiv_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst,
int len, int scaleFactor);
```

```
IpplStatus ippsDiv_16u_Sfs(const Ippl6u* pSrc1, const Ippl6u* pSrc2, Ippl6u*
pDst, int len, int scaleFactor);
```

```
IppStatus ippsDiv_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s*
pDst, int len, int scaleFactor);
```

```
IppStatus ippsDiv_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, Ipp32s*
pDst, int len, int scaleFactor);
```

```
IppStatus ippsDiv_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
Ipp16sc* pDst, int len, int scaleFactor);
```

```
IppStatus ippsDiv_32s16s_Sfs(const Ipp16s* pSrc1, const Ipp32s* pSrc2, Ipp16s*
pDst, int len, int scaleFactor);
```

### **Case 2. Not-in-place operations on floating point data.**

```
IppStatus ippsDiv_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst,
int len);
```

```
IppStatus ippsDiv_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst,
int len);
```

```
IppStatus ippsDiv_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc*
pDst, int len);
```

```
IppStatus ippsDiv_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc*
pDst, int len);
```

### **Case 3. In-place operations on integer data.**

```
IppStatus ippsDiv_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, int
ScaleFactor);
```

```
IppStatus ippsDiv_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, int
scaleFactor);
```

```
IppStatus ippsDiv_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int
ScaleFactor);
```

```
IppStatus ippsDiv_16sc_ISfs(const Ipp16sc* pSrc, Ipp16sc* pSrcDst, int len,
int ScaleFactor);
```

```
IppStatus ippsDiv_32s_ISfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len, int
ScaleFactor);
```

### **Case 4. In-place operations on floating point data**

```
IppStatus ippsDiv_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsDiv_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsDiv_32fc_I(const Ipp32fc* pSrc, Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippDiv_64fc_I(const Ipp64fc* pSrc, Ipp64fc* pSrcDst, int len);
```

## Parameters

<i>pSrc1</i>	Pointer to the vector whose elements are used as divisors.
<i>pSrc2</i>	Pointer to the vector whose elements are used as dividends.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector whose elements are used as divisors for in-place operations.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operations.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippDiv` is declared in the `ipps.h` file. This function divides the elements of the vector *pSrc2* by the elements of the vector *pSrc1*, and stores the result in *pDst*.

The in-place flavors of `ippDiv` divide the elements of the vector *pSrcDst* by the elements of the vector *pSrc* and store the result in *pSrcDst*.

Functions with *Sfs* suffixes perform scaling of the result value in accordance with the *scaleFactor* value. If the output value exceeds the data range, the result becomes saturated.

If the function `ippDiv` encounters a zero-valued divisor vector element, it returns a warning status and continues execution with the corresponding result value (see appendix A "[Handling of Special Cases](#)" for more information).

Example 5-3 shows that the use of the scaling factor in the integer functions increases operation accuracy. Example 5-4 considers division by zero exceptions ( $x / 0$ ,  $0 / 0$ ).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning for zero-valued divisor vector element.

**Example 5-3 Using the ippsDiv\_16s\_ISfs Function**

```
IppStatus div16s( void ) {
    Ipp16s x[4] = { -3, 2, 0, 300 };
    Ipp16s y[4] = { -2, 2, 0, 0 };
    IppStatus st = ippsDiv_16s_ISfs( y, x, 4, -1 );
    printf_16s("div16s =", x, 4, st );
    return st;
}
```

Output:

```
-- warning 6, Zero value(s) in the divisor of the function Div
div16s =  3 2 0 32767
```

**Example 5-4 Using the ippsDiv\_32f\_I Function**

```
IppStatus div32f( void ) {
    Ipp32f x[4] = { -3, 2, 0, 300 };
    Ipp32f y[4] = { -2, 2, 0, 0 };
    IppStatus st = ippsDiv_32f_I( y, x, 4 );
    printf_32f( "div32f =", x, 4, st );
    return st;
}
```

Output:

```
-- warning 6, Zero value(s) in the divisor of the function Div
div32f =  1.500000 1.000000 1.#IND00 1.#INF00
```

## Div\_Round

Divides the elements of two vectors with rounding.

### Syntax

#### Case 1. Not-in-place operations on integer data.

```
IppStatus ippsDiv_Round_8u_Sfs(const Ipp8u* pSrc1, const Ipp8u* pSrc2, Ipp8u* pDst, int len, IppRoundMode rndMode, int scaleFactor);
```

```
IppStatus ippsDiv_Round_16u_Sfs(const Ipp16u* pSrc1, const Ipp16u* pSrc2, Ipp16u* pDst, int len, IppRoundMode rndMode, int scaleFactor);
```

```
IppStatus ippsDiv_Round_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int len, IppRoundMode rndMode, int scaleFactor);
```

#### Case 2. In-place operations on integer data.

```
IppStatus ippsDiv_Round_8u_ISfs(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len, IppRoundMode rndMode, int scaleFactor);
```

```
IppStatus ippsDiv_Round_16u_ISfs(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len, IppRoundMode rndMode, int scaleFactor);
```

```
IppStatus ippsDiv_Round_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, IppRoundMode rndMode, int scaleFactor);
```

### Parameters

<i>pSrc1</i>	Pointer to the vector whose elements are used as divisors.
<i>pSrc2</i>	Pointer to the vector whose elements are used as dividends.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the source vector whose elements are used as divisors for in-place operations.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operations.
<i>len</i>	Number of elements in the vector
<i>roundMode</i>	Rounding mode, the following values are possible: <i>ippRndZero</i> Specifies that floating-point values must be truncated toward zero.

`ippRndNear` Specifies that floating-point values must be rounded to the nearest even integer.

`ippRndFinancial` Specifies that floating-point values must be rounded down to the nearest integer if decimal value is less than 0.5, or rounded up to the nearest integer if decimal value is equal or greater than 0.5.

`scaleFactor` Scale factor, refer to [Integer Scaling](#).

## Description

The function `ippsDiv_Round` is declared in the `ipps.h` file. This function divides the elements of the vector `pSrc2` by the elements of the vector `pSrc1`, the result is rounded using the rounding method specified by the parameter `roundMode` and stored in the vector `pDst`.

The in-place flavors of `ippsDiv_Round` divide the elements of the vector `pSrcDst` by the elements of the vector `pSrc`, the result is rounded using the rounding method specified by the parameter `roundMode` and stored in the vector `pSrcDst`.

Functions perform scaling of the result value in accordance with the `scaleFactor` value. If the output value exceeds the data range, the result becomes saturated.

If the function `ippsDiv_Round` encounters a zero-valued divisor vector element, it returns a warning status and continues execution with the corresponding result value (see appendix A ["Handling of Special Cases"](#) for more information).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning for zero-valued divisor vector element. The function execution is continued.
<code>ippStsRndModNotSupported</code>	Indicates an error condition if the <code>roundMode</code> has an illegal value.



## Abs

Computes absolute values of vector elements.

### Syntax

```
IppStatus ippsAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsAbs_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsAbs_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsAbs_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsAbs_64f_I(Ipp64f* pSrcDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operations.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsAbs` is declared in the `ipps.h` file. This function computes the absolute values of each element of the vector *pSrc* and stores the result in *pDst*. The in-place flavors of `ippsAbs` compute the absolute values of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

To compute the absolute values of complex data, use the function `ippsMagnitude`[ippsMagnitude](#).

Example 5-5 below shows how to call the function `ippsAbs_32f_I`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when the `pSrc`, `pDst`, or `pSrcDst` pointer is NULL.

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

## Example 5-5 Using the `ippsAbs` Function

```
void abs32f(void) {
    Ipp32f x[4] = {-1, 1, 0, 0};
    x[3] *= (-1);
    ippsAbs_32f_I(x, 4);
    printf_32f("abs =", x, 4, ippStsNoErr);
}
```

Output:

```
abs = 1.000000 1.000000 0.000000 0.000000
```

## Sqr

*Computes a square of each element of a vector.*

---

### Syntax

```
IppStatus ippsSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqr_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqr_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqr_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqr_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqr_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqr_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len, int
scaleFactor);
IppStatus ippsSqr_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
scaleFactor);
```

```

IppStatus ippsSqr_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len, int
scaleFactor);

IppStatus ippsSqr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, int
scaleFactor);

IppStatus ippsSqr_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqr_16sc_ISfs(Ipp16sc* pSrcDst, int len, int scaleFactor);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operations.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsSqr` is declared in the `ipps.h` file. This function computes the square of each element of the vector *pSrc*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = pSrc[n]^2$$

The in-place flavors of `ippsSqr` compute the square of each element of the vector *pSrcDst* and store the result in *pSrcDst*. The computation is performed as follows:

$$pSrcDst[n] = pSrcDst[n]^2$$

When computing the square of an integer number, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. Example 5-6 below shows how to get the value of  $200^2$ . Without scaling this result is clipped to 32767. Scaling retains the output data range but results in the precision loss in low-order bits.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when the `pSrc`, `pDst`, or `pSrcDst` pointer is NULL.

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

## Example 5-6 Using the `ippsSqr` Function

```
IppStatus sqr(void) {
    Ipp16s x[4] = {-3, 2, 30, 200};
    IppStatus st = ippsSqr_16s_ISfs(x, 4, 1);
    printf_16s("sqr =", x, 4, st);
    return st;
}
```

Output:

```
sqr = 4 2 450 20000
```

## Sqrt

*Computes a square root of each element of a vector.*

---

### Syntax

```
IppStatus ippsSqrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSqrt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSqrt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsSqrt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsSqrt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSqrt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSqrt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsSqrt_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsSqrt_8u_Sfs(const Ipp8u* pSrc, Ipp8u* pDst, int len, int
scaleFactor);
```

```

IppStatus ippsSqrt_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
scaleFactor);

IppStatus ippsSqrt_16u_Sfs(const Ipp16u* pSrc, Ipp16u* pDst, int len, int
scaleFactor);

IppStatus ippsSqrt_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len, int
scaleFactor);

IppStatus ippsSqrt_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len, int
scaleFactor);

IppStatus ippsSqrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int
scaleFactor);

IppStatus ippsSqrt_64s16s_Sfs(const Ipp64s* pSrc, Ipp16s* pDst, int len, int
scaleFactor);

IppStatus ippsSqrt_8u_ISfs(Ipp8u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16u_ISfs(Ipp16u* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_16sc_ISfs(Ipp16sc* pSrcDst, int len, int scaleFactor);
IppStatus ippsSqrt_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operations.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsSqrt` is declared in the `ipps.h` file. This function computes the square root of each element of the vector *pSrc*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = (pSrc[n])^{1/2}$$

The in-place flavors of `ippsSqrt` compute the square root of each element of the vector `pSrcDst` and store the result in `pSrcDst`. The computation is performed as follows:

$$pSrcDst[n] = (pSrcDst[n])^{1/2}.$$

The square root of complex vector elements is computed as follows:

$$\sqrt{a + j \cdot b} = \sqrt{\frac{\sqrt{a^2 + b^2} + a}{2}} + j \cdot \text{sign}(b) \cdot \sqrt{\frac{\sqrt{a^2 + b^2} - a}{2}}$$

If the function `ippsSqrt` encounters a negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A "[Handling of Special Cases](#)" for more information).

To increase precision of an integer output, use the scale factor.

Example 5-7 below shows how to call the function `ippsSqrt_16s_ISfs`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsSqrtNegArg</code>	Indicates a warning that a source element has a negative value.

### Example 5-7 Using the `ippsSqrt` Function

```
IppStatus sqrt(void) {
    Ipp16s x[4] = {-3, 2, 30, 300};
    IppStatus st = ippsSqrt_16s_ISfs(x, 4, -1);
    printf_16s("sqrt =", x, 4, st);
    return st;
}
```

Output:

```
-- warning 3, Negative value(s) in the argument of the function Sqrt
sqrt = 0 3 11 35
```

## Cubrt

Computes cube root of each element of a vector.

### Syntax

```
IppStatus ippsCubrt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsCubrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
int scaleFactor);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsCubrt` is declared in the `ipps.h` file. This function computes cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

$$pDst[n] = (pSrc[n])^{1/3}, 0 \leq n < len.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pSrc</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Exp

*Computes  $e$  to the power of each element of a vector.*

---

### Syntax

```

IppStatus ippExp_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippExp_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippExp_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippExp_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippExp_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippExp_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
scaleFactor);
IppStatus ippExp_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int
scaleFactor);
IppStatus ippExp_64s_Sfs(const Ipp64s* pSrc, Ipp64s* pDst, int len, int
scaleFactor);
IppStatus ippExp_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);
IppStatus ippExp_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
IppStatus ippExp_64s_ISfs(Ipp64s* pSrcDst, int len, int scaleFactor);

```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>pSrcDst</code>	Pointer to the source and destination vector <code>pSrcDst</code> for the in-place operation.



*len*                      Number of elements in the vector  
*scaleFactor*            Scale factor, refer to [Integer Scaling](#).

## Description

The function `ippsExp` is declared in the `ipps.h` file. This function computes the exponential function of each element of the vector *pSrc*, and stores the result in *pDst*.

The computation is performed as follows:

$$pDst[n] = e^{pSrc[n]}$$

The in-place flavors of `ippsExp` compute the exponential function of each element of the vector *pSrcDst* and store the result in *pSrcDst*.

The computation is performed as follows:

$$pSrcDst[n] = e^{pSrcDst[n]}$$

When an overflow occurs, the function continues operation with the corresponding result value (see appendix A ["Handling of Special Cases"](#) for more information).

When computing the exponent of an integer number, the output result can exceed the data range and become saturated. The scaling retains the output data range but results in precision loss in low-order bits. The function `ippsExp_32f64f` computes the output result in a higher precision data range.

Examples 5-8 and 5-9 below show how to call the functions `ippsExp_16s_ISfs` and `ippsExp_64f_I` respectively.

## Application Notes

For the functions `ippsExp` and `ippsLn` the result is rounded to the nearest integer after scaling.

## Return Values

`ippStsNoErr`            Indicates no error.  
`ippStsNullPtrErr`      Indicates an error when the *pSrc*, *pDst*, or *pSrcDst* pointer is NULL.  
`ippStsSizeErr`        Indicates an error when *len* is less than or equal to 0.

## Example 5-8 Using the `ippsExp_16s_ISfs` Function

```
IppStatus exp16s(void) {
    Ipp16s x[4] = {-1, 2, 30, 0};
    IppStatus st = ippsExp_16s_ISfs(x, 4, -1);
    printf_16s("exp16s =", x, 4, st);
    return st;
}
```

Output:

```
exp16s =  1 15 32767 2
```

## Example 5-9 Using the `ippsExp_64f_I` Function

```
IppStatus exp64f(void) {
    Ipp64f x[4] = {-1, 2, 1, log(1.234567)};
    IppStatus st = ippsExp_64f_I(x, 4);
    printf_64f("exp64f =", x, 4, st);
    return st;
}
```

Output:

```
exp64f =  0.367879 7.389056 2.718282 1.234567
```

## Ln

*Computes the natural logarithm of each element of a vector.*

---

### Syntax

```
IppStatus ippsLn_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsLn_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsLn_32f_I(Ipp32f* pSrcDst, int len);
```

```

IppStatus ippsLn_64f_I(Ipp64f* pSrcDst, int len);

IppStatus ippsLn_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int
scaleFactor);

IppStatus ippsLn_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int
scaleFactor);

IppStatus ippsLn_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len, int
scaleFactor);

IppStatus ippsLn_16s_ISfs(Ipp16s* pSrcDst, int len, int scaleFactor);

IppStatus ippsLn_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsLn` is declared in the `ipps.h` file. This function computes the natural logarithm of each element of the vector *pSrc* and stores the result in *pDst* as given by

$$pDst[n] = \log_e (pSrc[n])$$

The in-place flavors of `ippsLn` compute the natural logarithm of each element of the vector *pSrcDst* and store the result in *pSrcDst* as given by

$$pSrcDst[n] = \log_e (pSrcDst[n])$$

If the function `ippsLn` encounters a zero or negative value in the input, it returns a warning status and continues execution with the corresponding result value (see appendix A "[Handling of Special Cases](#)" for more information).

Example 5-10 below shows how to call the function `ippsLn_32f_I`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements.
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements.

## Example 5-10 Using the `ippLn` Function

```
IppStatus ln32f(void) {
    Ipp32f x[4] = {-1, (float)IPP_E, 0, (float)(exp(1.234567))};
    IppStatus st = ippLn_32f_I(x, 4);
    printf_32f("Ln =", x, 4, st);
    return st;
}
```

Output:

```
-- warning 8, Negative value(s) of argument in the Ln function
```

```
Ln = -1.#IND00 1.000000 -1.#INF00 1.234567
```

## 10Log10

*Computes the decimal logarithm of each element of a vector and multiplies it by 10.*

---

### Syntax

```
IppStatus ipps10Log10_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len, int
scaleFactor);
```

```
IppStatus ipps10Log10_32s_ISfs(Ipp32s* pSrcDst, int len, int scaleFactor);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>pSrcDst</code>	Pointer to the source and destination vector for the in-place operation.

*len*                      Number of elements in the vector  
*scaleFactor*            Refer to Integer Scaling in Chapter 2.

### Description

The function `ipps10Log10` is declared in the `ipps.h` file. This function computes the decimal logarithm of each element of the vector *pSrc*, multiplies it by 10, and stores the result in *pDst* as given by

$$pDst[n] = 10 * \log_{10}(pSrc[n]).$$

The in-place flavor of **`ipps10Log10`** computes the decimal logarithm of each element of the vector *pSrcDst*, multiplies it by 10, and stores the result in *pSrcDst* as given by

$$pSrcDst[n] = 10 * \log_{10}(pSrcDst[n]).$$

If the function `ipps10Log10` encounters a zero or negative value in the input, it returns a warning status and continues execution with the corresponding result value (see Appendix A ["Handling of Special Cases"](#) for more information).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements.
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements.

## SumLn

*Sums natural logarithms of each element of a vector.*

---

### Syntax

```

IppStatus ippsSumLn_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum);
IppStatus ippsSumLn_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_32f64f(const Ipp32f* pSrc, int len, Ipp64f* pSum);
IppStatus ippsSumLn_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pSum);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSum</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsSumLn` is declared in the `ipps.h` file. This function computes the sum of natural logarithms of each element of the vector *pSrc* and stores the result value in *pSum*. The summation is given by:

$$sum = \sum_{n=0}^{len-1} \ln(pSrc[n])$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pSum</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>-Inf</code> .
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>NaN</code> .

## Arctan

*Computes the inverse tangent of each element of a vector.*

---

### Syntax

```
IppStatus ippsArctan_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsArctan_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsArctan_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsArctan_64f_I(Ipp64f* pSrcDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector <i>pSrcDst</i> for the in-place operation.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsAtan` is declared in the `ipps.h` file. This function computes the inverse tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

The computation is performed as follows:

$$pDst[n] = \arctan(pSrc[n]), 0 \leq n < len.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Normalize

*Normalizes elements of a real or complex vector using offset and division operations.*

---

### Syntax

```
IppStatus ippsNormalize_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
vSub, Ipp32f vDiv);
IppStatus ippsNormalize_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f
vSub, Ipp64f vDiv);
```

```

IppStatus ippsNormalize_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
Ipp32fc vSub, Ipp32f vDiv);

IppStatus ippsNormalize_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
Ipp64fc vSub, Ipp64f vDiv);

IppStatus ippsNormalize_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s vSub, int vDiv, int scaleFactor);

IppStatus ippsNormalize_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
len, Ipp16sc vSub, int vDiv, int scaleFactor);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>vSub</i>	Subtrahend value.
<i>vDiv</i>	Denominator value.
<i>pDst</i>	Pointer to the vector which stores the normalized elements.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsNormalize` is declared in the `ipps.h` file. This function subtracts *vSub* from elements of the input vector *pSrc*, divides the differences by *vDiv*, and stores the result in *pDst*. The computation is performed as follows:

$$pDst[n] = (pSrc[n] - vSub) / vDiv.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when <i>vDiv</i> is equal to 0 or less than the minimum floating-point positive number.



## Cauchy, CauchyD, CauchyDD2

*Computes Cauchy robust error function and its first and second derivatives*

---

### Syntax

```
IppStatus ippsCauchy_32f_I(Ipp32f* pSrcDst, int len, Ipp32f param);
IppStatus ippsCauchyD_32f_I(Ipp32f* pSrcDst, int len, Ipp32f param);
IppStatus ippsCauchyDD2_32f_I(Ipp32f* pSrcDst, Ipp32f* pD2FVal, int len,
Ipp32f param);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>len</i>	Number of elements in the vector
<i>pD2FVal</i>	Pointer to the array of the values of the second derivatives.
<i>param</i>	Parameter $C$ of the Cauchy function.

### Description

These functions are declared in the `ipps.h` file. For *len* elements of the source vector *pSrcDst* the function perform the following operations:

the function `ippsCauchy` calculates the Cauchy function and stores the results in the vector *pSrcDst*;

the function `ippsCauchyD` calculates first derivatives of the Cauchy function and stores the results in the vector *pSrcDst*;

the function `ippsCauchyDD2` calculates first and second derivatives of the Cauchy function and stores the values of first derivatives in the vector *pSrcDst*, values of second derivatives - in the array *pD2FVal*.

The following pseudo-codes show how these operations are performed.

Calculating the Cauchy function (`ippsCauchy`):

```
for( int i = 0; i < len; i++ ) pSrcDst[i] =  $\phi_C$ (pSrcDst[i]);
```

Calculating first derivatives of the Cauchy function (`ippsCauchyD`):

```
for( int i = 0; i < len; i++ ) pSrcDst[i] =  $\psi_C$ (pSrcDst[i]);
```

Calculating first and second derivatives of the Cauchy function (`ippsCauchyDD2`):

```
for( int i = 0; i < len; i++ ){
    pD2FVal[i] =  $\psi'_C$ (pSrcDst[i]);
    pSrcDst[i] =  $\psi_C$ (pSrcDst[i]);
}
```

Here

$$\begin{aligned}\varphi_C(x) &= \frac{1}{2} \log \left( 1 + \left( \frac{x}{C} \right)^2 \right) \\ \psi_C(x) &= \varphi'_C(x) = \frac{x}{C^2 + x^2} \\ \psi'_C(x) &= \varphi''_C(x) = \frac{C^2 - x^2}{(C^2 + x^2)^2}\end{aligned}$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than 0.

## Conversion Functions

The functions described in this section perform the following conversion operations for vectors:

- Sorting all elements of a vector
- Data type conversion (including floating-point to integer and integer to floating-point)
- Joining several vectors
- Extracting components from a complex vector and constructing a complex vector
- Computing the complex conjugates of vectors
- Cartesian to polar and polar to Cartesian coordinate conversion.

This section also describes the Intel IPP functions that extract real and imaginary components from a complex vector or construct a complex vector using its real and imaginary components. The functions `ippsReal` and `ippsImag` return the real and imaginary parts of a complex vector in a separate vector, respectively. The function `ippsRealToCplx` constructs a complex vector from real and imaginary components stored in two respective vectors. The function `ippsCplxToReal` returns the real and imaginary parts of a complex vector in two respective vectors. The function `ippsMagnitude` computes the magnitude of a complex vector elements.

Additionally this section describes functions that perform the *Viterbi decoding* for V34 receiver.

## SortAscend, SortDescend

*Sorts all elements of a vector.*

---

### Syntax

```
IppStatus ippsSortAscend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortAscend_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSortAscend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortAscend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortAscend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortAscend_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsSortDescend_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSortDescend_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSortDescend_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsSortDescend_32s_I(Ipp32s* pSrcDst, int len);
IppStatus ippsSortDescend_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSortDescend_64f_I(Ipp64f* pSrcDst, int len);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>len</i>	Number of elements in the vector

## Description

The functions `ippsSortAscend` and `ippsSortDescend` are declared in the `ipps.h` file. These functions rearrange all elements of the source vector `pSrcDst` in the ascending or descending order, respectively, and store the result in the destination vector `pSrcDst`.

Example 5-11 below shows how to call the function `ippsSortAscend_8u_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 5-11 Using the Function `ippsSortAscend`

```
void func_sort()
{
    Ipp8u vec[10] = {0,2,4,5,1,8,9,4,6,7};
    IppStatus status;

    status = ippsSortAscend_8u_I(vec,10);
    if(ippStsNoErr != status)
        printf("IPP Error: %s",ippGetStatusString(status));
}

Result
0 1 2 4 4 5 6 7 8 9
```

## SortIndexAscend, SortIndexDescend

*Rearranges elements of the vector and their indexes.*

---

## Syntax

```
IppStatus ippsSortIndexAscend_8u_I(Ipp8u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_16u_I(Ipp16u* pSrcDst, int* pDstIdx, int len);
```

```

IppStatus ippsSortIndexAscend_16s_I(Ipp16s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_32s_I(Ipp32s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_32f_I(Ipp32f* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexAscend_64f_I(Ipp64f* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_8u_I(Ipp8u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_16u_I(Ipp16u* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_16s_I(Ipp16s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_32s_I(Ipp32s* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_32f_I(Ipp32f* pSrcDst, int* pDstIdx, int len);
IppStatus ippsSortIndexDescend_64f_I(Ipp64f* pSrcDst, int* pDstIdx, int len);

```

## Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>pDstIdx</i>	Pointer to the destination vector containing indexes.
<i>len</i>	Number of elements in the vector

## Description

The functions `ippsSortIndexAscend` and `ippsSortIndexDescend` are declared in the `ipps.h` file. These functions rearrange all elements of the source vector *pSrcDst* in the ascending or descending order, respectively, and store the elements in the destination vector *pSrcDst*, and their indexes in the destination vector *pDstIdx*. If some elements are identical, their indexes are not ordered.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## SortRadixAscend, SortRadixDescend

*Sorts all elements of a vector using radix sorting algorithm.*

---

### Syntax

```

IppStatus ippsSortRadixAscend_8u_I(Ipp8u* pSrcDst, Ipp8u* pTmp, Ipp32s len);
IppStatus ippsSortRadixAscend_16u_I(Ipp16u* pSrcDst, Ipp16u* pTmp, Ipp32s
len);
IppStatus ippsSortRadixAscend_16s_I(Ipp16s* pSrcDst, Ipp16s* pTmp, Ipp32s
len);
IppStatus ippsSortRadixAscend_32u_I(Ipp32u* pSrcDst, Ipp32u* pTmp, Ipp32s
len);
IppStatus ippsSortRadixAscend_32s_I(Ipp32s* pSrcDst, Ipp32s* pTmp, Ipp32s
len);
IppStatus ippsSortRadixAscend_32f_I(Ipp32f* pSrcDst, Ipp32f* pTmp, Ipp32s
len);
IppStatus ippsSortRadixAscend_64f_I(Ipp64f* pSrcDst, Ipp64f* pTmp, Ipp32s
len);
IppStatus ippsSortRadixDescend_8u_I(Ipp8u* pSrcDst, Ipp8u* pTmp, Ipp32s len);
IppStatus ippsSortRadixDescend_16u_I(Ipp16u* pSrcDst, Ipp16u* pTmp, Ipp32s
len);
IppStatus ippsSortRadixDescend_16s_I(Ipp16s* pSrcDst, Ipp16s* pTmp, Ipp32s
len);
IppStatus ippsSortRadixDescend_32u_I(Ipp32u* pSrcDst, Ipp32u* pTmp, Ipp32s
len);
IppStatus ippsSortRadixDescend_32s_I(Ipp32s* pSrcDst, Ipp32s* pTmp, Ipp32s
len);
IppStatus ippsSortRadixDescend_32f_I(Ipp32f* pSrcDst, Ipp32f* pTmp, Ipp32s
len);
IppStatus ippsSortRadixDescend_64f_I(Ipp64f* pSrcDst, Ipp64f* pTmp, Ipp32s
len);

```

## Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>pTmp</i>	Pointer to the temporary vector.
<i>len</i>	Number of elements in the vector

## Description

The functions `ippsSortRadixAscend` and `ippsSortRadixDescend` are declared in the `ipps.h` file. These functions rearrange all elements of the source vector *pSrcDst* in the ascending or descending order, respectively, using “radix sort” algorithm, and store the result in the destination vector *pSrcDst*. Temporary vector *pTmp* is required by the algorithm, its size must be equal to the size of the source vector.

Example 5-12 below shows how to call the function `ippsSortRadixAscend_8u_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> or <i>pTmp</i> is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

**Example 5-12 Using the Function ippsSortRadixAscend**

```
void func_sort()
{
    Ipp8u i, vec[10] = {0,2,4,5,1,8,9,4,6,7}, tmp[10];
    IppStatus status;
    status = ippsSortRadixAscend_8u_I(vec,tmp,10);
    if(ippStsNoErr != status)
        printf("IPP Error: %s",ippGetStatusString(status));
    else {
        for(i=0; i<10; i++) printf("%d ",vec[i]); printf("\n");
    }
}

Result

0 1 2 4 4 5 6 7 8 9
```

**SortRadixIndexAscend, SortRadixIndexDescend**

*Indirectly sorts all elements of a vector using radix sorting algorithm.*

---

**Syntax**

```
IppStatus ippsSortRadixIndexAscend_8u(const Ipp32f* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexAscend_16u(const Ipp16u* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexAscend_16s(const Ipp16s* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexAscend_32s(const Ipp32s* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexAscend_32u(const Ipp32u* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);
```



```

IppStatus ippsSortRadixIndexAscend_32f(const Ipp32f* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexDescend_8u(const Ipp32f* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexDescend_16u(const Ipp16u* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexDescend_16s(const Ipp16s* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexDescend_32s(const Ipp32s* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexDescend_32u(const Ipp32u* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

IppStatus ippsSortRadixIndexDescend_32f(const Ipp32f* const pSrc, Ipp32s
srcStrideBytes, Ipp32s* pDstIndx, Ipp32s* pTmpIndx, Ipp32s len);

```

## Parameters

<i>pSrc</i>	Pointer the source sparse keys vector.
<i>srcStrideBytes</i>	Distance in bytes between two consecutive elements of the source vector.
<i>pDstIndx</i>	Pointer to the destination vector of indexes.
<i>pTmpIndx</i>	Pointer to the temporary vector of indexes.
<i>len</i>	Number of elements in the vectors.

## Description

The functions `ippsSortRadixIndexAscend` and `ippsSortRadixIndexDescend` are declared in the `ipps.h` file. These functions indirectly sort all elements of the source sparse keys vector *pSrc* in the ascending or descending order, respectively, using "radix sort" algorithm and store the indexes of resulting arrangement order in the destination vector *pDstIndx*. Elements of the source vector are not rearranged.

Temporary vector *pTmpIndx* is required by the algorithm, its size must be equal to the size of the destination vector and be sufficient to contain *len* number of indexes. Intervals between the elements of the source sparse vector *pSrc* in memory must be equal to the value of *srcStrideBytes*, minimum value of which is equal to the size of the datatype of the key value. The sorting algorithm does not change the relative order of the elements with equal keys.

Example 5-13 below shows how to call the functions `ippsSortRadixindexAscend_8u_I` and `ippsSortRadixindexDescend_8u_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pTmpIndx</i> is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or <i>srcStrideBytes</i> is less than <i>sizeof(key type)</i> .

**Example 5-13 Using the Functions `ippsSortRadixIndexAscend/Descend`**

```

void testsort(void) {
    struct C {
        Ipp8u key1;
        Ipp8u key2;
        float data;
    } c_array[4] = {{0,2,1.0f}, {1,3,2.0f}, {1,4,3.0f}, {8,2,10.0f}};
    int    idx1[4], idx2[4], tmp[4], i;
    ippsSortRadixIndexDescend_8u(&c_array[0].key1, sizeof(C), idx1, tmp, 4);
    ippsSortRadixIndexAscend_8u(&c_array[0].key2, sizeof(C), idx2, tmp, 4);
    printf("%f, %f, %f, %f\n", c_array[idx1[0]].data, c_array[idx1[1]].data,
           c_array[idx1[2]].data, c_array[idx1[3]].data );
    printf("%f, %f, %f, %f\n", c_array[idx2[0]].data, c_array[idx2[1]].data,
           c_array[idx2[2]].data, c_array[idx2[3]].data );
}

```

Result

```

10.0  2.0  3.0  1.0
1.0 10.0  2.0  3.0

```

**SwapBytes**

*Reverses the byte order of a vector.*

---

**Syntax**

```

IppStatus ippsSwapBytes_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsSwapBytes_24u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsSwapBytes_32u(const Ipp32u* pSrc, Ipp32u* pDst, int len);
IppStatus ippsSwapBytes_64u(const Ipp64u* pSrc, Ipp64u* pDst, int len);
IppStatus ippsSwapBytes_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsSwapBytes_24u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsSwapBytes_32u_I(Ipp32u* pSrcDst, int len);

```

```
IppStatus ippsSwapBytes_64u_I(Ipp64u* pSrcDst, int len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsSwapBytes` is declared in the `ipps.h` file. This function reverses the endian order (byte order) of the source vector *pSrc* (*pSrcDst* for the in-place operation) and stores the result in *pDst* (*pSrcDst*). When the low-order byte is stored in memory at the lowest address, and the high-order byte at the highest address, the little-endian order is implemented. When the high-order byte is stored in memory at the lowest address, and the low-order byte at the highest address, the big-endian order is implemented. The function `ippsSwapBytes` allows to switch from one order to the other in either direction.

Example 5-14 below shows how to call the function `ippsSwapBytes_16u_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

**Example 5-14 Using the Function `ippsSwapBytes`**

```

void func_swap()
{
    Ipp16u vec[2] = {0x1234,0x5678};
    IppStatus status;
    status = ippsSwapBytes_16u_I(vec, 2);
    if(ippStsNoErr != status)
        printf("IPP Error: %s",ippGetStatusString(status));
}

result

vec[0] = 0x3412

vec[1] = 0x7856

```

**Convert**

*Converts the data type of a vector and stores the results in a second vector.*

---

**Syntax**

```

IppStatus ippsConvert_8s16s(const Ipp8s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsConvert_8s32f(const Ipp8s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s8s_Sfs(const Ipp16s* pSrc, Ipp8s* pDst, Ipp32u len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len);
IppStatus ippsConvert_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16u32f(const Ipp16u* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_32s16s(const Ipp32s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsConvert_32s32f(const Ipp32s* pSrc, Ipp32f* pDst, int len);

```

```

IppStatus ippsConvert_32s64f(const Ipp32s* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_64s64f(const Ipp64s* pSrc, Ipp64f* pDst, int len);
IppStatus ippsConvert_64f32f(const Ipp64f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsConvert_16s32f_Sfs(const Ipp16s* pSrc, Ipp32f* pDst, int len,
int scaleFactor);
IppStatus ippsConvert_16s64f_Sfs(const Ipp16s* pSrc, Ipp64f* pDst, int len,
int scaleFactor);
IppStatus ippsConvert_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
int scaleFactor);
IppStatus ippsConvert_32s32f_Sfs(const Ipp32s* pSrc, Ipp32f* pDst, int len,
int scaleFactor);
IppStatus ippsConvert_32s64f_Sfs(const Ipp32s* pSrc, Ipp64f* pDst, int len,
int scaleFactor);
IppStatus ippsConvert_32f8s_Sfs(const Ipp32f* pSrc, Ipp8s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f8u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16s_Sfs(const Ipp32f* pSrc, Ipp16s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f16u_Sfs(const Ipp32f* pSrc, Ipp16u* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_32f32s_Sfs(const Ipp32f* pSrc, Ipp32s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64s32s_Sfs(const Ipp64s* pSrc, Ipp32s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64f16s_Sfs(const Ipp64f* pSrc, Ipp16s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);
IppStatus ippsConvert_64f32s_Sfs(const Ipp64f* pSrc, Ipp32s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);

```

```

IppStatus ippsConvert_64f64s_Sfs(const Ipp64f* pSrc, Ipp64s* pDst, Ipp32f
len, IppRoundMode rndMode, int scaleFactor);

IppStatus ippsConvert_24u32u(const Ipp8u* pSrc, Ipp32u* pDst, int len);

IppStatus ippsConvert_24u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);

IppStatus ippsConvert_32u24u_Sfs(const Ipp32u* pSrc, Ipp8u* pDst, int len,
int scaleFactor);

IppStatus ippsConvert_32f24u_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
int scaleFactor);

IppStatus ippsConvert_24s32s(const Ipp8u* pSrc, Ipp32s* pDst, int len);

IppStatus ippsConvert_24s32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);

IppStatus ippsConvert_32s24s_Sfs(const Ipp32s* pSrc, Ipp8u* pDst, int len,
int scaleFactor);

IppStatus ippsConvert_32f24s_Sfs(const Ipp32f* pSrc, Ipp8u* pDst, int len,
int scaleFactor);

IppStatus ippsConvert_16s16f(const Ipp16s* pSrc, Ipp16f* pDst, int len,
IppRoundMode rndMode);

IppStatus ippsConvert_32f16f(const Ipp32f* pSrc, Ipp16f* pDst, int len,
IppRoundMode rndMode);

IppStatus ippsConvert_16f16s_Sfs(const Ipp16f* pSrc, Ipp16s* pDst, int len,
IppRoundMode rndMode, int scaleFactor);

IppStatus ippsConvert_16f32f(const Ipp16f* pSrc, Ipp32f* pDst, int len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>rndMode</i>	Rounding mode, possible values: <i>ippRndZero</i> Specifies that floating-point values must be truncated toward zero. <i>ippRndNear</i> Specifies that floating-point values must be rounded to the nearest even integer.

`ippRndFinancial` Specifies that floating-point values must be rounded down to the nearest integer if decimal value is less than 0.5, or rounded up to the nearest integer if decimal value is equal or greater than 0.5.

`len` Number of elements in the vector

`scaleFactor` Scale factor, refer to [Integer Scaling](#).

## Description

The function `ippsConvert` is declared in the `ipps.h` file. This function converts the type of data contained in the vector `pSrc` and stores the results in `pDst`.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the `scaleFactor` value. The converted result is saturated if it exceeds the output data range.

Functions that operate with `16f` data do not support the `ippRndFinancial` rounding mode.

Example 5-15 below shows how to use the function `ippsConvert`.



**ippConvert\_32f16f** This function has the following specific when it processes the number that are not in the range [MinVal16f..MaxVal16f]:

```
If x > MaxVal16f then {
    If ( rndMode == IppRndNear ) then {
        y = Convert_32f16f (x) = +INF
    }
    If ( rndMode == IppRndZero ) then {
        If ( x == +INF ) then {
            y = Convert_32f16f (x) = +INF
        } else {
            y = Convert_32f16f(x) = MaxVal16f
        }
    }
}

If x < MinVal16f then {
    If ( rndMode == IppRndNear ) then {
        y = Convert_32f16f (x) = -INF
    }
    If ( rndMode == IppRndZero ) then {
        If ( x == -INF ) then {
            y = Convert_32f16f (x) = -INF
        } else {
            y = Convert_32f16f(x) = M in Val16f
        }
    }
}
```

## Return Values

**ippStsNoErr** Indicates no error.

`ippStsNullPtrErr` Indicates an error when the `pDst` or `pSrc` pointer is NULL.

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

`ippStsRndModNotSupportedErr` Indicates an error when the specified rounding mode is not supported.

## Example 5-15 Using the Function `ippsConvert`.

```
Ipp32s src32s[2] = { 33000, -33000 };
Ipp32f src32f[2] = { 126.6, -125.4 };
Ipp32f src_32f[2] = { 113.12, -113.6 };
Ipp32f src1_32f[5] = { -2.5, -2.4, 1.4, 1.5, 1.6 };
Ipp8s src_8s[2] = { 125, -125 };
Ipp8u src8[1] = { 255 };
Ipp32f dst32f[1];
Ipp16s dst16[2];
Ipp16s dst_16s[2];
Ipp8u dst8u[2];
Ipp8s dstN8[2];
Ipp8s dstZ8[2];
int scaleFactor = 0; // no scaling
```

```

ippsConvert_8s16s ( src_8s, dst_16s, 2 );
ippsConvert_8u32f ( src8, dst32f, 1 );
ippsConvert_32s16s ( src32s, dst16, 2 );
ippsConvert_32f8s_Sfs ( src32f, dstN8, 2, ippRndNear, scaleFactor );
ippsConvert_32f8s_Sfs ( src32f, dstZ8, 2, ippRndZero, scaleFactor );
ippsConvert_32f8u_Sfs ( src_32f, dst8u, 2, ippRndNear, scaleFactor );
ippsConvert_32f8s_Sfs ( src1_32f, dstF8, 5, ippRndFinancial, scaleFactor );
result:
8s16s >>          dst_16s = { 125, -125}
8u32f >>          dst32f  = { 255.0 }
32s16s >>          dst16   = { 32767, -32768}    // max, min 16s values
// results for scaleFactor = 0
32f8s_Sfs >>      dstN8   = { 127, -125 }        // scaleFactor = 0
32f8s_Sfs >>      dstZ8   = { 126, -125 }        // scaleFactor = 0
32f8s_Sfs >>      dstF8   = { -3, -2, 1, 2, 2 } // scaleFactor = 0
32f8u_Sfs >>      dst8u   = { 113, 0 }           // scaleFactor = 0
// results for scaleFactor = 2
32f8s_Sfs >>      dstN8   = { 32, -31 }          // scaleFactor = 2
32f8s_Sfs >>      dstZ8   = { 31, -31 }          // scaleFactor = 2
32f8u_Sfs >>      dst8u   = { 28, 0 }            // scaleFactor = 2

```

## Join

*Converts the floating-point data of several vectors to integer data, and stores the results in a single vector.*

### Syntax

```

IppStatus ippsJoin_32f16s_D2L(const Ipp32f** pSrc, int nChannels, int chanLen,
Ipp16s* pDst);

```

## Parameters

<i>pSrc</i>	Pointer to an array of pointers to the source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>nChannels</i>	Number of source vectors.
<i>chanLen</i>	Number of elements in each source vector.

## Description

The function `ippsJoin` is declared in the `ipps.h` file. This function converts floating-point data of *nChannels* input vectors stored in the array *pSrc* to integer data type and writes the results to the destination vector *pDst* in the following order: first element of first vector, first element of second vector, ..., first element of the last vector in the array; second element of first vector, second element of second vector, and so on. The converted value is saturated if it exceeds the output data range.

Example 5-16 below shows how to use the function `ippsJoin`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if <i>nChannels</i> or <i>chanLen</i> is less than or equal to 0.

### Example 5-16 Using the Function `ippsJoin`

```

Ipp32f** pSrc;
Ipp16s pDst[8];
int nChannels = 2;
int chanLen = 4;
int bufSize;
float k = 1.0;
pSrc = (Ipp32f**) ippiMalloc_32f_C1 ( 2, 1, &bufSize );
for(int I = 0; I < 2; i++) {
    pSrc[i] = ( Ipp32f*) ippiMalloc_32f_C1 ( 4, 1, &bufSize );
    for(int j = 0; j < 4; j++)
        pSrc[i][j] = k++;
}
ippsJoin_32f16s_D2L ( (const Ipp32f**) pSrc, nChannels, chanLen, (Ipp16s*) pDst );

```

Result:

1.0	2.0	3.0	4.0	
5.0	6.0	7.0	8.0	pSrc
1	5	2	6	3
7	4	8		pDst

## JoinScaled

*Converts with scaling the floating-point data of several vectors to integer data and stores the results in a single vector.*

### Syntax

```

IppStatus ippsJoinScaled_32f16s_D2L(const Ipp32f** pSrc, int nChannels, int
chanLen, Ipp16s* pDst);

IppStatus ippsJoinScaled_32f24s_D2L(const Ipp32f** pSrc, int nChannels, int
chanLen, Ipp8u* pDst);

```

## Parameters

<i>pSrc</i>	Pointer to an array of pointers to the source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>nChannels</i>	Number of source vectors.
<i>chanLen</i>	Number of elements in each source vector.

## Description

The function `ippJoinScaled` is declared in the `ipp.h` file. This function converts floating-point data of *nChannels* input vectors stored in the array *pSrc* to integer data type with scaling. Input data must be in the range [-1.0, 1.0] that is mapped onto output data type range [*dst\_Min*..*dst\_Max*] (see [Data Ranges](#) for more information on data types and ranges). If the input data exceed the range [-1.0, 1.0], they are saturated. The function uses the following formula for scaling:

$$pDst[n] = dst\_Min + k * [pSrc[n] - (-1)]$$

where  $k = (dst\_Max - dst\_Min) / (1 - (-1))$

Mapped values are rounded to the nearest integer.

The function `ippJoinScaled` writes the results to the destination vector *pDst* in the following order: first element of first vector, first element of second vector, ..., first element of the last vector in the array; second element of first vector, second element of second vector, and so on.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates a then error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if <i>nChannels</i> or <i>chanLen</i> is less than or equal to 0.

## SplitScaled

*Converts the integer data of a vector to floating-point data with scaling, and stores the result in several vectors.*

---

### Syntax

```
IppStatus ippsSplitScaled_16s32f_D2L(const Ipp16s* pSrc, Ipp32f** pDst, int
nChannels, int chanLen);
```

```
IppStatus ippsSplitScaled_24s32f_D2L(const Ipp8u* pSrc, Ipp32f** pDst, int
nChannels, int chanLen);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to an array of pointers to the destination vectors.
<i>nChannels</i>	Number of destination vectors.
<i>chanLen</i>	Number of elements in each destination vector.

### Description

The function `ippsSplitScaled` is declared in the `ipps.h` file. This function converts integer data of the source vector *pSrc* to the floating-point data type with scaling. The whole range of the input data [*src\_Min*..*src\_Max*] (see [Data Ranges](#) for more information on data types and ranges) is mapped onto the output data range [-1.0, 1.0]. The function uses the following formula for scaling:

$$pDst[n] = (-1) + k * (pSrc[n] - src\_Min),$$

where  $k = (1 - (-1)) / (src\_Max - src\_Min)$ .

The function `ippsSplitScaled` writes the results to the *nChannels* destination vectors *pDst* in the following order: first element of the input vector becomes the first element of the first output vector, second element of the input vector becomes the first element of the second output vector, and so on.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates a then error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if <code>nChannels</code> or <code>chanLen</code> is less than or equal to 0.

## Conj

*Stores the complex conjugate values of a vector in a second vector or in-place.*

---

### Syntax

```

IppStatus ippconj_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippconj_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippconj_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippconj_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippconj_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippconj_64fc_I(Ipp64fc* pSrcDst, int len);

```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>pSrcDst</code>	Pointer to the source and destination vector for the in-place operation.
<code>len</code>	Number of elements in the vector.

### Description

The function `ippconj` is declared in the `ipp.h` file. This function stores in `pDst` the element-wise conjugation of the complex vector `pSrc`. The element-wise conjugation of the vector is defined as follows:

```

pDst[n].re = pSrc[n].re
pDst[n].im = pSrc[n].im

```

The in-place flavors of `ippconj` store in `pSrcDst` the element-wise conjugation of the complex vector `pSrcDst`.



The element-wise conjugation of the vector is defined as follows:

```
pSrcDst[n].re = pSrcDst[n].re
pSrcDst[n].im = pSrcDst[n].im
```

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## ConjFlip

*Computes the complex conjugate of a vector and stores the result in reverse order.*

---

### Syntax

```
IppStatus ippConjFlip_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippConjFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippConjFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>len</code>	Number of elements in the vector.

### Description

The function `ippConjFlip` is declared in the `ipps.h` file. This function computes the conjugate of the vector `pSrc` and stores the result, in reverse order, in `pDst`. The complex conjugate, stored in reverse order, is defined as follows:

```
pDst[n] = conj(pSrc[len - n - 1]).
```

Note that if `pSrc` and `pDst` overlap in memory, the function returns unpredictable results.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Magnitude

*Computes the magnitudes of the elements of a complex vector.*

---

### Syntax

```

IppStatus ippsMagnitude_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
Ipp32f* pDst, int len);

IppStatus ippsMagnitude_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
Ipp64f* pDst, int len);

IppStatus ippsMagnitude_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);

IppStatus ippsMagnitude_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp32f* pDst, int len);

IppStatus ippsMagnitude_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsMagnitude_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsMagnitude_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len,
int scaleFactor);

IppStatus ippsMagnitude_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len,
int scaleFactor);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSrcRe</i>	Pointer to the vector with the real parts of complex elements.
<i>pSrcIm</i>	Pointer to the vector with the imaginary parts of complex elements.

<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsMagnitude` is declared in the `ipps.h` file. The complex flavor of this function computes the element-wise magnitude of the complex vector *pSrc* and stores the result in *pDst*. The element-wise magnitude is defined by the formula:

$$\text{magn}[n] = (pSrc[n].re^2 + pSrc[n].im^2)^{1/2}$$

The real flavor of the function `ippsMagnitude` computes the element-wise magnitude of the complex vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the result in *pDst*. The element-wise magnitude is defined by the formula:

$$\text{magn}[n] = (pSrcRe[n]^2 + pSrcIm[n]^2)^{1/2}$$

Example 5-17 below shows how the function `ippsMagnitude` is used to verify the identity  $\sin^2 x + \cos^2 x = 1$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

### Example 5-17 Using the Function `ippsMagnitude`

```
void magn(void) {
    Ipp64f x[6], magn[4];
    int n;
    for (n = 0; n<6; ++n) x[n] = sin(IPP_2PI * n / 8);
    ippsMagnitude_64f(x, x+2, magn, 4);
    printf_64f("magn =", magn, 4, ippStsNoErr);
}
```

Output:

```
magn = 1.000000 1.000000 1.000000 1.000000
Matlab* Analog:

>> n = 0:9; x = sin(2*pi*n/8); z = [x(1:8)+j*x(3:10)]; abs(z(1:4))
```

## MagSquared

*Computes the squared magnitudes of the elements of a complex vector.*

---

### Syntax

```
IppStatus ippMagSquared_32sc32s_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int
len, int scaleFactor);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippMagSquared` is declared in the `ipps.h` file. This function computes the element-wise squared magnitude of the complex vector *pSrc* and stores the result in *pDst*. The element-wise squared magnitude is defined by the formula:

$$magn[n] = pSrc[n].re^2 + pSrc[n].im^2$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Phase

*Computes the phase angles of elements of a complex vector.*

---

### Syntax

```

IppStatus ippsPhase_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPhase_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int len);
IppStatus ippsPhase_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int len, int
scaleFactor);
IppStatus ippsPhase_32sc_Sfs(const Ipp32sc* pSrc, Ipp32s* pDst, int len, int
scaleFactor);
IppStatus ippsPhase_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm, Ipp64f*
pDst, int len);
IppStatus ippsPhase_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm, Ipp32f*
pDst, int len);
IppStatus ippsPhase_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp32f*
pDst, int len);
IppStatus ippsPhase_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp16s* pDst, int len, int scaleFactor);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSrcRe</i>	Pointer to the source vector which stores the real components.
<i>pSrcIm</i>	Pointer to the source vector which stores the imaginary components.
<i>pDst</i>	Pointer to the vector which stores the phase (angle) components of the elements in radians. Phase values are in the range $(-\pi, \pi]$ .
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsPhase` is declared in the `ipps.h` file. This function returns the phase angles of elements of the complex input vector `pSrc`, or the complex input vector whose real and imaginary components are specified in the vectors `pSrcRe` and `pSrcIm`, respectively, and stores the result in the vector `pDst`. Phase values are returned in radians and are in the range  $(-\pi, \pi]$ .

Example 5-18 below shows how to call the function `ippsPhase_32f`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 5-18 Using the `ippsPhase` Function

```
void func_phase()
{
    Ipp32f re[4]= {0.0, 1.0, 0.0, 1.0};
    Ipp32f im[4]= {0.0, 0.0, 1.0, 1.0};
    Ipp32f pDst[4];
    ippsPhase_32f(re,im,pDst, 4);
}

Result
pDst -> {0.0, 0.0, 1.6 0.8}
```

## PowerSpectr

Computes the power spectrum of a complex vector.

### Syntax

```
IppStatus ippsPowerSpectr_64fc(const Ipp64fc* pSrc, Ipp64f* pDst, int len);
IppStatus ippsPowerSpectr_32fc(const Ipp32fc* pSrc, Ipp32f* pDst, int len);
```

```

IppStatus ippsPowerSpectr_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDst, int
len, int scaleFactor);

IppStatus ippsPowerSpectr_16sc32f(const Ipp16sc* pSrc, Ipp32f* pDst, int
len);

IppStatus ippsPowerSpectr_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
Ipp64f* pDst, int len);

IppStatus ippsPowerSpectr_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
Ipp32f* pDst, int len);

IppStatus ippsPowerSpectr_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsPowerSpectr_16s32f(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp32f* pDst, int len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSrcRe</i>	Pointer to the source vector which stores the real components.
<i>pSrcIm</i>	Pointer to the source vector which stores the imaginary components.
<i>pDst</i>	Pointer to the vector which stores the spectrum components of the elements.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsPowerSpectr` is declared in the `ipps.h` file. This function returns the power spectrum of the complex input vector *pSrc*, or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, and stores the results in the vector *pDst*. The power spectrum elements are squares of the magnitudes of the complex input vector elements:

$$pDst[n] = (pSrc[n].re)^2 + (pSrc[n].im)^2, \text{ or } pDst[n] = (pSrcRe[n])^2 + (pSrcIm[n])^2.$$

To compute magnitudes, use the function [ippsMagnitude](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Real

Returns the real part of a complex vector in a second vector.

---

### Syntax

```

IppStatus ippsReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, int len);
IppStatus ippsReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, int len);
IppStatus ippsReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, int len);

```

### Parameters

<i>pSrc</i>	Pointer to the complex source vector.
<i>pDstRe</i>	Pointer to the destination vector with real parts.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsReal` is declared in the `ipps.h` file. This function returns the real part of the complex vector *pSrc* in the vector *pDstRe*.

Example 5-19 below shows how to call the function `ippsReal_32fc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDstRe</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.



### Example 5-19 Using the Function `ippsReal`

```
void func_real()
{
    Ipp32fc pSrc[5] = {{1.2, 3.2}, {0.0, 0.3}, {4.3, 6.7}, {4.2, 0.0}, {1.2, 1.2}};
    Ipp32f pDstRe[5];
    ippsReal_32fc(pSrc, pDstRe, 5);
}

Result
pDstRe -> (1.2, 0.0, 4.3, 4.2, 1.2)
```

## Imag

*Returns the imaginary part of a complex vector in a second vector.*

---

### Syntax

```
IppStatus ippsImag_16sc(const Ipp16sc* pSrc, Ipp16s* pDstIm, int len);
IppStatus ippsImag_32fc(const Ipp32fc* pSrc, Ipp32f* pDstIm, int len);
IppStatus ippsImag_64fc(const Ipp64fc* pSrc, Ipp64f* pDstIm, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the complex source vector.
<i>pDstIm</i>	Pointer to the destination vector with imaginary parts.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsImag` is declared in the `ipps.h` file. This function returns the imaginary part of a complex vector *pSrc* in the vector *pDstIm*.

Example 5-20 below shows how to call the function `ippsImag_32fc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when the *pDstIm* or *pSrc* pointer is *NULL*.  
`ippStsSizeErr` Indicates an error when *len* is less than or equal to 0.

## Example 5-20 Using the Function `ippsImag`

```
void func_imag()
{
    Ipp32fc pSrc[5] = {{1.2, 3.2}, {0.0, 0.3}, {4.3, 6.7}, {4.2, 0.0}, {1.2, 1.2}};
    Ipp32f pDstIm[5];
    ippsImag_32fc(pSrc, pDstIm, 5);
}

Result
pDstIm ->(3.2, 0.3, 6.7, 0.0, 1.2)
```

## RealToCplx

*Returns a complex vector constructed from the real and imaginary parts of two real vectors.*

---

### Syntax

```
IppStatus ippsRealToCplx_16s(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp16sc* pDst, int len);

IppStatus ippsRealToCplx_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
Ipp32fc* pDst, int len);

IppStatus ippsRealToCplx_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
Ipp64fc* pDst, int len);
```

### Parameters

<i>pSrcRe</i>	Pointer to the vector with real parts of complex elements.
<i>pSrcIm</i>	Pointer to the vector with imaginary parts of complex elements.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsRealToCplx` is declared in the `ipps.h` file. This function returns a complex vector `pDst` constructed from the real and imaginary parts of the input vectors `pSrcRe` and `pSrcIm`.

If `pSrcRe` is `NULL`, the real component of the vector is set to zero.

If `pSrcIm` is `NULL`, the imaginary component of the vector is set to zero.

Note that the pointers can not be both `NULL`.

Example 5-21 below shows how to call the function `ippsRealToCplx_32f`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> pointer is <code>NULL</code> . The pointer <code>pSrcRe</code> or <code>pSrcIm</code> can be <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 5-21 Using the Function `ippsRealToCplx`

```
void func_realtocplx()
{
    Ipp32f pSrcRe[6] = {12.2, -2.1, 4.3, 1.1, 2.2, 0.0};
    Ipp32f pSrcIm[6] = {-9.3, 2.3, -5.2, 0.0, -1.0, -1.2};
    Ipp32fc pDst[6];
    ippsRealToCplx_32f(pSrcRe, pSrcIm, pDst, 6);
}

Result
pDst ->{(12.2, -9.3), (-2.1, 2.3), (4.3, -5.2), (1.1, 0.0), (2.2, -1.0), (0.0, 10.0)}
```

## CplxToReal

Returns the real and imaginary parts of a complex vector in two respective vectors.

---

### Syntax

```
IppStatus ippsCplxToReal_16sc(const Ipp16sc* pSrc, Ipp16s* pDstRe, Ipp16s* pDstIm, int len);

IppStatus ippsCplxToReal_32fc(const Ipp32fc* pSrc, Ipp32f* pDstRe, Ipp32f* pDstIm, int len);

IppStatus ippsCplxToReal_64fc(const Ipp64fc* pSrc, Ipp64f* pDstRe, Ipp64f* pDstIm, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the complex vector <i>pSrc</i> .
<i>pDstRe</i>	Pointer to the output vector with real parts.
<i>pDstIm</i>	Pointer to the output vector with imaginary parts.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsCplxToReal` is declared in the `ipps.h` file. This function returns the real and imaginary parts of a complex vector *pSrc* in two vectors *pDstRe* and *pDstIm*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the data vector pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## DemodulateFM

*Converts frequency modulated signal into the initial demodulated form.*

---

### Syntax

```
IppStatus ippsDemodulateFM_CToR_16s(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm, Ipp16s* pDst, int len, Ipp16sc* pDlyPoint);
```

### Parameters

<i>pSrcRe</i>	Pointer to the source vector with real parts of complex elements.
<i>pSrcIm</i>	Pointer to the source vector with imaginary parts of complex elements.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.
<i>pDlyPoint</i>	Pointer to the value required for operation.

### Description

The function `ippsDemodulateFM_CToR` is declared in the `ipps.h` file. This function converts the frequency modulated signal to the initial demodulated form. The modulated signal is presented as a complex vector, *pSrcRe* and *pSrcIm*. The demodulated signal is presented as the real vector *pDst*.

On input the pointer *pDlyPoint* points to the value that corresponds to the demodulated value of the element preceding the first element in the source vector. When operation is completed the pointer points to the demodulated value of the last element.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Threshold

*Performs the threshold operation on the elements of a vector by limiting the element values by specified value.*

---

### Syntax

```

IppStatus ippsThreshold_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s
level, IppCmpOp relOp);

IppStatus ippsThreshold_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
level, IppCmpOp relOp);

IppStatus ippsThreshold_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, Ipp64f
level, IppCmpOp relOp);

IppStatus ippsThreshold_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
Ipp32f level, IppCmpOp relOp);

IppStatus ippsThreshold_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
Ipp64f level, IppCmpOp relOp);

IppStatus ippsThreshold_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
Ipp16s level, IppCmpOp relOp);

IppStatus ippsThreshold_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level, IppCmpOp
relOp);

IppStatus ippsThreshold_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level, IppCmpOp
relOp);

IppStatus ippsThreshold_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level, IppCmpOp
relOp);

IppStatus ippsThreshold_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level,
IppCmpOp relOp);

IppStatus ippsThreshold_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level,
IppCmpOp relOp);

IppStatus ippsThreshold_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level,
IppCmpOp relOp);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>level</i>	Value used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> . This parameter must always be real. For complex versions, it must be positive and represent magnitude.
<i>relOp</i>	Values of this argument specify which relational operator to use and whether <i>level</i> is an upper or lower bound for the input. The <i>relOp</i> must have one of the following values:  <i>ippCmpLess</i> Specifies the "less than" operator and <i>level</i> is a lower bound.  <i>ippCmpGreater</i> Specifies the "greater than" operator and <i>level</i> is an upper bound.

## Description

The function `ippsThreshold` is declared in the `ipps.h` file. This function performs the threshold operation on the vector *pSrc* by limiting each element by the threshold value *level*. Function operation is similar to that of the functions `ippsThreshold_LT`, `ippsThreshold_GT` but its interface contains the *relOp* parameter that specifies the type of the comparison operation to perform.

The in-place flavors of `ippsThreshold` perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

The *relOp* argument specifies which relational operator to use: when its value is `ippCmpGreater` - "greater than", when `ippCmpLess` - "less than," and determines whether *level* is an upper or lower bound for the input, respectively.

The formula for `ippsThreshold` called with the *relOp* = `ippCmpLess` is:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

The formula for `ippsThreshold` called with the `relOp = ippCmpGreater` is:

$$pDst[n] = \begin{cases} level, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold`, the `level` argument is always real. The formula for complex `ippsThreshold` called with the `relOp = ippCmpLess` is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

The formula for complex `ippsThreshold` called with the `relOp = ippCmpGreater` is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

## Application Notes

For all complex versions, `level` must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the function `ippsThreshold`. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the “less than” operation (with the `ippCmpLess` flag) the coordinates are rounded to the infinity (+Inf for positive coordinates, and -Inf for negative), and for the “greater than”



operation (with the *ippCmpGreater* flag) the coordinates are rounded to 0. Code examples 5-22 and 5-23 below show how to use the “complex” function `ippsThreshold_16sc_I` the “real” function `ippsThreshold_16s_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates an error when <i>relOp</i> has an invalid value.
<code>ippStsThreshNegLevelErr</code>	Indicates an error when <i>level</i> for the complex version is negative (see appendix A “ <a href="#">Handling of Special Cases</a> ” for more information).

## Example 5-22 Using the Complex Version of the `ippsThreshold` Function

```

IppStatus cmplx_threshold(void){
    Ipp16sc x[4] = {{2,3},{3,3}, {4,3}, {4,2}};
    /// level is near to the point {2,3} = 3.6
    /// the point {2,3} is to be replaced
    /// the computed coordinates are {2.2188,3.3282}
    /// the point used is {3,4};
    /// notice that it is the point with the phase,
    /// nearest to the source
    IppStatus st = ippsThreshold_16sc_I(x, 4, 4, ippCmpLess);
    printf_16sc("complex threshold result =", x, 4, st);
    return st;
}

```

Output:

```
complex threshold result = {3, 4} {3, 3} {4, 3} {4, 2}
```

**Example 5-23 Using the Real Version of the ippsThreshold Function**

```

IppStatus threshold( void) {
    Ipp16s x[4] = { -1, 0, 2, 3 };
    IppStatus st = ippsThreshold_16s_I(x, 4, 2, ippCmpLess );
    printf_16s("threshold result =", x, 4, st );
    return st;
}

```

Output:

```
threshold result = 2 2 2 3
```

**Threshold\_LT, Threshold\_GT**

*Performs the threshold operation on the elements of a vector by limiting the element values by the specified value.*

---

**Syntax**

```

IppStatus ippsThreshold_LT_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s level);

IppStatus ippsThreshold_LT_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
Ipp32s level);

IppStatus ippsThreshold_LT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f level);

IppStatus ippsThreshold_LT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
Ipp64f level);

IppStatus ippsThreshold_LT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
Ipp32f level);

IppStatus ippsThreshold_LT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
Ipp64f level);

IppStatus ippsThreshold_LT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
Ipp16s level);

IppStatus ippsThreshold_LT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);

```

```

IppStatus ippsThreshold_LT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s level);
IppStatus ippsThreshold_GT_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
Ipp32s level);
IppStatus ippsThreshold_GT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f level);
IppStatus ippsThreshold_GT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
Ipp64f level);
IppStatus ippsThreshold_GT_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
Ipp32f level);
IppStatus ippsThreshold_GT_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
Ipp64f level);
IppStatus ippsThreshold_GT_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
Ipp16s level);
IppStatus ippsThreshold_GT_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GT_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_GT_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GT_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_GT_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level);

```

## Parameters

*pSrc*                                      Pointer to the source vector.

<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>level</i>	Value used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> . This argument must always be real. For complex versions, it must be positive and represent magnitude.

## Description

The functions `ippsThreshold_LT`, `ippsThreshold_GT` are declared in the `ipps.h` file. They implement thresholding of the vector *pSrc* by limiting each element by the threshold value *level*. These functions perform the similar operation to the `ippsThreshold` function but are designed for the fixed type of the compare operation to use: `ippsThreshold_LT` is for the "less than" comparison, while `ippsThreshold_GT` is for the "greater than" comparison.

The in-place flavors perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value *level*.

**ippsThreshold\_LT.** The `ippsThreshold_LT` function performs the operation "less than", and *level* is a lower bound for the input. The formula for `ippsThreshold_LT` is the following:

$$pDst[n] = \begin{cases} level, & pSrc[n] < level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold_LT`, the parameter *level* is always real.

The formula for complex `ippsThreshold_LT` is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) < level \\ pSrc[n], & otherwise \end{cases}$$

**ippsThreshold\_GT.** The function `ippsThreshold_GT` performs the operation "greater than" and *level* is an upper bound for the input.

The formula for `ippsThreshold_GT` is the following:

$$pDst[n] = \begin{cases} level, & pSrc[n] > level \\ pSrc[n], & otherwise \end{cases}$$

For complex versions of the function `ippsThreshold_GT`, the parameter `level` is always real.

The formula for complex `ippsThreshold_GT` is:

$$pDst[n] = \begin{cases} \frac{pSrc[n] \cdot level}{abs(pSrc[n])}, & abs(pSrc[n]) > level \\ pSrc[n], & otherwise \end{cases}$$

## Application Notes

For all complex versions, `level` must be positive and represents a magnitude. The magnitude of the input is limited, but the phase remains unchanged. Zero-valued input is assumed to have zero phase.

A special rule is applied to the integer complex versions of the threshold functions. In general, the resulting point coordinates at the complex plane are not integer. The function rounds them off to integer in such a way that the threshold operation is not performed. Thus, for the “less than” operation (the `ippsThreshold_LT` function) the coordinates are rounded to the infinity (+Inf for positive coordinates, and -Inf for negative), and for the “greater than” operation (the `ippsThreshold_GT` function) the coordinates are rounded to 0.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0
<code>ippStsThreshNegLevelErr</code>	Indicates an error when <code>level</code> for the complex version is negative (see appendix A <a href="#">"Handling of Special Cases"</a> for more information).

## Threshold\_LTAbs, Threshold\_GTAbs

*Performs the threshold operation on the absolute values of elements of a vector.*

---

### Syntax

```

IppStatus ippsThreshold_LTAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s level);

IppStatus ippsThreshold_LTAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
Ipp32s level);

IppStatus ippsThreshold_LTAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f level);

IppStatus ippsThreshold_LTAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
Ipp64f level);

IppStatus ippsThreshold_LTAbs_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_LTAbs_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_LTAbs_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTAbs_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);

IppStatus ippsThreshold_GTAbs_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s level);

IppStatus ippsThreshold_GTAbs_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len,
Ipp32s level);

IppStatus ippsThreshold_GTAbs_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f level);

IppStatus ippsThreshold_GTAbs_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
Ipp64f level);

IppStatus ippsThreshold_GTAbs_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level);
IppStatus ippsThreshold_GTAbs_32s_I(Ipp32s* pSrcDst, int len, Ipp32s level);
IppStatus ippsThreshold_GTAbs_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_GTAbs_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>level</i>	Value used to limit each element of source vector. This argument can not be negative.

## Description

The functions `ippsThreshold_LTAbs` and `ippsThreshold_GTAbs` are declared in the `ipps.h` file. They implement thresholding of the vector *pSrc* by limiting absolute value of each element by the threshold value *level*. These functions perform the compare operation of the fixed type: `ippsThreshold_LTAbs` is for the "less than" comparison, while `ippsThreshold_GTAbs` is for the "greater than" comparison. Elements of the result vector *pDst* have the same sign that the source elements.

The in-place flavors perform the threshold operation on the vector *pSrcDst*.

**`ippsThreshold_LTAbs`.** The `ippsThreshold_LTAbs` function performs the operation "less than", and *level* is a lower bound for the input. The formula for `ippsThreshold_LTAbs` is the following:

$$pDst[n] = \begin{cases} level & \text{if } abs(pSrc[n]) < level, & pSrc[n] \geq 0 \\ -level & \text{if } abs(pSrc[n]) < level, & pSrc[n] < 0 \\ pSrc[n], & \text{otherwise} \end{cases}$$

**`ippsThreshold_GTAbs`.** The function `ippsThreshold_GTAbs` performs the operation "greater than" and *level* is an upper bound for the input. The formula for `ippsThreshold_GTAbs` is the following:

$$pDst[n] = \begin{cases} level & \text{if } abs(pSrc[n]) > level, & pSrc[n] \geq 0 \\ -level & \text{if } abs(pSrc[n]) > level, & pSrc[n] < 0 \\ pSrc[n], & \text{otherwise} \end{cases}$$

Example 5-24 below shows how to use the function `ippsThreshold_LTAbs_64f_I`.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error if *pSrc*, *pDst*, or *pSrcDst* pointer is `NULL`.  
`ippStsSizeErr` Indicates an error if *len* is less than or equal to 0.  
`ippStsThreshNegLevelErr` Indicates an error if *level* is negative.

### Example 5-24 Using the `ippsThreshold_LTAbs` function

```
IppStatus thresholdLTabs( void ) {
    Ipp64f    vec[7] = { -4.01, -4.0, -3.9, 0.0, 2.5, 4.0, 4.5 };
    Ipp64f    level = 4.0;
    IppStatus  st = ippsThreshold_LTAbs_64f_I( vec, 7, level );
    printf("threshold abs    = %f %f %f %f\n",
           vec[0],vec[1],vec[2],vec[3]);
    printf("                  %f %f %f\n",
           vec[4],vec[5],vec[6]);
    return st;
}
```

Output:

```
threshold abs = -4.010000 -4.000000 -4.000000 4.000000
                4.000000  4.000000  4.500000
```

## Threshold\_LTVal, Threshold\_GTVal, Threshold\_LTValGTVal

*Performs the threshold operation on the elements of a vector by limiting the element values by the specified value.*

---

### Syntax

```
IppStatus ippsThreshold_LTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s level, Ipp16s value);
```



---

```
IppStatus ippsThreshold_LTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f level, Ipp32f value);

IppStatus ippsThreshold_LTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
Ipp64f level, Ipp64f value);

IppStatus ippsThreshold_LTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
len, Ipp16s level, Ipp16sc value);

IppStatus ippsThreshold_LTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
len, Ipp32f level, Ipp32fc value);

IppStatus ippsThreshold_LTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
len, Ipp64f level, Ipp64fc value);

IppStatus ippsThreshold_LTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level,
Ipp16s value);

IppStatus ippsThreshold_LTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level,
Ipp32f value);

IppStatus ippsThreshold_LTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level,
Ipp64f value);

IppStatus ippsThreshold_LTVal_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level,
Ipp16sc value);

IppStatus ippsThreshold_LTVal_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level,
Ipp32fc value);

IppStatus ippsThreshold_LTVal_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level,
Ipp64fc value);

IppStatus ippsThreshold_GTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s level, Ipp16s value);

IppStatus ippsThreshold_GTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f level, Ipp32f value);

IppStatus ippsThreshold_GTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
Ipp64f level, Ipp64f value);

IppStatus ippsThreshold_GTVal_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
len, Ipp16s level, Ipp16sc value);

IppStatus ippsThreshold_GTVal_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
len, Ipp32f level, Ipp32fc value);
```

```

IppStatus ippsThreshold_GTVal_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
len, Ipp64f level, Ipp64fc value);

IppStatus ippsThreshold_GTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s level,
Ipp16s value);

IppStatus ippsThreshold_GTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level,
Ipp32f value);

IppStatus ippsThreshold_GTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level,
Ipp64f value);

IppStatus ippsThreshold_GTVal_16sc_I(Ipp16sc* pSrcDst, int len, Ipp16s level,
Ipp16sc value);

IppStatus ippsThreshold_GTVal_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level,
Ipp32fc value);

IppStatus ippsThreshold_GTVal_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level,
Ipp64fc value);

IppStatus ippsThreshold_LTValGTVal_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
len, Ipp16s levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);

IppStatus ippsThreshold_LTValGTVal_32s(const Ipp32s* pSrc, Ipp32s* pDst, int
len, Ipp32s levelLT, Ipp32s valueLT, Ipp32s levelGT, Ipp32s valueGT);

IppStatus ippsThreshold_LTValGTVal_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
len, Ipp32f levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);

IppStatus ippsThreshold_LTValGTVal_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
len, Ipp64f levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);

IppStatus ippsThreshold_LTValGTVal_16s_I(Ipp16s* pSrcDst, int len, Ipp16s
levelLT, Ipp16s valueLT, Ipp16s levelGT, Ipp16s valueGT);

IppStatus ippsThreshold_LTValGTVal_32s_I(Ipp32s* pSrcDst, int len, Ipp32s
levelLT, Ipp32s valueLT, Ipp32s levelGT, Ipp32s valueGT);

IppStatus ippsThreshold_LTValGTVal_32f_I(Ipp32f* pSrcDst, int len, Ipp32f
levelLT, Ipp32f valueLT, Ipp32f levelGT, Ipp32f valueGT);

IppStatus ippsThreshold_LTValGTVal_64f_I(Ipp64f* pSrcDst, int len, Ipp64f
levelLT, Ipp64f valueLT, Ipp64f levelGT, Ipp64f valueGT);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>level</i>	Value used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> . This argument must always be real. For complex versions, it must be positive and represent magnitude.
<i>levelLT</i>	Low bound used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> for the <code>ippsThreshold_LTVaLTVal</code> function.
<i>levelGT</i>	Upper bound used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> for the <code>ippsThreshold_LTVaGTVal</code> function.
<i>value</i>	Value to be assigned to vector elements which are "less than" or "greater than" <i>level</i> .
<i>valueLT</i>	Value to be assigned to vector elements which are less than <i>levelLT</i> for the <code>ippsThreshold_LTVaLTVal</code> function.
<i>valueGT</i>	Value to be assigned to vector elements which are greater than <i>levelGT</i> for the <code>ippsThreshold_LTVaGTVal</code> function.

## Description

These functions are declared in the `ipps.h` file. They perform the threshold operation on the vector *pSrc* by limiting each element by the threshold value.

The in-place flavors of the function perform the threshold operation on the vector *pSrcDst* by limiting each element by the threshold value.

**ippsThreshold\_LTVaL.** The function `ippsThreshold_LTVaLn` performs the operation "less than" and *level* is a lower bound for the input. The vector elements less than *level* are set to *value*.

The formula for `ippsThreshold_LTVaL` is:

$$pDst[n] = \begin{cases} \text{value}, & pSrc[n] < \text{level} \\ pSrc[n], & \text{otherwise} \end{cases}$$

For complex versions of the function `ippsThreshold_LTVal`, the parameter *level* is always real.

The formula for complex `ippsThreshold_LTVal` is:

$$pDst[n] = \begin{cases} \text{value}, & \text{abs}(pSrc[n]) < \text{level} \\ pSrc[n], & \text{otherwise} \end{cases}$$

**ippsThreshold\_GTVal.** The function `ippsThreshold_GTVal` performs the operation “greater than” and *level* is an upper bound for the input. The vector elements greater than *level* are set to *value*.

The formula for `ippsThreshold_GTVal` is:

$$pDst[n] = \begin{cases} \text{value}, & pSrc[n] > \text{level} \\ pSrc[n], & \text{otherwise} \end{cases}$$

For complex versions of the function `ippsThreshold_GTVal`, the parameter *level* is always real.

The formula for complex `ippsThreshold_GTVal` is:

$$pDst[n] = \begin{cases} \text{value}, & \text{abs}(pSrc[n]) > \text{level} \\ pSrc[n], & \text{otherwise} \end{cases}$$

**ippsThreshold\_LTValGTVal.** The function `ippsThreshold_LTValGTVal` checks both the “less than” and “greater than” conditions. The parameter *levelLT* is a lower bound and the parameter *levelGT* is an upper bound for the input. The source vector elements less than *levelLT* are set to *valueLT*, and the source vector elements greater than *levelGT* are set to *valueGT*. The value of *levelLT* must be less than or equal to *levelGT*.

The formula for `ippsThreshold_LTVaIGTVal` is:

$$pDst[n] = \begin{cases} valueLT, & pSrc[n] < levelLT \\ pSrc[n], & levelLT \leq pSrc[n] \leq levelGT \\ valueGT, & pSrc[n] > levelGT \end{cases}$$

For all complex versions, *level* must be positive and represent a magnitude.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsThresholdErr</code>	Indicates an error when <i>levelLT</i> is greater than <i>levelGT</i> .
<code>ippStsThreshNegLevelErr</code>	Indicates an error when <i>level</i> for the complex version is negative (see appendix A “Handling of Special Cases” for more information).

## Threshold\_LTInv

*Computes the inverse of vector elements after limiting their magnitudes by the given lower bound.*

### Syntax

```

IppStatus ippsThreshold_LTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f level);

IppStatus ippsThreshold_LTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
Ipp64f level);

IppStatus ippsThreshold_LTInv_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
len, Ipp32f level);

IppStatus ippsThreshold_LTInv_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
len, Ipp64f level);

IppStatus ippsThreshold_LTInv_32f_I(Ipp32f* pSrcDst, int len, Ipp32f level);

```

```

IppStatus ippsThreshold_LTInv_64f_I(Ipp64f* pSrcDst, int len, Ipp64f level);
IppStatus ippsThreshold_LTInv_32fc_I(Ipp32fc* pSrcDst, int len, Ipp32f level);
IppStatus ippsThreshold_LTInv_64fc_I(Ipp64fc* pSrcDst, int len, Ipp64f level);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>level</i>	Value used to limit each element of <i>pSrc</i> or <i>pSrcDst</i> . This argument must always be real and positive.

### Description

The function `ippsThreshold_LTInv` is declared in the `ipps.h` file. This function computes the inverse of elements of the vector *pSrc* and stores the result in *pDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The in-place flavors of `ippsThreshold_LTInv` compute the inverse of elements of the vector *pSrcDst* and store the result in *pSrcDst*. The computation occurs after first limiting the magnitude of each element by the threshold value *level*.

The threshold operation is performed to avoid division by zero. Since *level* represents a magnitude, it is always real and must be positive. The formula for `ippsThreshold_LTInv` is the following:

$$pDst[n] = \begin{cases} \frac{1}{level}, & abs(pSrc[n]) = 0 \\ \frac{abs(pSrc[n])}{pSrc[n] \cdot level}, & 0 < abs(pSrc[n]) < level \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

If the function encounters zero-valued vector elements and *level* is also 0 (see appendix A "Handling of Special Cases"), the output value is set to *Inf* (infinity), but operation execution is not aborted:

$$pDst[n] = \begin{cases} Inf, & pSrc[n] = 0 \\ \frac{1}{pSrc[n]}, & otherwise \end{cases}$$

Example 5-25 below shows how to use the function `ippsThreshold_LTInv_32f_I`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsThreshNegLevelErr</code>	Indicates an error when <i>level</i> is negative.
<code>ippStsInvZero</code>	Indicates a warning when <i>level</i> and a vector element are equal to 0. Operation execution is not aborted. The value of the destination vector element is <i>Inf</i> .

### Example 5-25 Using the `ippsThreshold_LTInv` Function

```
IppStatus invThreshold(void) {
    Ipp32f x[4] = {-1, 0, 2, 3};
    IppStatus st = ippsThreshold_LTInv_32f_I(x, 4, 0);
    printf_32f("inv threshold =", x, 4, st);
    return st;
}
```

Output:

```
-- warning 4, INF result. Zero value met by invThreshold with zero level
inv threshold = -1.000000 1.#INF00 0.500000 0.333333
```

## CartToPolar

*Converts the elements of a complex vector to polar coordinate form.*

---

### Syntax

```
IppStatus ippsCartToPolar_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
Ipp32f* pDstMagn, Ipp32f* pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
Ipp64f* pDstMagn, Ipp64f* pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_32fc(const Ipp32fc* pSrc, Ipp32f* pDstMagn, Ipp32f*
pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_64fc(const Ipp64fc* pSrc, Ipp64f* pDstMagn, Ipp64f*
pDstPhase, int len);
```

```
IppStatus ippsCartToPolar_16sc_Sfs(const Ipp16sc* pSrc, Ipp16s* pDstMagn,
Ipp16s* pDstPhase, int len, int magnScaleFactor, int phaseScaleFactor);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSrcRe</i>	Pointer to the source vector which stores the real components of Cartesian X,Y pairs.
<i>pSrcIm</i>	Pointer to the source vector which stores the imaginary components of Cartesian X,Y pairs.
<i>pDstMagn</i>	Pointer to the vector which stores the magnitude (radius) component of the elements of the vector <i>pSrc</i> .
<i>pDstPhase</i>	Pointer to the vector which stores the phase (angle) component of the elements of the vector <i>pSrc</i> in radians. Phase values are in the range $(-\pi, \pi]$ .
<i>len</i>	Number of elements in the vector.
<i>magnScaleFactor</i>	Integer scale factor for the magnitude component, refer to <a href="#">Integer Scaling</a> .
<i>phaseScaleFactor</i>	Integer scale factor for the phase component, refer to <a href="#">Integer Scaling</a> .



## Description

The function `ippsCartToPolar` is declared in the `ipps.h` file. This function converts the elements of a complex input vector *pSrc* or the complex input vector whose real and imaginary components are specified in the vectors *pSrcRe* and *pSrcIm*, respectively, to polar coordinate form, and stores the magnitude (radius) component of each element in the vector *pDstMagn* and the phase (angle) component of each element in the vector *pDstPhase*.

Example 5-26 below verifies that points are lying in the unit radius circle.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 5-26 Using the `ippsCartToPolar` Function

```
IppStatus cart2polar( void ) {  
    Ipp64f cart[6], magn[4], phase[4];  
    int n;  
    for (n=0; n<6; ++n) cart[n] = sin(IPP_2PI * n / 8);  
    IppStatus st = ippsCartToPolar_64f( cart, cart+2, magn, phase, 4 );  
    printf_64f( "magn =", magn, 4, st );  
    return st;  
}
```

Output:

```
magn = 1.000000 1.000000 1.000000 1.000000
```

## PolarToCart

*Converts the polar form magnitude/phase pairs stored in input vectors to Cartesian coordinate form.*

---

### Syntax

```
IppStatus ippsPolarToCart_32f(const Ipp32f* pSrcMagn, const Ipp32f* pSrcPhase,
Ipp32f* pDstRe, Ipp32f* pDstIm, int len);
```

```
IppStatus ippsPolarToCart_64f(const Ipp64f* pSrcMagn, const Ipp64f* pSrcPhase,
Ipp64f* pDstRe, Ipp64f* pDstIm, int len);
```

```
IppStatus ippsPolarToCart_32fc(const Ipp32f* pSrcMagn, const Ipp32f*
pSrcPhase, Ipp32fc* pDst, int len);
```

```
IppStatus ippsPolarToCart_64fc(const Ipp64f* pSrcMagn, const Ipp64f*
pSrcPhase, Ipp64fc* pDst, int len);
```

```
IppStatus ippsPolarToCart_16sc(const Ipp16s* pSrcMagn, const Ipp16s*
pSrcPhase, int phaseFixedPoint, Ipp16sc* pDst, int len);
```

```
IppStatus ippsPolarToCart_32sc(const Ipp32s* pSrcMagn, const Ipp32s*
pSrcPhase, int phaseFixedPoint, Ipp32sc* pDst, int len);
```

```
IppStatus ippsPolarToCart_16sc_Sfs(const Ipp16s* pSrcMagn, const Ipp16s*
pSrcPhase, Ipp16sc* pDst, int len, int magnScaleFactor, int phaseScaleFactor);
```

### Parameters

<i>pSrcMagn</i>	Pointer to the source vector which stores the magnitude (radius) components of the elements in polar coordinate form.
<i>pSrcPhase</i>	Pointer to the vector which stores the phase (angle) components of the elements in polar coordinate form in radians.
<i>pDst</i>	Pointer to the resulting vector which stores the complex pairs in Cartesian coordinates (X + iY).
<i>pDstRe</i>	Pointer to the resulting vector which stores the real components of Cartesian X,Y pairs.
<i>pDstIm</i>	Pointer to the resulting vector which stores the imaginary components of Cartesian X,Y pairs.

<i>len</i>	Number of elements in the vectors.
<i>phaseFixedPoint</i>	Specified the position of the decimal fixed point for the phase.
<i>magnScaleFactor</i>	Integer scale factor for the magnitude component, refer to <a href="#">Integer Scaling</a> .
<i>phaseScaleFactor</i>	Integer scale factor for the phase component, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsPolarToCart` is declared in the `ipps.h` file. This function converts the polar form magnitude/phase pairs stored in the input vectors *pSrcMagn* and *pSrcPhase* into a complex vector and stores the results in the vector *pDst*, or stores the real components of the result in the vector *pDstRe* and the imaginary components in the vector *pDstIm*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## MaxOrder

Computes the maximum order of a vector.

### Syntax

```
IppStatus ippsMaxOrder_16s(const Ipp16s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32s(const Ipp32s* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_32f(const Ipp32f* pSrc, int len, int* pOrder);
IppStatus ippsMaxOrder_64f(const Ipp64f* pSrc, int len, int* pOrder);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>len</i>	Number of elements in the vector.
<i>pOrder</i>	Pointer to the result value.

## Description

The function `ippsMaxOrder` is declared in the `ipps.h` file. This function finds the maximum binary number in elements of the exponent vector `pSrc`, and stores the result in `pOrder`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pOrder</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsNaNArg</code>	Indicates a warning when <code>NaN</code> is encountered in the input data vector.

## Preemphasize

*Computes preemphasis of a single precision real signal in-place.*

---

## Syntax

```

IppStatus ippsPreemphasize_16s(Ipp16s* pSrcDst, int len, Ipp32f val);
IppStatus ippsPreemphasize_32f(Ipp32f* pSrcDst, int len, Ipp32f val);

```

## Parameters

<code>pSrcDst</code>	Pointer to the source and destination vector for the in-place operation.
<code>len</code>	Number of elements in the vector.
<code>val</code>	Multiplier factor used in the difference signal preemphasis equation.

## Description

The in-place function `ippsPreemphasize` is declared in the `ipps.h` file. This function computes preemphasis of a real signal `pSrcDst`. The computation is performed according to the difference signal preemphasis equation:

$$y(n) = x(n) - val * x(n - 1),$$

where  $y(n)$  is the preemphasized output,  $x(n)$  is the input, and `val` is the multiplier factor.

Note that usually `val` = 0.95 for speech signals.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Flip

*Reverses the order of elements in a vector.*

### Syntax

```

IppStatus ippsFlip_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsFlip_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsFlip_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsFlip_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsFlip_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsFlip_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsFlip_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsFlip_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsFlip_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsFlip_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsFlip_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsFlip_64fc_I(Ipp64fc* pSrcDst, int len);

```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>pSrcDst</code>	Pointer to the source and destination vector for the in-place operation.
<code>len</code>	Number of elements in the vector.

## Description

The function `ippsFlip` is declared in the `ipps.h` file. This function stores the elements of a source vector `pSrc` to a destination vector `pDst` in reverse order according to the following formula:

$$pDst[n] = pSrc[len - n - 1], \quad n = 0 \dots len - 1$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## FindNearestOne

*Finds an element of the table which is closest to the specified value.*

---

### Syntax

```
IppStatus ippsFindNearestOne_16u(Ipp16u inpVal, Ipp16u* pOutVal, int*
pOutIndex, const Ipp16u *pTable, int tblLen);
```

### Parameters

<code>inpVal</code>	Reference value.
<code>pOutVal</code>	Pointer to the output value.
<code>pOutIndex</code>	Pointer to the output index.
<code>pTable</code>	Pointer to the table for searching.
<code>tblLen</code>	Number of elements in the table.

### Description

The function `ippsFindNearestOne` is declared in the `ipps.h` file. This function searches through the table `pTable` for an element which is closest to the specified reference value `inpVal`. The resulting element and its index are stored in `pOutVal` and `pOutIndex`, respectively. The table elements must satisfy the condition `pTable[n] ≤ pTable[n+1]`. The function uses the following distance criterion for determining the table closest element closest:  $\min(|inpVal - pTable[n]|)$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>tblLen</i> is less than or equal to 0.

## FindNearest

*Finds table elements that are closest to the elements of the specified vector.*

---

### Syntax

```
ippStatus ippsFindNearest_16u(const Ipp16u* pVals, Ipp16u* pOutVals, int*
pOutIndexes, int len, const Ipp16u *pTable, int tblLen);
```

### Parameters

<i>pVals</i>	Pointer to the vector containing reference values.
<i>pOutVals</i>	Pointer to the output vector.
<i>pOutIndexes</i>	Pointer to the array that stores output indexes.
<i>len</i>	Number of elements in the input vector.
<i>pTable</i>	Pointer to the table for searching.
<i>tblLen</i>	Number of elements in the table.

### Description

The function `ippsFindNearest` is declared in the `ipps.h` file. This function searches through the table *pTable* for elements which are closest to the reference elements of the input vector *pVals*. The resulting elements and their indexes are stored in *pOutVals* and *pOutIndexes*, respectively. The table elements must satisfy the condition  $pTable[n] \leq pTable[n+1]$ . The function uses the following distance criterion for determining the table element closest to  $pVals[k] : \min(|pVals[k] - pTable[n]|)$ .

Example 5-27 below shows how to use the function `ippsFindNearest`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>tblLen</code> or <code>len</code> is less than or equal to 0.

## Example 5-27 Using the Function `ippsFindNearest`

```

Ipp16u pVals[] = { 12, 6, 78 };
Ipp16u pTable[] = { 1, 3, 5, 7, 18, 24, 35, 48 };
Ipp16u pOutVals[3];
int pOutIndexes[3];
ippsFindNearest_16u (pVals, pOutVals, pOutIndexes, 3, pTable, 8 );

Result

pOutVals = { 7, 5, 48 }
pOutIndexes = { 3, 2, 7 }

```

## Viterbi Decoder Functions

This section describes the functions that perform operations of Viterbi decoding in the V34 receiver. The encoding is performed in accordance with the algorithm that is described in the section 9.6.3 of the ITU-T Recommendation V.34 (see [\[ITUV34\]](#)).

### GetVarPointDV

*Fills the array with the information about points that are closest to the received point.*

---

#### Syntax

```

IppStatus ippsGetVarPointDV_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, Ipp16sc*
pVariantPoint, const Ipp8u* pLabel, int state);

```

#### Parameters

<code>pSrc</code>	Pointer to the reference point in format 9:7
<code>pDst</code>	Pointer to the closest to the reference point left and bottom complex point in format 9:7



<i>pVariantPoint</i>	Pointer to the array where the point information is stored
<i>pLabel</i>	Pointer to the table that stores the labels.
<i>state</i>	Number of states of a convolution coder.

### Description

The function `ippsGetVarPointDV` is declared in the `ipps.h` file. This function fills the specified array *pVariantPoint* with the information about 2D complex points that are closest to the reference point *pSrc*. This information includes the corresponding labels from the offset table and calculated errors. The number of possible states may be 16, 32, and 64. The number of points in the array depends on the number of states: if *state* = 16, then 4 points will be referenced in the array, if *state* = 32 or 64, then 8 points will be referenced.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .

## CalcStatesDV

Calculates the states of the Viterbi decoder.

### Syntax

```
IppStatus ippsCalcStatesDV_16sc(const Ipp16u* pathError, const Ipp8u*
pNextState, Ipp16u* pBranchError, const Ipp16s* pCurrentSubsetPoint, Ipp16s*
pPathTable, int state, int presentIndex);
```

### Parameters

<i>pPathError</i>	Pointer to the table of path error metrics.
<i>pNextState</i>	Pointer to the next state table.
<i>pBranchError</i>	Pointer to the branch error table.
<i>pCurrentSubsetPoint</i>	Pointer to the current 4D subset.
<i>pPathTable</i>	Pointer to the Viterbi path table.
<i>state</i>	Number of states of a convolutional encoder.
<i>presentIndex</i>	Start index in Viterbi path table.

## Description

The function *ippsCalcStatesDV* is declared in the *ipps.h* file. This function computes the possible states of the Viterbi decoder and fills the table of the accumulated errors *pPathError* and the working Viterbi path table *pPathTable*.

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when any of the specified pointers is <i>NULL</i> .

## BuildSymblTableDV4D

*Fills the array with the information about possible 4D symbols.*

---

## Syntax

```
IppStatus ippsBuildSymblTableDV4D_16sc(const Ipp16sc* pVariantPoint, Ipp16sc*
pCurrentSubsetPoint, int state, int bitInversion);
```

## Parameters

<i>pVariantPoint</i>	Pointer to the array of possible 2D points.
<i>pCurrentSubsetPoint</i>	Pointer to the array with information about possible 4D symbols.
<i>state</i>	Number of states of a convolutional encoder.
<i>bitInversion</i>	Bit inversion.

## Description

The function *ippsBuildSymblTableDV4D* is declared in the *ipps.h* file. This function fills the array *pCurrentSubsetPoint* with information about possible 4D symbols.

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when one of the specified pointers is <i>NULL</i> .

## UpdatePathMetricsDV

*Searches for the state with the minimum path metric.*

---

### Syntax

```
IppStatus ippsUpdatePathMetricsDV_16u(Ipp16u* pBranchError, Ipp16u*
pMinPathError, Ipp8u* pMinSost, Ipp16u* pPathError, int state);
```

### Parameters

<i>pBranchError</i>	Pointer to the branch error table.
<i>pMinPathError</i>	Pointer to the current minimum path error metric.
<i>pMinSost</i>	Pointer to the number of states with the minimum metric.
<i>pPathError</i>	Pointer to the table of accumulated path errors.
<i>state</i>	Number of states of a convolutional encoder.

### Description

The function `ippsBuildSymb1TableDV4D` is declared in the `ipps.h` file. This function searches for the state with the minimum path error metric and stores its number in the *pMinSost*. For all states, the minimum metric is subtracted from the path metric. The branch errors for all states are set to be infinitely large as is required for the next step of the Viterbi decoding.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .

## Windowing Functions

This chapter describes several of the windowing functions commonly used in signal processing. A window is a mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

## Understanding Window Functions

The Intel IPP provides the following functions to generate window samples:

- [Bartlett](#) windowing function
- [Blackman](#) family of windowing functions
- [Hamming](#) windowing function
- [Hann](#) windowing function
- [Kaiser](#) windowing function

These functions generate the window samples and multiply them into an existing signal. To obtain the window samples themselves, initialize the vector argument to the unity vector before calling the window function.

If you want to multiply different frames of a signal by the same window multiple times, it is better to first calculate the window by calling one of the windowing functions (`ippsWinHann`, for example) on a vector with all elements set to 1.0. Then use one of the vector multiplication functions (`ippsMul`, for example) to multiply the window into the signal each time a new set of input samples is available. This avoids repeatedly calculating the window samples. This is illustrated in the following code example 5-28.

### Example 5-28 Window and FFT Many Frames of a Signal

```
void multiFrameWin( void ) {  
    Ipp32f win[LEN], x[LEN], X[LEN];  
    IppsFFTSpec_R_32f* ctx;  
    ippsSet_32f( 1, win, LEN );  
    ippsWinHann_32f_I( win, LEN );  
    /// ... initialize FFT context  
    while(1){  
        /// ... get x signal  
        ///  
        ippsMul_32f_I( win, x, LEN );  
        ippsFFTFwd_RToPack_32f( x, X, ctx, 0 );  
    }  
}
```

### Related Topics

For more information on windowing, see: [Jack89], section 7.3, *Windows in Spectrum Analysis*; [Jack89], section 9.1, *Window-Function Technique*; and [Mit93], section 16-2, *Fourier Analysis of Finite-Time Signals*. For more information on these references, see also the Bibliography at the end of this manual.

## WinBartlett

*Multiplies a vector by a Bartlett windowing function.*

### Syntax

```
IppStatus ippsWinBartlett_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBartlett_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBartlett_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBartlett_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinBartlett_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinBartlett_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinBartlett_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBartlett_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBartlett_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBartlett_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBartlett_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBartlett_64fc_I(Ipp64fc* pSrcDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsWinBartlett` is declared in the `ipps.h` file. This function multiplies the vector `pSrc` by the Bartlett (triangle) window, and stores the result in `pDst`.

The in-place flavors of `ippsWinBartlett` multiply the `pSrcDst` by the Bartlett (triangle) window and store the result in `pSrcDst`.

The complex types multiply both the real and imaginary parts of the vector by the same window.

The Bartlett window is defined as follows:

$$w_{\text{bartlett}}(n) = \begin{cases} \frac{2n}{len-1}, & 0 \leq n \leq \frac{len-1}{2} \\ 2 - \frac{2n}{len-1}, & \frac{len-1}{2} < n \leq len-1 \end{cases}$$

Example 5-29 below shows how to use the function `ippsWinBartlett_32f_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than 3.

## Example 5-29 Using the `ippsWinBartlett` Function

```
void bartlett(void) {
    Ipp32f x[8];
    ippsSet_32f(1, x, 8);
    ippsWinBartlett_32f_I(x, 8);
    printf_32f("bartlett (half) =", x, 4, ippStsNoErr);
}
```

Output:

```

    bartlett (half) = 0.000000 0.285714 0.571429 0.857143
Matlab* Analog:

>> b = bartlett(8); b(1:4)'
```

## WinBlackman

*Multiplies a vector by a Blackman windowing function.*

---

### Syntax

```

IppStatus ippsWinBlackmanQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
int alphaQ15);

IppStatus ippsWinBlackmanQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
len, int alphaQ15);

IppStatus ippsWinBlackman_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
float alpha);

IppStatus ippsWinBlackman_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
float alpha);

IppStatus ippsWinBlackman_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
float alpha);

IppStatus ippsWinBlackman_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
float alpha);

IppStatus ippsWinBlackman_64f(Ipp64f* pSrc, Ipp64f* pDst, int len, double
alpha);

IppStatus ippsWinBlackman_64fc(Ipp64fc* pSrc, Ipp64fc* pDst, int len, double
alpha);

IppStatus ippsWinBlackmanQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinBlackmanQ15_16s_ISfs(Ipp16s* pSrcDst, int len, int alphaQ15,
int scaleFactor);

IppStatus ippsWinBlackmanQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);
IppStatus ippsWinBlackman_16s_I(Ipp16s* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);
```

```

IppStatus ippsWinBlackman_32f_I(Ipp32f* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);
IppStatus ippsWinBlackman_64f_I(Ipp64f* pSrcDst, int len, double alpha);
IppStatus ippsWinBlackman_64fc_I(Ipp64fc* pSrcDst, int len, double alpha);
IppStatus ippsWinBlackmanStd_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBlackmanStd_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
len);
IppStatus ippsWinBlackmanStd_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBlackmanStd_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
len);
IppStatus ippsWinBlackmanStd_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBlackmanStd_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
len);
IppStatus ippsWinBlackmanStd_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanStd_64fc_I(Ipp64fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinBlackmanOpt_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int
len);
IppStatus ippsWinBlackmanOpt_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinBlackmanOpt_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
len);
IppStatus ippsWinBlackmanOpt_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinBlackmanOpt_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
len);
IppStatus ippsWinBlackmanOpt_16s_I(Ipp16s* pSrcDst, int len);

```



```

IppStatus ippsWinBlackmanOpt_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinBlackmanOpt_64fc_I(Ipp64fc* pSrcDst, int len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>alpha</i>	Adjustable parameter associated with the Blackman windowing equation.
<i>alphaQ15</i>	Scaled version of <i>alpha</i> . The <i>scaleFactor</i> value is 15.
<i>len</i>	Number of elements in the vector
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The `ippsWinBlackman` family of functions are declared in the `ipps.h` file. These functions multiply the vector *pSrc* by the Blackman window, and store the result in *pDst*.

The in-place flavors of `ippsWinBlackman` multiply the vector *pSrcDst* by the Blackman window, and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The functions for the Blackman family of windows are defined below.

**ippsWinBlackman.** The function `ippsWinBlackman` allows the application to specify *alpha*. The Blackman window is defined as follows:

$$w_{blackman}(n) = \frac{\alpha + 1}{2} - 0.5 \cos\left(\frac{2\pi n}{len - 1}\right) - \frac{\alpha}{2} \cos\left(\frac{4\pi n}{len - 1}\right)$$

**ippsWinBlackmanQ15.** The function `ippsWinBlackmanQ15` multiplies a vector by a Blackman window with *alphaQ15* scaled according to the factor 15.

**ippsWinBlackmanStd.** The standard Blackman window is provided by the function `ippsWinBlackmanStd`, which simply multiplies a vector by a Blackman window with the standard value of *alpha* shown below:

$$\alpha = -0.16$$

**ippsWinBlackmanOpt.** The function `ippsWinBlackmanOpt` provides a modified window that has a 30 dB/octave roll-off by multiplying a vector by a Blackman window with the optimal value of *alpha* shown below:

$$\alpha = -\frac{0.5}{1 + \cos \frac{2\pi}{len-1}}$$

The minimum *len* is equal to 4. For large *len*, the optimal *alpha* converges asymptotically to the asymptotic *alpha*; the application can use the asymptotic value of *alpha* shown below:

$$\alpha = -0.25$$

Example 5-30 below shows how to use the function `ippsWinBlackmanStd_32f_I`

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than 4 for the function <code>ippsWinBlackmanOpt</code> and less than 3 for all other functions of the family.

### Example 5-30 Using the `ippsWinBlackmanStd` Function

```
void blackman(void) {
    Ipp32f x[8];
    ippsSet_32f(1, x, 8);
    ippsWinBlackmanStd_32f_I(x, 8);
    printf_32f("blackman (half) =", x, 4, ippsNoErr);
}
```

Output:

```
blackman(half) = 0.000000 0.090453 0.459183 0.920364
```

Matlab\* Analog:

```
>> b = blackman(8)'; b(1:4)
```

## WinHamming

*Multiplies a vector by a Hamming windowing function.*

---

### Syntax

```
IppStatus ippsWinHamming_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHamming_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHamming_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHamming_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHamming_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHamming_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHamming_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHamming_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHamming_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHamming_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHamming_32fc_I(Ipp32fc* pSrcDst, int len);
```

```
IppStatus ippsWinHamming_64fc_I(Ipp64fc* pSrcDst, int len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsWinHamming` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Hamming window and stores the result in *pDst*.

The in-place flavors of `ippsWinHamming` multiply the vector *pSrcDst* by the Hamming window and store the result in *pSrcDst*.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hamming window is defined as follows:

$$w_{\text{hamming}}(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{len-1}\right)$$

Example 5-31 below shows how to use the function `ippsWinHamming_32f_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , or <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than 3.

## Example 5-31 Using the ippsWinHamming Function

```
void hamming(void) {
    Ipp32f x[8];
    ippsSet_32f(1, x, 8);
    ippsWinHamming_32f_I(x, 8);
    printf_32f("hamming(half) =", x, 4, ippStsNoErr);
}
```

```

}
Output:
    hamming(half) = 0.080000 0.253195 0.642360 0.954446
Matlab* Analog:
    >> b = hamming(8); b(1:4)'

```

## WinHann

*Multiplies a vector by a Hann windowing function.*

### Syntax

```

IppStatus ippsWinHann_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len);
IppStatus ippsWinHann_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len);
IppStatus ippsWinHann_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsWinHann_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len);
IppStatus ippsWinHann_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsWinHann_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len);
IppStatus ippsWinHann_16s_I(Ipp16s* pSrcDst, int len);
IppStatus ippsWinHann_16sc_I(Ipp16sc* pSrcDst, int len);
IppStatus ippsWinHann_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsWinHann_32fc_I(Ipp32fc* pSrcDst, int len);
IppStatus ippsWinHann_64f_I(Ipp64f* pSrcDst, int len);
IppStatus ippsWinHann_64fc_I(Ipp64fc* pSrcDst, int len);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsWinHann` is declared in the `ipps.h` file. This function multiplies the vector `pSrc` by the Hann window and stores the result in `pDst`.

The in-place flavors of `ippsWinHann` multiply the vector `pSrcDst` by the Hann window and store the result in `pSrcDst`.

The complex types multiply both the real and imaginary parts of the vector by the same window. The Hann window is defined as follows:

$$w_{\text{hann}}(n) = 0.5 - 0.5 \cos\left(\frac{2\pi n}{len-1}\right)$$

Example 5-32 below shows how to use the function `ippsWinHann_32f_I`

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pDst</code> , or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than 3.

## Example 5-32 Using the `ippsWinHann` Function

```
void hann(void) {
    Ipp32f x[8];
    ippsSet_32f(1, x, 8);
    ippsWinHann_32f_I(x, 8);
    printf_32f("hann(half) =", x, 4, ippStsNoErr);
}
Output:
    hann(half) = 0.000000 0.188255 0.611260 0.950484
Matlab* Analog:
    >> N = 8; n = 0:N-1; 0.5*(1-cos(2*pi*n/(N-1)))
```

## WinKaiser

Multiplies a vector by a Kaiser windowing function.

### Syntax

```

IppStatus ippsWinKaiser_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, float
alpha);

IppStatus ippsWinKaiser_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, float
alpha);

IppStatus ippsWinKaiser_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, float
alpha);

IppStatus ippsWinKaiser_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
float alpha);

IppStatus ippsWinKaiser_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
float alpha);

IppStatus ippsWinKaiser_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
float alpha);

IppStatus ippsWinKaiserQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
int alphaQ15);

IppStatus ippsWinKaiserQ15_16sc(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
int alphaQ15);

IppStatus ippsWinKaiser_16s_I(Ipp16s* pSrcDst, int len, float alpha);

IppStatus ippsWinKaiser_32f_I(Ipp32f* pSrcDst, int len, float alpha);

IppStatus ippsWinKaiser_64f_I(Ipp64f* pSrcDst, int len, float alpha);

IppStatus ippsWinKaiser_16sc_I(Ipp16sc* pSrcDst, int len, float alpha);

IppStatus ippsWinKaiser_32fc_I(Ipp32fc* pSrcDst, int len, float alpha);

IppStatus ippsWinKaiser_64fc_I(Ipp64fc* pSrcDst, int len, float alpha);

IppStatus ippsWinKaiserQ15_16s_I(Ipp16s* pSrcDst, int len, int alphaQ15);

IppStatus ippsWinKaiserQ15_16sc_I(Ipp16sc* pSrcDst, int len, int alphaQ15);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>alpha</i>	Adjustable parameter associated with the Kaiser windowing equation.
alphaQ15	Scaled version of <i>alpha</i> . The <i>scaleFactor</i> value is 15.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsWinKaiser` is declared in the `ipps.h` file. This function multiplies the vector *pSrc* by the Kaiser window, and stores the result in *pDst*.

The in-place flavors of `ippsWinKaiser` multiply the vector *pSrcDst* by the Kaiser window and store the result in *pSrcDst*.

**ippsWinKaiser.** The function `ippsWinKaiser` allows the application to specify *alpha*. The function multiplies both real and imaginary parts of the complex vector by the same window. The Kaiser family of windows are defined as follows:

$$w_{kaiser}(n) = \frac{I_0\left(\alpha \sqrt{\left(\frac{len-1}{2}\right)^2 - \left(n - \left(\frac{len-1}{2}\right)\right)^2}\right)}{I_0\left(\alpha \left(\frac{len-1}{2}\right)\right)}$$

Here  $I_0()$  is the modified zero-order Bessel function of the first kind.

**ippsWinKaiserQ15.** The function `ippsWinKaiserQ15` multiplies a vector by a Kaiser window with *alphaQ15* scaled according to the factor 15.

Example 5-33 below shows how to use the function `ippsWinKaiser_32f_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------



`ippStsNullPtrErr` Indicates an error when the `pDst`, `pSrc`, or `pSrcDst` pointer is NULL.

`ippStsSizeErr` Indicates an error when `len` is less than 1.

`ippStsHugeWinErr` Indicates an error when the Kaiser window is too big.

### Example 5-33 Using the `ippsWinKaiser` Function

```
void kaiser(void) {
    Ipp32f x[8];
    IppStatus st;
    ippsSet_32f(1, x, 8);
    st = ippsWinKaiser_32f_I( x, 8, 1.0f );
    printf_32f("kaiser(half) =", x, 4, ippStsNoErr);
}
Output:
kaiser(half) = 0.135534 0.429046 0.755146 0.970290
Matlab* Analog:
>> kaiser(8,7/2)'
```

## Statistical Functions

This section describes the Intel IPP functions that compute the vector measure values: maximum, minimum, mean, and standard deviation.

### Sum

*Computes the sum of the elements of a vector.*

#### Syntax

```
IppStatus ippsSum_32f(const Ipp32f* pSrc, int len, Ipp32f* pSum,
IppHintAlgorithm hint);

IppStatus ippsSum_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pSum,
IppHintAlgorithm hint);

IppStatus ippsSum_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);

IppStatus ippsSum_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pSum);

IppStatus ippsSum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pSum, int
scaleFactor);
```

```

IppStatus ippsSum_32s_Sfs(const Ipp32s* pSrc, int len, Ipp32s* pSum, int
scaleFactor);

IppStatus ippsSum_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pSum, int
scaleFactor);

IppStatus ippsSum_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pSum, int
scaleFactor);

IppStatus ippsSum_16sc32sc_Sfs(const Ipp16sc* pSrc, int len, Ipp32sc* pSum,
int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSum</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>hint</i>	Suggests using specific code. The possible values for the <i>hint</i> argument are described in <a href="#">Hint Arguments</a> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsSum` is declared in the `ipps.h` file. This function computes the sum of the elements of the vector *pSrc* and stores the result in *pSum*.

The sum of the elements of *pSrc* is defined by the formula:

$$sum = \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

When computing the sum of integer numbers, the output result can exceed the data range and become saturated. To get a precise result, use the scale factor. The scaling is performed in accordance with the *scaleFactor* value.

Example 5-34 below shows how to use the function `ippsSum`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSum</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 5-34 Using the `ippsSum` Function

```
void sum(void) {
    Ipp16s x[4] = {-32768, 32767, 32767, 32767}, sm;
    ippsSum_16s_Sfs(x, 4, &sm, 1);
    printf_16s("sum =", &sm, 1, ippStsNoErr);
}
Output:
    sum = 32766
Matlab* Analog:
    >> x = [-32768, 32767, 32767, 32767]; sum(x)/2
```

## Max

*Returns the maximum value of a vector.*

### Syntax

```
IppStatus ippsMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax);
IppStatus ippsMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax);
IppStatus ippsMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax);
IppStatus ippsMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMax</code>	Pointer to the output result.
<code>len</code>	Number of elements in the vector

### Description

The function `ippsMax` is declared in the `ipps.h` file. This function returns the maximum value of the input vector `pSrc`, and stores the result in `pMax`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMax</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## MaxIdx

*Returns the maximum value of a vector and the index of the maximum element.*

---

### Syntax

```

IppStatus ippMaxIdx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMax, int*
pIdx);

IppStatus ippMaxIdx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMax, int*
pIdx);

IppStatus ippMaxIdx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMax, int*
pIdx);

IppStatus ippMaxIdx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMax, int*
pIdx);

```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMax</code>	Pointer to the output result.
<code>len</code>	Number of elements in the vector.
<code>pIdx</code>	Pointer to the index value of the maximum element.

### Description

The function `ippMaxIdx` is declared in the `ipp.h` file. This function returns the maximum value of the input vector `pSrc`, and stores the result in `pMax`. If `pIdx` is not a `NULL` pointer, the function returns the index of the maximum element and stores it in `pIdx`. If there are several equal maximum elements, the first index from the beginning is returned. Code example 5-35 below demonstrates how to use the function `ippMaxIdx`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 5-35 Using the function `ippsMaxIndx`

```
Ipp16s src[] = { 1, -2, 3, 8, -6 };  
Ipp16s max;  
int len = 5;  
int indx;  
ippsMaxIndx_16s ( src, len, &max, &indx );  
result: max = 8  indx = 3
```

## MaxAbs

Returns the maximum absolute value of a vector.

### Syntax

```
IppStatus ippsMaxAbs_16s(const Ipp16s* pSrc, int len, Ipp16s* pMaxAbs);  
IppStatus ippsMaxAbs_32s(const Ipp32s* pSrc, int len, Ipp32s* pMaxAbs);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMaxAbs</code>	Pointer to the output result.
<code>len</code>	Number of elements in the vector.

### Description

The function `ippsMaxAbs` is declared in the `ipps.h` file. This function returns the maximum absolute value of the input vector `pSrc`, and stores the result in `pMaxAbs`.

Example 5-36 below shows how to use the function `ippsMaxAbs_16s`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when the `pMaxAbs` or `pSrc` pointer is NULL.  
`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

## Example 5-36 Using the Function `ippsMaxAbs`

```
Ippl6s src[5] = { 2, -8, -3, -1, 7 };
```

```
Ippl6s maxAbs;
```

```
ippsMaxAbs_16s ( src, 5, &maxAbs );
```

Result

```
maxAbs = 8
```

## MaxAbsIndx

*Returns the maximum absolute value of a vector and the index of the corresponding element.*

---

### Syntax

```
IppStatus ippsMaxAbsIndx_16s(const Ippl6s* pSrc, int len, Ippl6s* pMaxAbs,
int* pIndx);
```

```
IppStatus ippsMaxAbsIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMaxAbs,
int* pIndx);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMaxAbs</code>	Pointer to the output result.
<code>len</code>	Number of elements in the vector.
<code>pIndx</code>	Pointer to the index value of the maximum element.

### Description

The function `ippsMaxAbsIndx` is declared in the `ipps.h` file. This function returns the maximum absolute value `pMaxAbs` of the input vector `pSrc`, and the index of the corresponding element `pIndx`. If there are several elements with the equal maximum absolute value, the first index from the beginning is returned.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Min

*Returns the minimum value of a vector.*

---

### Syntax

```
IppStatus ippMin_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin);  
IppStatus ippMin_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin);  
IppStatus ippMin_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin);  
IppStatus ippMin_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMin</code>	Pointer to the output result.
<code>len</code>	Number of elements in the vector.

### Description

The function `ippMin` is declared in the `ipps.h` file. This function returns the minimum value of the input vector `pSrc`, and stores the result in `pMin`.

Example 5-37 below shows how to use the function `ippMin`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMin</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 5-37 Using the function `ippsMin`

```

Ipp16s src = { 1, -2, 3, 8, -6};

Ipp16s min;

int len = 5;

ippsMin_16s (src, len, &min );

result: min = -6

```

## MinIndx

*Returns the minimum value of a vector and the index of the minimum element.*

---

### Syntax

```

IppStatus ippsMinIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, int*
pIndx);

IppStatus ippsMinIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, int*
pIndx);

IppStatus ippsMinIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, int*
pIndx);

IppStatus ippsMinIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, int*
pIndx);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMin</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>pIndx</i>	Pointer to the index value of the minimum element.

### Description

The function `ippsMinIndx` is declared in the `ipps.h` file. This function returns the minimum value of the input vector *pSrc* and stores the result in *pMin*. If *pIndx* is not a NULL pointer, the function returns the index of the minimum element and stores it in *pIndx*. If there are several equal minimum elements, the first index from the beginning is returned.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMin</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## MinAbs

Returns the minimum absolute value of a vector.

### Syntax

```
IppStatus ippMinAbs_16s(const Ipp16s* pSrc, int len, Ipp16s* pMinAbs);  
IppStatus ippMinAbs_32s(const Ipp32s* pSrc, int len, Ipp32s* pMinAbs);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMinAbs</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippMinAbs` is declared in the `ipps.h` file. This function returns the minimum absolute value of the input vector *pSrc*, and stores the result in *pMinAbs*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMinAbs</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## MinAbsIndx

Returns the minimum absolute value of a vector and the index of the corresponding element.

---

### Syntax

```
IppStatus ippsMinAbsIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMinAbs,
int* pIndx);

IppStatus ippsMinAbsIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMinAbs,
int* pIndx);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMinAbs</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>pIndx</i>	Pointer to the index value of the corresponding element.

### Description

The function `ippsMinAbsIndx` is declared in the `ipps.h` file. This function returns the minimum absolute value *pMinAbs* of the input vector *pSrc*, and the index of the corresponding element *pIndx*. If there are several elements with equal maximum absolute value, the first index from the beginning is returned.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## MinMax

Returns the maximum and minimum values of a vector.

---

### Syntax

```
ippStatus ippMinMax_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, Ipp8u*
pMax);

ippStatus ippMinMax_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin, Ipp16u*
pMax);

ippStatus ippMinMax_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, Ipp16s*
pMax);

ippStatus ippMinMax_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin, Ipp32u*
pMax);

ippStatus ippMinMax_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, Ipp32s*
pMax);

ippStatus ippMinMax_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, Ipp32f*
pMax);

ippStatus ippMinMax_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, Ipp64f*
pMax);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMin</i>	Pointer to the minimum value.
<i>pMax</i>	Pointer to the maximum value.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippMinMax` is declared in the `ipp.h` file. This function returns the minimum and maximum values of the input vector *pSrc*, and stores the results in *pMin* and *pMax*, respectively.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMin</code> or <code>pSrc</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## MinMaxIndx

Returns the maximum and minimum values of a vector and the indexes of the corresponding elements.

---

### Syntax

```

IppStatus ippMinMaxIndx_8u(const Ipp8u* pSrc, int len, Ipp8u* pMin, int*
pMinIndx, Ipp8u* pMax, int* pMaxIndx);

IppStatus ippMinMaxIndx_16u(const Ipp16u* pSrc, int len, Ipp16u* pMin, int*
pMinIndx, Ipp16u* pMax, int* pMaxIndx);

IppStatus ippMinMaxIndx_16s(const Ipp16s* pSrc, int len, Ipp16s* pMin, int*
pMinIndx, Ipp16s* pMax, int* pMaxIndx);

IppStatus ippMinMaxIndx_32u(const Ipp32u* pSrc, int len, Ipp32u* pMin, int*
pMinIndx, Ipp32u* pMax, int* pMaxIndx);

IppStatus ippMinMaxIndx_32s(const Ipp32s* pSrc, int len, Ipp32s* pMin, int*
pMinIndx, Ipp32s* pMax, int* pMaxIndx);

IppStatus ippMinMaxIndx_32f(const Ipp32f* pSrc, int len, Ipp32f* pMin, int*
pMinIndx, Ipp32f* pMax, int* pMaxIndx);

IppStatus ippMinMaxIndx_64f(const Ipp64f* pSrc, int len, Ipp64f* pMin, int*
pMinIndx, Ipp64f* pMax, int* pMaxIndx);

```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pMin</code>	Pointer to the minimum value.
<code>pMax</code>	Pointer to the maximum value.
<code>len</code>	Number of elements in the vector.
<code>pMinIndx</code>	Pointer to the index value of the minimum element.
<code>pMaxIndx</code>	Pointer to the index value of the maximum element.

## Description

The function `ippsMinMaxIndx` is declared in the `ipps.h` file. This function returns the minimum and maximum values of the input vector *pSrc* and stores the result in *pMin* and *pMax*, respectively. The function also returns the indexes of the minimum and maximum elements and stores them in *pMinIndx* and *pMaxIndx*, respectively. If there are several equal minimum or maximum elements, the first index from the beginning is returned.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Mean

*Computes the mean value of a vector.*

---

### Syntax

```

IppStatus ippsMean_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean,
IppHintAlgorithm hint);

IppStatus ippsMean_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pMean,
IppHintAlgorithm hint);

IppStatus ippsMean_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean);

IppStatus ippsMean_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pMean);

IppStatus ippsMean_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean, int
scaleFactor);

IppStatus ippsMean_32s_Sfs(const Ipp32s* pSrc, int len, Ipp32s* pMean, int
scaleFactor);

IppStatus ippsMean_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pMean,
int scaleFactor);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMean</i>	Pointer to the output result.

<i>len</i>	Number of elements in the vector
<i>hint</i>	Suggests using specific code. The possible values for the <i>hint</i> argument are described in <a href="#">Hint Arguments</a> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsMean` is declared in the `ipps.h` file. This function computes the mean (average) of the vector *pSrc*, and stores the result in *pMean*. The mean of *pSrc* is defined by the formula:

$$mean = \frac{1}{len} \sum_{n=0}^{len-1} pSrc[n]$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Example 5-38 below shows how to use the function `ippsMean_32f`

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMean</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 5-38 Using the `ippsMean` Function

```
void mean(void) {
    Ipp32f *x = ippsMalloc_32f(1000), mean;
    int i;
    for(i = 0; i<1000; ++i) x[i] = (float)rand() / RAND_MAX;
    ippsMean_32f(x, 1000, &mean, ippAlgHintFast);
    printf_32f("mean =", &mean, 1, ippStsNoErr);
    ippsFree(x);
}
```

Output:

```
mean = 0.492591
```

Matlab® Analog:

```
>> x = rand(1,1000); mean(x)
```

## StdDev

Computes the standard deviation value of a vector.

### Syntax

```

IppStatus ippsStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pStdDev,
IppHintAlgorithm hint);

IppStatus ippsStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pStdDev);

IppStatus ippsStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pStdDev,
int scaleFactor);

IppStatus ippsStdDev_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pStdDev,
int scaleFactor);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pStdDev</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>hint</i>	Suggests using specific code. The possible values for the <i>hint</i> argument are described in <a href="#">Hint Arguments</a> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsStdDev` is declared in the `ipps.h` file. This function computes the standard deviation of the input vector *pSrc*, and stores the result in *pStdDev*. The vector length can not be less than 2. The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$\text{stdDev} = \sqrt{\frac{\sum_{n=0}^{\text{len}-1} (pSrc[n] - \text{mean}(pSrc))^2}{\text{len} - 1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

Example 5-39 below shows how to use the function `ippStdDev_32f`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pStdDev</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 1.

### Example 5-39 Using the `ippStdDev` Function

```
void stdev(void) {
    Ipp32f *x = ippMalloc_32f(1000), stdev;
    int i;
    for (i = 0; i < 1000; ++i) x[i] = (float)rand() / RAND_MAX;
    ippStdDev_32f(x, 1000, &stdev, ippAlgHintFast);
    printf_32f("stdev =", &stdev, 1, ippStsNoErr);
    ippFree(x);
}
```

Output:

```
stdev = 0.286813
```

Matlab® Analog:

```
>> x = rand(1,1000); std(x)
```

## MeanStdDev

*Computes the mean value and the standard deviation value of a vector.*

---

### Syntax

```
IppStatus ippMeanStdDev_32f(const Ipp32f* pSrc, int len, Ipp32f* pMean,
Ipp32f* pStdDev, IppHintAlgorithm hint);
```

```
IppStatus ippMeanStdDev_64f(const Ipp64f* pSrc, int len, Ipp64f* pMean,
Ipp64f* pStdDev);
```

```
IppStatus ippMeanStdDev_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pMean,
Ipp16s* pStdDev, int scaleFactor);
```

```
IppStatus ippMeanStdDev_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s*
pMean, Ipp32s* pStdDev, int scaleFactor);
```



## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pMean</i>	Pointer to the output result - mean value.
<i>pStdDev</i>	Pointer to the output result - standard deviation.
<i>len</i>	Number of elements in the vector
<i>hint</i>	Suggests using specific code. The possible values for the <i>hint</i> argument are described in <a href="#">Hint Arguments</a> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsMeanStdDev` is declared in the `ipps.h` file. This function computes both the mean value and the standard deviation of the input vector *pSrc*, and stores the results in *pMean* and *pStdDev* respectively. The vector length can not be less than 2. The mean of *pSrc* is defined by the formula:

$$\text{mean} = \frac{1}{\text{len}} \sum_{n=0}^{\text{len}-1} pSrc[n]$$

The standard deviation of *pSrc* is defined by the unbiased estimate formula:

$$\text{stdDev} = \sqrt{\frac{\sum_{n=0}^{\text{len}-1} (pSrc[n] - \text{mean}(pSrc))^2}{\text{len} - 1}}$$

The *hint* argument suggests using specific code, either faster but less accurate calculation, or more accurate but slower calculation.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.  
`ippStsSizeErr` Indicates an error when `len` is less than or equal to 1.

## Norm

Computes the C, L1, L2, or L2Sqr norm of a vector.

### Syntax

```

IppStatus ippsNorm_Inf_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_32fc32f(const Ipp32fc* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_Inf_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_Inf_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm,
int scaleFactor);

IppStatus ippsNorm_L1_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L1_32fc64f(const Ipp32fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L1_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm,
int scaleFactor);

IppStatus ippsNorm_L1_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s* pNorm,
int scaleFactor);

IppStatus ippsNorm_L2_32f(const Ipp32f* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_64f(const Ipp64f* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_16s32f(const Ipp16s* pSrc, int len, Ipp32f* pNorm);
IppStatus ippsNorm_L2_32fc64f(const Ipp32fc* pSrc, int len, Ipp64f* pNorm);
IppStatus ippsNorm_L2_64fc64f(const Ipp64fc* pSrc, int len, Ipp64f* pNorm);

```

```
IppStatus ippsNorm_L2_16s32s_Sfs(const Ipp16s* pSrc, int len, Ipp32s* pNorm,
int scaleFactor);
```

```
IppStatus ippsNorm_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc, int len, Ipp64s*
pNorm, int scaleFactor);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pNorm</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsNorm` is declared in the `ipps.h` file. This function computes the C, L1, L2, or L2Sqr norm of the source vector *pSrc* and stores the result in *pNorm*.

**ippsNorm\_Inf.** The function `ippsNorm_Inf` computes the C norm defined by the formula:

$$Norm_C = \max_{n=0}^{len-1} |pSrc[n]|$$

**ippsNorm\_L1.** The function `ippsNorm_L1` computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc[n]|$$

**ippsNorm\_L2.** The function `ippsNorm_L2` computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc[n]|^2}$$

**ippsNorm\_L2Sqr.** The function `ippsNorm_L2Sqr` computes the L2Sqr norm defined as square of the L2 norm.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the `scaleFactor` value.

## Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pNorm</code> pointer is <code>NULL</code> .
<code>ippsStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## NormDiff

*Computes the C, L1, L2, or L2Sqr norm of two vectors' difference.*

---

### Syntax

```
IppStatus ippsNormDiff_Inf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len, Ipp32f* pNorm);
```

```
IppStatus ippsNormDiff_Inf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len, Ipp64f* pNorm);
```

```
IppStatus ippsNormDiff_Inf_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len, Ipp32f* pNorm);
```

```
IppStatus ippsNormDiff_Inf_32fc32f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int len, Ipp32f* pNorm);
```

```
IppStatus ippsNormDiff_Inf_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int len, Ipp64f* pNorm);
```

```
IppStatus ippsNormDiff_Inf_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len, Ipp32s* pNorm, int scaleFactor);
```

```
IppStatus ippsNormDiff_L1_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len, Ipp32f* pNorm);
```

```
IppStatus ippsNormDiff_L1_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len, Ipp64f* pNorm);
```

```
IppStatus ippsNormDiff_L1_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int len, Ipp32f* pNorm);
```

```

IppStatus ippsNormDiff_L1_32fc64f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
int len, Ipp64f* pNorm);

IppStatus ippsNormDiff_L1_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
int len, Ipp64f* pNorm);

IppStatus ippsNormDiff_L1_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, int len, Ipp32s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L1_16s64s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, int len, Ipp64s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L2_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int
len, Ipp32f* pNorm);

IppStatus ippsNormDiff_L2_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int
len, Ipp64f* pNorm);

IppStatus ippsNormDiff_L2_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
int len, Ipp32f* pNorm);

IppStatus ippsNormDiff_L2_32fc64f(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
int len, Ipp64f* pNorm);

IppStatus ippsNormDiff_L2_64fc64f(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
int len, Ipp64f* pNorm);

IppStatus ippsNormDiff_L2_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, int len, Ipp32s* pNorm, int scaleFactor);

IppStatus ippsNormDiff_L2Sqr_16s64s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, int len, Ipp64s* pNorm, int scaleFactor);

```

## Parameters

<i>pSrc1, pSrc2</i>	Pointers to the two source vectors; <i>pSrc2</i> can be NULL.
<i>pNorm</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsNorm` is declared in the `ipps.h` file. This function computes the C, L1, L2, or L2Sqr norm of the source vectors' difference, and stores the result in *pNorm*.

**ippsNormDiff\_Inf.** The function `ippsNormDiff_Inf` computes the C norm defined by the formula:

$$Norm_{Inf} = \max_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

**ippsNormDiff\_L1.** The function `ippsNormDiff_L1` computes the L1 norm defined by the formula:

$$Norm_{L1} = \sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|$$

**ippsNormDiff\_L2.** The function `ippsNormDiff_L2` computes the L2 norm defined by the formula:

$$Norm_{L2} = \sqrt{\sum_{n=0}^{len-1} |pSrc1[n] - pSrc2[n]|^2}$$

**ippsNormDiff\_L2Sqr.** The function `ippsNormDiff_L2Sqr` computes the L2Sqr norm defined as square of the L2 norm.

Functions with `Sfs` suffixes perform scaling of the result value in accordance with the `scaleFactor` value.

Example 5-40 below shows how to use the function `ippsNormDiff`.

### Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> , <code>pSrc2</code> , or <code>pNorm</code> pointer is NULL.
<code>ippsStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

### Example 5-40 Using the ippsNorm Function

```
int norm( void ) {
    Ipp16s x[LEN];
    Ipp32f Norm[3];
    IppStatus st;
    int i;
    for( i=0; i<LEN; ++i ) x[i] = (Ipp16s)rand();
    ippsNormDiff_Inf_16s32f( x, 0, LEN, Norm );
    ippsNormDiff_L1_16s32f( x, 0, LEN, Norm+1 );
    st = ippsNormDiff_L2_16s32f( x, 0, LEN, Norm+2 );
    printf_32f("Norm (oo,L1,L2) =", Norm, 3, st );
    return Norm[2] <= Norm[1] && Norm[1] <= LEN*Norm[0];
}
Output:
    Norm (oo,L1,L2) = 31993.000000 1526460.000000 180270.781250
Matlab* analog:
    >> x = 32767*rand(1,100);norm(x,inf),norm(x,1),norm(x,2)
```

## DotProd

*Computes the dot product of two vectors.*

### Syntax

```
IppStatus ippsDotProd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int len,
Ipp32f* pDp);

IppStatus ippsDotProd_32fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, int
len, Ipp32fc* pDp);

IppStatus ippsDotProd_32f32fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2, int
len, Ipp32fc* pDp);

IppStatus ippsDotProd_32f64f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, int
len, Ipp64f* pDp);

IppStatus ippsDotProd_32fc64fc(const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
int len, Ipp64fc* pDp);

IppStatus ippsDotProd_32f32fc64fc(const Ipp32f* pSrc1, const Ipp32fc* pSrc2,
int len, Ipp64fc* pDp);

IppStatus ippsDotProd_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, int len,
Ipp64f* pDp);

IppStatus ippsDotProd_64fc(const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, int
len, Ipp64fc* pDp);
```

```

IppStatus ippsDotProd_64f64fc(const Ipp64f* pSrc1, const Ipp64fc* pSrc2, int
len, Ipp64fc* pDp);

IppStatus ippsDotProd_16s64s(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
len, Ipp64s* pDp);

IppStatus ippsDotProd_16sc64sc(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
int len, Ipp64sc* pDp);

IppStatus ippsDotProd_16s16sc64sc(const Ipp16s* pSrc1, const Ipp16sc* pSrc2,
int len, Ipp64sc* pDp);

IppStatus ippsDotProd_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
len, Ipp32f* pDp);

IppStatus ippsDotProd_16sc32fc(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
int len, Ipp32fc* pDp);

IppStatus ippsDotProd_16s16sc32fc(const Ipp16s* pSrc1, const Ipp16sc* pSrc2,
int len, Ipp32fc* pDp);

IppStatus ippsDotProd_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, int
len, Ipp16s* pDp, int scaleFactor);

IppStatus ippsDotProd_16sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc* pSrc2,
int len, Ipp16sc* pDp, int scaleFactor);

IppStatus ippsDotProd_32s_Sfs(const Ipp32s* pSrc1, const Ipp32s* pSrc2, int
len, Ipp32s* pDp, int scaleFactor);

IppStatus ippsDotProd_32sc_Sfs(const Ipp32sc* pSrc1, const Ipp32sc* pSrc2,
int len, Ipp32sc* pDp, int scaleFactor);

IppStatus ippsDotProd_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
int len, Ipp32s* pDp, int scaleFactor);

IppStatus ippsDotProd_16s16sc32sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc*
pSrc2, int len, Ipp32sc* pDp, int scaleFactor);

IppStatus ippsDotProd_16s32s32s_Sfs(const Ipp16s* pSrc1, const Ipp32s* pSrc2,
int len, Ipp32s* pDp, int scaleFactor);

IppStatus ippsDotProd_16s16sc_Sfs(const Ipp16s* pSrc1, const Ipp16sc* pSrc2,
int len, Ipp16sc* pDp, int scaleFactor);

IppStatus ippsDotProd_16sc32sc_Sfs(const Ipp16sc* pSrc1, const Ipp16sc*
pSrc2, int len, Ipp32sc* pDp, int scaleFactor);

```



```
IppStatus ippsDotProd_32s32sc_Sfs(const Ipp32s* pSrc1, const Ipp32sc* pSrc2,
int len, Ipp32sc* pDp, int scaleFactor);
```

## Parameters

<i>pSrc1</i>	Pointer to the first vector to compute the dot product value.
<i>pSrc2</i>	Pointer to the second vector to compute the dot product value.
<i>pDp</i>	Pointer to the output result.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsDotProd` is declared in the `ipps.h` file. This function computes the dot product (scalar value) of two vectors, *pSrc1* and *pSrc2*, and stores the result in *pDp*.

The computation is performed as follows:

$$dp = \sum_{n=0}^{len-1} pSrc1[n] \cdot pSrc2[n]$$

To compute the dot product of complex data, use the function `ippsConj` to conjugate one of the operands. The vectors *pSrc1* and *pSrc2* must be of equal length.

Example 5-41 below shows how to use the function `ippsDotProd_64f` to verify orthogonality of the sine and cosine functions. Two vectors are orthogonal to each other when the dot product of the two vectors is zero.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDp</i> , <i>pSrc1</i> , or <i>pSrc2</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

**Example 5-41 Using the Function ippsDotProd to Verify Orthogonality of Sin and Cos**

```
void dotprod(void) {
    Ipp64f x[10], dp;
    int n;
    for (n = 0; n<10; ++n) x[n] = sin(IPP_2PI * n / 8);
    ippsDotProd_64f(x, x+2, 8, &dp);
    printf_64f("dp =", &dp, 1, ippsStsNoErr);
}
Output:
dp = 0.000000
Matlab* Analog:
>> n = 0:9; x = sin(2*pi*n/8); a = x(1:8); b = x(3:10); a*b'
```

**MaxEvery, MinEvery**

*Computes maximum or minimum value for each pair of elements of two vectors.*

---

**Syntax**

```
IppStatus ippsMaxEvery_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsMaxEvery_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsMaxEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMaxEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMaxEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMinEvery_8u_I(const Ipp8u* pSrc, Ipp8u* pSrcDst, int len);
IppStatus ippsMinEvery_16u_I(const Ipp16u* pSrc, Ipp16u* pSrcDst, int len);
IppStatus ippsMinEvery_16s_I(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len);
IppStatus ippsMinEvery_32s_I(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len);
IppStatus ippsMinEvery_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMinEvery_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, Ipp32u len);
```

## Parameters

<i>pSrc</i>	Pointer to the first input vector.
<i>pSrcDst</i>	Pointer to the second input vector which stores the result.
<i>len</i>	Number of elements in the vector.

## Description

The function `ippsMaxEvery` is declared in the `ipps.h` file. This function computes the maximum between each pair of corresponding elements of two input vectors and stores the result in *pSrcDst*.

The function `ippsMinEvery` is declared in the `ipps.h` file. This function computes minimum values likewise.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pSrcDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## ZeroCrossing

Computes specific zero crossing measure.

### Syntax

```
IppStatus ippsZeroCrossing_16s32f(const Ipp16s* pSrc, Ipp32u len, Ipp32f* pValZC, IppsZCType zCType);
```

```
IppStatus ippsZeroCrossing_32f(const Ipp32f* pSrc, Ipp32u len, Ipp32f* pValZC, IppsZCType zCType);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>len</i>	Number of elements in the vector.
<i>pValZC</i>	Pointer to the output value of the zero crossing measure.

*zcType*

Type of the zero crossing measure, possible values are `ippsZCR`, `ippsZCxor` or `ippsZCC`.

## Description

The function `ippsZeroCrossing` is declared in the `ipps.h` file. This function computes specific zero crossing measure according to the parameter *zcType*. The result of zero crossing measurement is stored in *pValZC*. The calculations are performed in accordance with the formulas below.

If *zcType* = `ippsZCR`, the function uses the same formula as the function `ippsSignChangeRate` does, that is:

$$\sum_{i=1}^{len-1} (x_i \cdot x_{i-1}) < 0$$

If *zcType* = `ippsZCxor`, the function uses the same formula as the function `ippsSignChangeRateXor` does, that is:

$$\sum_{i=1}^{len-1} \text{sign}(x_i) \wedge \text{sign}(x_{i-1}) \quad , \text{ where } \text{sign}(x) = \begin{cases} 0: & x > 0, \quad x = +0 \\ 1: & x < 0, \quad x = -0 \end{cases};$$

If *zcType* = `ippsZCC`, the function uses the same formula as the function `ippsSignChangeRate_Count0` does, that is:

$$\sum_{i=1}^{len-1} \frac{\text{abs}(\text{sign}(x_i) - \text{sign}(x_{i-1}))}{2} \quad , \text{ where } \text{sign}(x) = \begin{cases} 1: & x > 0 \\ 0: & x = 0. \\ -1: & x < 0 \end{cases}$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pValZC</i> pointer is <i>NULL</i> .
<code>ippStsRangeErr</code>	Indicates an error when <i>zcType</i> has an invalid value.

## CountInRange

*Computes the number of elements of the vector whose values are in the specified range.*

---

### Syntax

```
IppStatus ippCountInRange_32s(const Ipp32s* pSrc, int len, int* pCounts,
Ipp32s lowerBound, Ipp32s upperBound);
```

### Parameters

<i>pSrc</i>	Pointer to the first input vector.
<i>pCounts</i>	Pointer to the second input vector which stores the result.
<i>len</i>	Number of elements in the vector.
<i>lowerBound</i>	Lower boundary of the range.
<i>upperBound</i>	Upper boundary of the range.

### Description

The function `ippCountInRange` is declared in the `ipps.h` file. This function computes the number of elements of the vector *pSrc* whose values are in the range  $lowerBound < pSrc[n] < upperBound$ . The total number of such elements are stored in the *pCounts*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pSCounts</i> pointer is <i>NULL</i> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Sampling Functions

The functions described in this section manipulate signal samples. Sampling functions are used to change the sampling rate of the input signal and thus to obtain the signal vector of a required length. The functions perform the following operations:

- Insert zero-valued samples between neighboring samples of a signal (up-sample).
- Remove samples from between neighboring samples of a signal (down-sample).

The upsampling and downsampling functions are used by some filtering functions described in Chapter 6.

### SampleUp

*Up-samples a signal, conceptually increasing its sampling rate by an integer factor.*

---

#### Syntax

```

IppStatus ippsSampleUp_16s (const Ipp16s* pSrc, int srcLen, Ipp16s* pDst,
int* pDstLen, int factor, int* pPhase);

IppStatus ippsSampleUp_32f (const Ipp32f* pSrc, int srcLen, Ipp32f* pDst,
int* pDstLen, int factor, int* pPhase);

IppStatus ippsSampleUp_64f (const Ipp64f* pSrc, int srcLen, Ipp64f* pDst,
int* pDstLen, int factor, int* pPhase);

IppStatus ippsSampleUp_16sc (const Ipp16sc* pSrc, int srcLen, Ipp16sc* pDst,
int* pDstLen, int factor, int* pPhase);

IppStatus ippsSampleUp_32fc (const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst,
int* pDstLen, int factor, int* pPhase);

IppStatus ippsSampleUp_64fc (const Ipp64fc* pSrc, int srcLen, Ipp64fc* pDst,
int* pDstLen, int factor, int* pPhase);

```

#### Parameters

<i>pSrc</i>	Pointer to the source array (the signal to be up-sampled).
<i>srcLen</i>	Number of samples in the source array <i>pSrc</i> .
<i>pDst</i>	Pointer to the destination array.

<i>pDstLen</i>	Pointer to the length of the destination array <i>pDst</i> .
<i>factor</i>	Factor by which the signal is up-sampled. That is, <i>factor</i> - 1 zeros are inserted after each sample of the source array <i>pSrc</i> .
<i>pPhase</i>	Pointer to the <i>input</i> phase value which determines where each sample from <i>pSrc</i> lies within each output block of <i>factor</i> samples in <i>pDst</i> . The value of <i>pPhase</i> is required to be in the range [0; <i>factor</i> -1].

## Description

The function `ippsSampleUp` is declared in the `ipps.h` file. This function up-samples the *srcLen*-length source array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Up-sampling inserts *factor*-1 zeros between each sample of *pSrc*. The *pPhase* argument determines where each sample from the input array lies within each output block of *factor* samples. The value of *pPhase* is required to be in the range [0; *factor*-1].

For example, if the input phase is 0, then every *factor* samples of the destination array begin with the corresponding source array sample, the other *factor*-1 samples are equal to 0. The length of the destination array is stored by the *pDstLen* address.

The *pPhase* value is the phase of an source array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

The `ippsSampleUp` functionality can be described as follows:

$$pDst[factor * n + phase] = pSrc[n], \quad 0 \leq n < srcLen$$

$$pDst[factor * n + m] = 0, \quad 0 \leq n < srcLen, \quad 0 \leq m < factor, \quad m \neq phase$$

$$pDstLen = factor * srcLen .$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pDst</i> , <i>pSrc</i> , <i>pDstLen</i> , or <i>pPhase</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>srcLen</i> is less than or equal to 0.
<code>ippStsSampleFactorErr</code>	Indicates an error if <i>factor</i> is less than or equal to 0.

`ippStsSamplePhaseErr` Indicates an error when `pPhase` is negative, or bigger than or equal to `factor`.

## SampleDown

*Down-samples a signal, conceptually decreasing its sampling rate by an integer factor.*

---

### Syntax

```
IppStatus ippsSampleDown_16s(const Ipp16s* pSrc, int srcLen, Ipp16s* pDst,
int* pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst,
int* pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst,
int* pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_16sc(const Ipp16sc* pSrc, int srcLen, Ipp16sc* pDst,
int* pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst,
int* pDstLen, int factor, int* pPhase);
```

```
IppStatus ippsSampleDown_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc* Dst,
int* pDstLen, int factor, int* pPhase);
```

### Parameters

<code>pSrc</code>	Pointer to the source array holding the samples to be down-sampled.
<code>srcLen</code>	Number of samples in the input array <code>pSrc</code> .
<code>pDst</code>	Pointer to the destination array.
<code>pDstLen</code>	Pointer to the length of the destination array <code>pDst</code> .
<code>factor</code>	Factor by which the signal is down-sampled. That is, <code>factor - 1</code> samples are discarded from every block of <code>factor</code> samples in <code>pSrc</code> .



*pPhase* Pointer to the input phase value that determines which of the samples within each block of *factor* samples from *pSrc* is not discarded and copied to *pDst*. The value of *pPhase* is required to be in the range  $[0; factor-1]$ .

## Description

The function `ippsSampleDown` is declared in the `ipps.h` file. This function down-samples the *srcLen*-length source array *pSrc* by factor *factor* with phase *pPhase*, and stores the result in the array *pDst*, ignoring its length value by the *pDstLen* address.

Down-sampling discards *factor* - 1 samples from *pSrc*, copying one sample from each block of *factor* samples from *pSrc* to *pDst*. The *pPhase* argument determines which of the samples in each block is not discarded and where it lies within each input block of *factor* samples. The value of *pPhase* is required to be in the range  $[0; factor-1]$ . The length of the destination array is stored by the *pDstLen* address.

The *pPhase* value is the phase of an source array sample. It is also a returned output phase which can be used as an input phase for the first sample in the next block to process. Use *pPhase* for block mode processing to get a continuous output signal.

You can use the FIR multi-rate filter to combine filtering and resampling, for example, for antialiasing filtering before the sub-sampling procedure.

The `ippsSampleDown` functionality can be described as follows:

$$pDstLen = (srcLen + factor - 1 - phase) / factor$$

$$pDst[n] = pSrc[factor * n + phase], 0 \leq n < pDstLen$$

$$phase = (factor + phase - srcLen \% factor) \% factor.$$

Example 5-42 below shows how to use the function `ippsSampleDown`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> , <i>pSrc</i> , <i>pDstLen</i> , or <i>pPhase</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>srcLen</i> is less than or equal to 0.
<code>ippStsSampleFactorErr</code>	Indicates an error when <i>factor</i> is less than or equal to 0.

`ippStsSamplePhaseErr` Indicates an error when *pPhase* is negative, or bigger than or equal to *factor.i*

## Example 5-42 Using the `ippsSampleDown` Function

```
void sampling( void ) {  
    Ipp16s x[8] = { 1,2,3,4,5,6,7,8 };  
    Ipp16s y[8] = { 9,10,11,12,13,14,15,16 }, z[8];  
    int dstLen1, dstLen2, phase = 2;  
    IppStatus st = ippsSampleDown_16s(x, 8, z, &dstLen1, 3, &phase);  
    st = ippsSampleDown_16s(y, 8, z+dstLen1, &dstLen2, 3, &phase);  
    printf_16s("down-sampling =", z, dstLen1+dstLen2, st);  
}
```

Output:

```
down-sampling = 3 6 9 12 15
```

# Filtering Functions

This chapter describes the Intel® IPP functions that perform convolution and correlation operations, as well as linear and non-linear filtering. These functions are grouped into the following sections:

## Convolution and Correlation Functions

### Filtering Functions

Each section starts with a table that lists functions described in more detail later in this section, together with the brief description of operations that these functions perform.

## Convolution and Correlation Functions

Convolution is an operation used to define an output signal from any linear time-invariant (LTI) processor in response to any input signal.

The correlation functions described later in this section estimate either the auto-correlation of a source vector or the cross-correlation of two vectors

The full list of functions in this group is given in Table 6-1 below.

**Table 6-1 Intel IPP Convolution and Correlation Functions**

Function Base Name	Operation
<a href="#">Conv</a>	Performs finite, linear convolution of two sequences.
<a href="#">ConvBiased</a>	Computes the specified number of elements of the full finite linear convolution of two vectors.
<a href="#">ConvCyclic</a>	Performs cyclic convolution of two sequences of the fixed size.
<a href="#">AutoCorr</a>	Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.
<a href="#">CrossCorr</a>	Estimates the cross-correlation of two vectors.
<a href="#">UpdateLinear</a>	Integrates an input vector with specified integration weight.
<a href="#">UpdatePower</a>	Integrates the square of an input vector with specified integration weight

## Conv

Performs finite, linear convolution of two vectors.

### Syntax

```
IppStatus ippsConv_32f(const Ipp32f* pSrc1, int src1Len, const Ipp32f* pSrc2,
int src2Len, Ipp32f* pDst);
```

```
IppStatus ippsConv_64f(const Ipp64f* pSrc1, int src1Len, const Ipp64f* pSrc2,
int src2Len, Ipp64f* pDst);
```

```
IppStatus ippsConv_16s_Sfs(const Ipp16s* pSrc1, int src1Len, const Ipp16s*
pSrc2, int src2Len, Ipp16s* pDst, int scaleFactor);
```

### Parameters

<i>pSrc1, pSrc2</i>	Pointers to the source vectors.
<i>src1Len</i>	Number of elements in the vector <i>pSrc1</i> .
<i>src2Len</i>	Number of elements in the vector <i>pSrc2</i> .
<i>pDst</i>	Pointer to the destination vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsConv` is declared in the `ipps.h` file. This function performs finite linear convolution of two sequences. The *src1Len* elements of the vector *pSrc1* is convolved with the *src2Len* elements of the vector *pSrc2* to produce an (*src1Len* + *src2Len* - 1)-length vector *pDst*. The result of the convolution is defined as follows:

$$pDst[n] = \sum_{k=0}^n pSrc1[k] \cdot pSrc2[n-k] \quad 0 \leq n \leq src1Len + src2Len - 1$$

Here  $pSrc1[i] = 0$ , if  $i \geq src1Len$ , and  $pSrc2[j] = 0$ , if  $j \geq src2Len$ .

Example 6-1 below shows the code for the convolution of two vectors using `ippsConv_16s_Sfs` function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>src1Len</code> or <code>src2Len</code> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when there is not enough memory for internal buffers.

## Example 6-1 Using the Function `ippsConv` to Convolve Two Vectors

```

IppStatus convolution(void) {
    Ipp16s x[5] = {-2,0,1,-1,3}, h[2] = {0,1}, y[6];
    IppStatus st = ippsConv_16s_Sfs(x, 5, h, 2, y, 0);
    printf_16s("conv =", y, 6, st);
    return st;
}

```

Output:

```
conv = 0 -2 0 1 -1 3
```

Matlab\* Analog:

```
>> x = [-2,0,1,-1,3]; h = [0,1]; y = conv(x,h)
```

## ConvBiased

*Computes the specified number of elements of the full finite linear convolution of two vectors.*

### Syntax

```
IppStatus ippsConvBiased_32f(const Ipp32f* pSrc1, int src1Len, const Ipp32f*
pSrc2, int src2Len, Ipp32f* pDst, int dstLen, int bias);
```

### Parameters

<code>pSrc1, pSrc2</code>	Pointers to the two vectors to be convolved.
<code>src1Len</code>	Number of elements in the vector <code>pSrc1</code> .
<code>src2Len</code>	Number of elements in the vector <code>pSrc2</code> .

<i>pDst</i>	Pointer to the vector <i>pDst</i> . This vector stores the result of the convolution.
<i>dstLen</i>	Number of elements in the vector <i>pDst</i> .
<i>bias</i>	Parameter that specifies the starting element of the convolution.

## Description

The function `ippsConvBiased` is declared in the `ipps.h` file. This function computes *dstLen* elements of finite linear convolution of two specified vectors *pSrc1* and *pSrc2* starting with an element that is specified by the *bias*. The result is stored in the vector *pDst*.

Example 6-2 below shows how to call the function `ippsConvBiased`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>src1Len</i> or <i>src2Len</i> is less than or equal to 0.

## Example 6-2 Using the Function `ippsConvBiased`

```
void func_convbiased()
{
    Ipp32f pSrc1[5] = {1.1, -2.0, 3.5, 2.2, 0.0};
    Ipp32f pSrc2[4] = {0.0, 0.2, 2.5, -1.0};
    const int len = 10;
    Ipp32f pDst[len];
    int bias = 1;
    ippsZero_32f(pDst, len);
    ippsConvBiased_32f(pSrc1, 5, &pSrc2[1], 3, pDst, len, bias);
}
```

Result:

```
pDst -> 0.2  2.3  -4.3  9.2  5.5  0.0  0.0  0.0  0.0  0.0
```

## ConvCyclic

*Performs cyclic convolution of two sequences of the fixed size.*

---

### Syntax

```

IppStatus ippsConvCyclic8x8_32f(const Ipp32f* x, const Ipp32f* h, Ipp32f*
y);

IppStatus ippsConvCyclic4x4_32f32fc(const Ipp32f* x, const Ipp32fc* h,
Ipp32fc* y);

IppStatus ippsConvCyclic8x8_16s_Sfs(const Ipp16s* x, const Ipp16s* h, Ipp16s*
y, int scaleFactor);

```

### Parameters

<i>x, h</i>	Pointers to the source vectors.
<i>y</i>	Pointer to the destination vector that stores the result of convolution.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsConvCyclic` is declared in the `ipps.h` file. This function performs cyclic convolution of two sequences of the fixed size.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

## AutoCorr

*Estimates normal, biased, and unbiased auto-correlation of a vector and stores the result in a second vector.*

---

### Syntax

```

IppStatus ippsAutoCorr_32f(const Ipp32f* pSrc, int srcLen, Ipp32f* pDst, int
dstLen);

```

```

IppStatus ippsAutoCorr_NormA_32f(const Ipp32f* pSrc, int srcLen, Ipp32f*
pDst, int dstLen);

IppStatus ippsAutoCorr_NormB_32f(const Ipp32f* pSrc, int srcLen, Ipp32f*
pDst, int dstLen);

IppStatus ippsAutoCorr_64f(const Ipp64f* pSrc, int srcLen, Ipp64f* pDst, int
dstLen);

IppStatus ippsAutoCorr_NormA_64f(const Ipp64f* pSrc, int srcLen, Ipp64f*
pDst, int dstLen);

IppStatus ippsAutoCorr_NormB_64f(const Ipp64f* pSrc, int srcLen, Ipp64f*
pDst, int dstLen);

IppStatus ippsAutoCorr_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc* pDst,
int dstLen);

IppStatus ippsAutoCorr_NormA_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc*
pDst, int dstLen);

IppStatus ippsAutoCorr_NormB_32fc(const Ipp32fc* pSrc, int srcLen, Ipp32fc*
pDst, int dstLen);

IppStatus ippsAutoCorr_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc* pDst,
int dstLen);

IppStatus ippsAutoCorr_NormA_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc*
pDst, int dstLen);

IppStatus ippsAutoCorr_NormB_64fc(const Ipp64fc* pSrc, int srcLen, Ipp64fc*
pDst, int dstLen);

IppStatus ippsAutoCorr_16s_Sfs(const Ipp16s* pSrc, int srcLen, Ipp16s* pDst,
int dstLen, int scaleFactor);

IppStatus ippsAutoCorr_NormA_16s_Sfs( const Ipp16s* pSrc, int srcLen, Ipp16s*
pDst, int dstLen, int scaleFactor);

IppStatus ippsAutoCorr_NormB_16s_Sfs(const Ipp16s* pSrc, int srcLen, Ipp16s*
pDst, int dstLen, int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>srcLen</i>	The number of elements in the source vector.



<i>pDst</i>	Pointer to the destination vector, which stores the estimated auto-correlation results of the source vector.
<i>dstLen</i>	The number of elements in the destination vector (length of auto-correlation).
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The `ippsAutoCorr` function is declared in the `ipps.h` file. This function estimates normal auto-correlation of the *srcLen*-length source vector *pSrc* and stores the results in the *dstLen*-length vector *pDst*. Function flavors `ippsAutoCorr_NormA` and `ippsAutoCorr_NormB` compute biased and unbiased auto-correlation of the source vector, respectively. The resulting vector *pDst* is defined by the following equations:

$$pDst[n] = \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \leq n < dstLen \quad (\text{normal})$$

$$pDst[n] = \frac{1}{srcLen} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \leq n < dstLen \quad (\text{biased})$$

$$pDst[n] = \frac{1}{srcLen - n} \sum_{i=0}^{srcLen-1} conj(pSrc[i]) \cdot pSrc[i+n], \quad 0 \leq n < dstLen \quad (\text{unbiased})$$

where

$$pSrc[i] = \begin{cases} pSrc[i], & 0 \leq i < srcLen \\ 0, & \text{otherwise} \end{cases}$$

Example 6-3 below shows how to call the function `ippsAutoCorr_NormB_32f`.



**NOTE.** The auto-correlation estimates are computed only for positive lags, since the auto-correlation for a negative lag value is the complex conjugate of the auto-correlation for the equivalent positive lag.

**See also** [ippCrossCorr](#) that estimates the cross-correlation of two vectors.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>srcLen</code> or <code>dstLen</code> is less than or equal to 0.

## Example 6-3 Using the Function `ippsAutoCorr`

```
void func_autocorr()
{
    Ipp32f pSrc[5] = {0.2, 3.1, 2.0, 1.2, -1.1};
    const int dstLen = 10;
    Ipp32f pDst[dstLen];

    ippsAutoCorr_NormB_32f(pSrc, 5, pDst, dstLen);
}

Result:
pDst -> 3.3 2.0 0.6 -1.6 -0.2 0.0 0.0 0.0 0.0 0.0
```

## CrossCorr

*Estimates the cross-correlation of two vectors.*

### Syntax

```
IppStatus ippsCrossCorr_32f(const Ipp32f* pSrc1, int src1Len, const Ipp32f*
pSrc2, int src2Len, Ipp32f* pDst, int dstLen, int lowLag);

IppStatus ippsCrossCorr_64f(const Ipp64f* pSrc1, int src1Len, const Ipp64f*
pSrc2, int src2Len, Ipp64f* pDst, int dstLen, int lowLag);
```

```

IppStatus ippsCrossCorr_32fc(const Ipp32fc* pSrc1, int src1Len, const Ipp32fc*
pSrc2, int src2Len, Ipp32fc* pDst, int dstLen, int lowLag);

IppStatus ippsCrossCorr_64fc(const Ipp64fc* pSrc1, int src1Len, const Ipp64fc*
pSrc2, int src2Len, Ipp64fc* pDst, int dstLen, int lowLag);

IppStatus ippsCrossCorr_16s_Sfs(const Ipp16s* pSrc1, int src1Len, const
Ipp16s* pSrc2, int src2Len, Ipp16s* pDst, int dstLen, int lowLag, int
scaleFactor);

IppStatus ippsCrossCorr_16s64s(const Ipp16s* pSrc1, int src1Len, const Ipp16s*
pSrc2, int src2Len, Ipp64s* pDst, int dstLen, int lowLag);

```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>src1Len</i>	Number of elements in the vector <i>pSrc1</i> .
<i>pSrc2</i>	Pointer to the second source vector.
<i>src2Len</i>	Number of elements in the vector <i>pSrc2</i> .
<i>pDst</i>	Pointer to the vector which stores the results of the estimated cross-correlation of the vectors <i>pSrc1</i> and <i>pSrc2</i> .
<i>dstLen</i>	Number of elements in the vector <i>pDst</i> , which determines the range of lags at which the correlation estimates are computed.
<i>lowLag</i>	Lower value of the range of lags at which the correlation estimates are computed.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsCrossCorr` is declared in the `ipps.h` file. This function estimates cross-correlation of the *src1Len* elements of the vector *pSrc1* and the *src2Len* elements of the vector *pSrc2*, and stores the results in the vector *pDst*. The resulting vector *pDst* is defined by the equation:

$$pDst[n] = \sum_{i=0}^{len1-1} conj(pSrc1[i]) \cdot pSrc2[n+i+lowLag] ,$$

,

where  $0 \leq n < dstLen$ , and

$$pSrc2[j] = \begin{cases} pSrc2[j], & 0 < j < len2 \\ 0, & otherwise \end{cases}$$

Example 6-4 below shows how to use the function `ippsCrossCorr`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> or <code>pSrc2</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>src1Len</code> or <code>src2Len</code> is less than or equal to 0.

### Example 6-4 Using the Function `ippsCrossCorr`

```
void crossCorr(void) {
    #undef LEN
    #define LEN 11
    Ipp32f win[LEN], y[LEN];
    IppStatus st;
    ippsSet_32f (1, win, LEN);
    ippsWinHamming_32f_I (win, LEN);
    st = ippsCrossCorr_32f (win, LEN, win, LEN, y, LEN, -(LEN-1));
    printf_32f("cross corr =", y,7,st);
}
```

Output:

```
cross corr = 0.006400 0.026856 0.091831 0.242704 0.533230
1.009000 1.672774
```

Matlab\* analog:

```
>> x = hamming(11)'; y = xcorr(x,x); y(1:7)
```

## UpdateLinear

*Integrates an input vector with specified integration weight.*

---

### Syntax

```
IppStatus ippsUpdateLinear_16s32s_I(const Ipp16s* pSrc, int len, Ipp32s*
pSrcDst, int srcShiftRight, Ipp16s alpha, IppHintAlgorithm hint);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>len</i>	Number of elements of the vector.
<i>pSrcDst</i>	Pointer to the input value and output result.
<i>srcShiftRight</i>	Shift value; must be non-negative.
<i>alpha</i>	Integration weight.

*hint*                      Suggests using specific code. The possible values for the *hint* argument are described in [Hint Arguments](#).

## Description

The function `ippsUpdateLinear` is declared in the `ipps.h` file. This function performs *len* iterations in which the sum

$\alpha * pSrcDst + (1 - \alpha) * pSrc[i]_{shift}$  is calculated and stored in *pSrcDst*. Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and *pSrc[i]\_shift* is the element of the source vector right-shifted by the non-negative value *srcShiftRight*.

Example 6-5 below shows how to use the function *ippsUpdateLinear*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 6-5 Using the Function `ippsUpdateLinear`

```
void func_updatelinear()
{
    Ipp16s pSrc[2] = {16, 32};
    Ipp32s pSrcDst =1;
    Ipp16s alpha = 2;
    int shift = 2;
    int srcLen = 2;

    ippsUpdateLinear_16s32s_I(pSrc, 2, &pSrcDst, shift, alpha, ippAlgHintFast);
}

Result: -12
```

## UpdatePower

*Integrates the square of an input vector with specified integration weight.*

---

### Syntax

```
IppStatus ippsUpdatePower_16s32s_I(const Ipp16s* pSrc, int len, Ipp32s* pSrcDst, int srcShiftRight, Ipp16s alpha, IppHintAlgorithm hint);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>len</i>	Number of elements of the vector.
<i>pSrcDst</i>	Pointer to input and output
<i>srcShiftRight</i>	Shift value.
<i>alpha</i>	Integration weight.
<i>hint</i>	Suggests using specific code. The possible values for the <i>hint</i> argument are described in <a href="#">Hint Arguments</a> .

### Description

The function `ippsUpdatePower` is declared in the `ipps.h` file. This function performs *len* iterations in which the sum  $\alpha * pSrcDst + (1 - \alpha) * (pSrc[i] * pSrc[i])_{shift}$  is calculated and stored in *pSrcDst*. Here *i* is the number of previous iterations, *pSrcDst* is the result of previous iteration, and  $(pSrc[i] * pSrc[i])_{shift}$  is the squared *i*-th element right-shifted by the non-negative value *srcShiftRight*.

Example 6-6 below shows how to use the function `ippsUpdatePower`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

### Example 6-6 Using the Function `ippsUpdatePower`

```
void func_updatepower()
{
    Ipp16s pSrc[2] = {16, 32};
    Ipp32s pSrcDst = 1;
    Ipp16s alpha = 3;
    int shift = 1;
    int srcLen = 2;

    ippsUpdatePower_16s32s_I(pSrc, 2, &pSrcDst, shift, alpha, ippAlgHintFast);
}

Result:  -1783
```

## Filtering Functions

The Intel IPP functions described in this section implement the following types of filters:

- finite impulse response (FIR) filter
- adaptive finite impulse response using least mean squares (LMS) filter
- infinite impulse response (IIR) filter
- median filter.

A special set of functions is designed to generate filter coefficients for different types of FIR filters.

The full list of filtering functions is given in Table 6-2 below.

**Table 6-2 Intel IPP Filtering Functions**

Function Base Name	Operation
<code>SumWindow</code>	Sums elements in the mask applied to each element of the vector.
FIR Filter Functions	
<code>FIRInitAlloc</code>	Allocates memory and initializes a single-rate FIR filter state structure.
<code>FIRStreamInitAlloc</code>	Allocates memory and initializes a single-rate stream FIR filter state structure.
<code>FIRMRInitAlloc</code>	Allocates memory and initializes a multi-rate FIR filter state structure.



Function Base Name	Operation
<a href="#">FIRMRStreamInitAlloc</a>	Allocates memory and initializes a multi-rate stream FIR filter state structure.
<a href="#">FIRFree</a>	Frees memory allocated for the FIR filter state structure.
<a href="#">FIRInit</a>	Initializes a single-rate FIR filter state structure.
<a href="#">FIRStreamInit</a>	Initializes a single-rate stream FIR filter state structure.
<a href="#">FIRMRInit</a>	Initializes a multi-rate FIR filter state structure.
<a href="#">FIRMRStreamInit</a>	Initializes a multi-rate stream FIR filter state structure.
<a href="#">FIRGetStateSize</a> , <a href="#">FIRMRGetStateSize</a>	Calculates the size of an external buffer for the FIR filter structure.
<a href="#">FIRStreamGetStateSize</a> , <a href="#">FIRMRStreamGetStateSize</a>	Calculates the size of an external buffer for the stream FIR filter structure.
<a href="#">FIRGetTaps</a>	Retrieves the tap values from the FIR filter state structure.
<a href="#">FIRSetTaps</a>	Sets the taps values in the FIR filter state structure.
<a href="#">FIRGetDlyLine</a>	Retrieves the delay line contents from the FIR filter state structure.
<a href="#">FIRSetDlyLine</a>	Sets the delay line contents in the FIR filter state structure.
<a href="#">FIROne</a>	Filters a single sample through a FIR filter.
<a href="#">FIR</a>	Filters a block of samples through a FIR filter.
Direct versions	
<a href="#">FIROne_Direct</a>	Directly filters a single sample through a FIR filter.
<a href="#">FIR_Direct</a>	Directly filters a block of samples through a single-rate FIR filter.
<a href="#">FIRMR_Direct</a>	Directly filters a block of samples through a multi-rate FIR filter.
Sparse FIR Filter	
<a href="#">FIRSparseInit</a>	Initializes a sparse FIR filter structure.
<a href="#">FIRSparseGetStateSize</a>	Computes the size of the external buffer for the sparse FIR filter structure.
<a href="#">FIRSparse</a>	Filters a block of samples through a sparse FIR filter.
FIR Filter Coefficient Generating Functions	
<a href="#">FIRGenLowpass</a>	Computes the lowpass FIR filter coefficients.
<a href="#">FIRGenHighpass</a>	Computes the highpass FIR filter coefficients.
<a href="#">FIRGenBandpass</a>	Computes the bandpass FIR filter coefficients.
<a href="#">FIRGenBandstop</a>	Computes the bandstop FIR filter coefficients.
Single-Rate FIR LMS Filter Functions	

Function Base Name	Operation
<a href="#">FIRLMSInitAlloc</a>	Allocates memory and initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.
<a href="#">FIRLMSFree</a>	Closes an adaptive FIR filter that uses the LMS algorithm.
<a href="#">FIRLMSGetTaps</a>	Gets the taps of a FIR LMS filter.
<a href="#">FIRLMSGetDlyLine</a>	Retrieves the delay line contents from the FIR LMS filter.
<a href="#">FIRLMSSetDlyLine</a>	Sets the delay line contents in the FIR LMS filter.
<a href="#">FIRLMS</a>	Filters an array through a FIR LMS filter.
Direct versions	
<a href="#">FIRLMSOne_Direct</a>	Filters a single sample through a FIR LMS filter.
Multi-Rate FIR LMS Filter Functions	
<a href="#">FIRLMSMRInitAlloc</a>	Allocates memory and initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.
<a href="#">FIRLMSMRFree</a>	Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.
<a href="#">FIRLMSMRSetMu</a>	Sets the adaptation step.
<a href="#">FIRLMSMRUpdateTaps</a>	Updates the filter coefficients using the adaptation error value.
<a href="#">FIRLMSMRGetTaps</a>	Retrieves tap values in the multi-rate FIR LMS filter.
<a href="#">FIRLMSMRSetTaps</a>	Sets tap values in the multi-rate FIR LMS filter.
<a href="#">FIRLMSMRGetTapsPointer</a>	Returns the pointer to the filter coefficients.
<a href="#">FIRLMSMRGetDlyLine</a>	Retrieves the delay line contents from the multi-rate FIR LMS filter state.
<a href="#">FIRLMSMRSetDlyLine</a>	Sets the delay line contents in the multi-rate FIR LMS filter state.
<a href="#">FIRLMSMRGetDlyVal</a>	Gets one delay line values from the specified position.
<a href="#">FIRLMSMRPutVal</a>	Places the input value in the delay line.
<a href="#">FIRLMSMROne</a>	Filters data placed in the delay line.
<a href="#">FIRLMSMROneVal</a>	Filters one input value.
IIR Filter Functions	
<a href="#">IIRGenLowpass, IIRGenHighpass</a>	Computes lowpass and highpass IIR filter coefficients.
<a href="#">IIRInitAlloc</a>	Allocates memory and initializes an arbitrary IIR filter state.
<a href="#">IIRInitAlloc_BiQuad</a>	Allocates memory and initializes a biquad IIR filter state.
<a href="#">IIRFree</a>	Closes an IIR filter state.

Function Base Name	Operation
<a href="#">IIRInit</a>	Initializes an arbitrary IIR filter state.
<a href="#">IIRInit_BiQuad</a>	Initializes a biquad IIR filter state
<a href="#">IIRGetStateSize</a>	Computes the length of the external buffer for the arbitrary IIR filter state structure.
<a href="#">IIRGetStateSize_BiQuad</a>	Computes the length of the external buffer for the biquad IIR filter state structure.
<a href="#">IIRSetTaps</a>	Sets the taps in an IIR filter state
<a href="#">IIRGetDlyLine</a>	Retrieves the delay line values from the IIR filter state.
<a href="#">IIRSetDlyLine</a>	Sets the delay line values in the IIR filter state.
<a href="#">IIROne</a>	Filters a single sample through an IIR filter.
<a href="#">IIR</a>	Filters a block of samples through an IIR filter.
Direct versions	
<a href="#">IIROne_Direct</a>	Directly filters a single sample through the arbitrary IIR filter.
<a href="#">IIROne_BiQuadDirect</a>	Directly filters a single sample through the biquad IIR filter.
<a href="#">IIR_Direct</a>	Directly filters a block of samples through the arbitrary IIR filter.
<a href="#">IIR_BiQuadDirect</a>	Directly filters a block of samples through the biquad IIR filter
Sparse IIR Filter	
<a href="#">IIRSparseInit</a>	Initializes a sparse IIR filter structure.
<a href="#">IIRSparseGetStateSize</a>	Computes the size of the external buffer for the sparse IIR filter structure.
<a href="#">IIRSparse</a>	Filters a block of samples through a sparse IIR filter.
Median Filter Functions	
<a href="#">FilterMedian</a>	Computes median values for each source vector element.

## Sum Window

*Sums elements in the mask applied to each element of a vector.*

### Syntax

```
IppStatus ippsSumWindow_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len, int maskSize);
```

```
IppStatus ippsSumWindow_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len,
int maskSize);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements of the vector.
<i>maskSize</i>	Size of the mask.

## Description

The function `ippsSumWindow` is declared in the `ipps.h` file. This function sets each element in the destination vector *pDst* as the sum of *maskSize* elements of the source vector *pSrc*. The computation is performed as follows:

$$pDst[n] = \sum_{k=n}^{maskSize} pSrc[k], 0 \leq n < len$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsMaskSizeErr</code>	Indicates an error when <i>maskSize</i> is less than or equal to 0.

## FIR Filter Functions

The functions described in this section perform a finite impulse response (FIR) filtering of input data. The functions initialize different FIR filter structures, get and set the delay lines and filter coefficients (taps), and perform filtering. Intel IPP contains the functions that implement the FIR filters without the delay line - stream FIR filters.

To use the FIR filter functions, follow this general scheme:

1. Call either `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc` to allocate memory and initialize the taps and the delay line in the filter state structure of a single-rate or multi-rate filter, respectively. Or call either `ippsFIRInit` or `ippsFIRMRInit` to initialize the taps and the

delay line in the corresponding filter state structure in the previously created external buffer. The size of this buffer must be computed beforehand by calling the functions `ippsFIRGetStateSize` or `ippsFIRMRGetStateSize`, respectively.

2. Call `ippsFIROne` to filter a single sample through a single-rate filter, `ippsFIR` to filter a block of consecutive samples through a single-rate or multi-rate filter.
3. To set new taps and delay line values in the previously initialized filter state, call the functions `ippsFIRSetTaps` and `ippsFIRSetDlyLine`. To get taps and delay line values of the initialized filter state, call the functions `ippsFIRGetTaps` and `ippsFIRGetDlyLine`.
4. Call `ippsFIRFree` to free dynamic memory associated with the FIR filter state structure created by `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

Alternatively, you may use the direct version of the functions `ippsFIROne_Direct`, `ippsFIR_Direct`, `ippsFIRMR_Direct`. These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

Special set of functions allows to compute the filter coefficients for different filters.

## FIRInitAlloc

*Allocates memory and initializes a single-rate FIR filter state.*

### Syntax

#### Case 1: Operation on integer samples

```
IppStatus ippsFIRInitAlloc_16s(IppsFIRState_16s** ppState, const Ipp16s*
pTaps, int tapsLen, int tapsFactor, const Ipp16s* pDlyLine);

IppStatus ippsFIRInitAlloc32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
pTaps, int tapsLen, int tapsFactor, const Ipp16s* pDlyLine);

IppStatus ippsFIRInitAlloc32s_16s32f(IppsFIRState32s_16s** ppState, const
Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine);

IppStatus ippsFIRInitAlloc32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
pTaps, int tapsLen, const Ipp16s* pDlyLine);

IppStatus ippsFIRInitAlloc64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
pTaps, int tapsLen, const Ipp16s* pDlyLine);

IppStatus ippsFIRInitAlloc_32s(IppsFIRState_32s** ppState, const Ipp32s*
pTaps, int tapsLen, const Ipp32s* pDlyLine);

IppStatus ippsFIRInitAlloc64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
pTaps, int tapsLen, const Ipp32s* pDlyLine);
```

```

IppStatus ippsFIRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** ppState, const
Ipp32sc* pTaps, int tapsLen, int tapsFactor, const Ipp16sc* pDlyLine);

IppStatus ippsFIRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** ppState,
const Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);

IppStatus ippsFIRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** ppState, const
Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);

IppStatus ippsFIRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** ppState, const
Ipp64fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine);

IppStatus ippsFIRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** ppState, const
Ipp64fc* pTaps, int tapsLen, const Ipp32sc* pDlyLine);

```

## Case 2: Operation on floating point samples

```

IppStatus ippsFIRInitAlloc_32f(IppsFIRState_32f** ppState, const Ipp32f*
pTaps, int tapsLen, const Ipp32f* pDlyLine);

IppStatus ippsFIRInitAlloc64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
pTaps, int tapsLen, const Ipp32f* pDlyLine);

IppStatus ippsFIRInitAlloc_64f(IppsFIRState_64f** ppState, const Ipp64f*
pTaps, int tapsLen, const Ipp64f* pDlyLine);

IppStatus ippsFIRInitAlloc_32fc(IppsFIRState_32fc** ppState, const Ipp32fc*
pTaps, int tapsLen, const Ipp32fc* pDlyLine);

IppStatus ippsFIRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** ppState, const
Ipp64fc* pTaps, int tapsLen, const Ipp32fc* pDlyLine);

IppStatus ippsFIRInitAlloc_64fc(IppsFIRState_64fc** ppState, const Ipp64fc*
pTaps, int tapsLen, const Ipp64fc* pDlyLine);

```

## Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <i>Ipp32s</i> data type (for integer versions only).
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>tapsLen</i> .

*ppState* Pointer to the pointer to the FIR state structure to be created.

### Description

The function `ippsFIRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a single-rate FIR filter state. The initialization function copies the tap values from the *tapsLen*-length array *pTaps* into the state structure *ppState*. To scale integer taps use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the *tapsLen*-length array *pDlyLine* is not NULL, the array content is copied into the state structure *ppState*, otherwise the delay line values in the state structure are initialized to 0.

Note that the delay line length is different than that for direct FIR filters (where this length is doubled).

If the state is not created, the initialization function returns an error status.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pTaps</i> or <i>ppState</i> is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRStreamInitAlloc

*Allocates memory and initializes a state structure for the single-rate stream FIR filter.*

---

### Syntax

```
IppStatus ippsFIRStreamInitAlloc_16s(IppsFIRState_16s** ppState, const Ipp16s* pTaps, int tapsLen, int tapsFactor, IppRoundMode rndMode);
```

```
IppStatus ippsFIRStreamInitAlloc_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps, int tapsLen);
```

### Parameters

<i>pTaps</i>	Pointer to the array containing the tap values.
<i>tapsLen</i>	Number of elements in the array <i>pTaps</i> .

<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer version only).
<i>rndMode</i>	Rounding mode, the following values are possible: <code>ippRndZero</code> Specifies that floating-point values must be truncated toward zero. <code>ippRndNear</code> Specifies that floating-point values must be rounded to the nearest even integer. <code>ippRndFinancial</code> Specifies that floating-point values must be rounded down to the nearest integer if decimal value is less than 0.5, or rounded up to the nearest integer if decimal value is equal or greater than 0.5.
<i>ppState</i>	Double pointer to the FIR state structure to be created.

## Description

The function `ippsFIRStreamInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a state structure for a single-rate stream FIR filter. The initialization function copies the tap values from the *tapsLen*-length array *pTaps* into the state structure *ppState*. To scale integer taps use the *tapsFactor* value.

The parameter *rndMode* sets the rounding mode that is used by the functions `ippsFIR` and `ippsFIROne`.

If the state is not created, the initialization function returns an error status.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pTaps</i> or <i>ppState</i> is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.
<code>ippStsRndModNotSupportedErr</code>	Indicates an error when the <i>rndMode</i> has an illegal value.



## FIRMRInitAlloc

*Allocates memory and initializes a multi-rate FIR filter state.*

---

### Syntax

#### Case 1: Operation on integer samples

```
IppStatus ippsFIRMRInitAlloc_16s(IppsFIRState_16s** ppState, const Ipp16s*
pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int downFactor,
int downPhase, const Ipp16s* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32s_16s(IppsFIRState32s_16s** ppState, const
Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
downFactor, int downPhase, const Ipp16s* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32s_16s32f(IppsFIRState32s_16s** ppState, const
Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16s* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32f_16s(IppsFIRState32f_16s** ppState, const
Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16s* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64f_16s(IppsFIRState64f_16s** ppState, const
Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16s* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64f_32s(IppsFIRState64f_32s** ppState, const
Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp32s* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32sc_16sc(IppsFIRState32sc_16sc** ppState, const
Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
downFactor, int downPhase, const Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32sc_16sc32fc(IppsFIRState32sc_16sc** ppState,
const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
int downPhase, const Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc32fc_16sc(IppsFIRState32fc_16sc** ppState, const
Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64fc_16sc(IppsFIRState64fc_16sc** ppState, const
Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16sc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64fc_32sc(IppsFIRState64fc_32sc** ppState, const
Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp32sc* pDlyLine);
```

## Case 2: Operation on floating point samples

```
IppStatus ippsFIRMRInitAlloc_32f(IppsFIRState_32f** ppState, const Ipp32f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64f_32f(IppsFIRState64f_32f** ppState, const
Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc_64f(IppsFIRState_64f** ppState, const Ipp64f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp64f* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc_32fc(IppsFIRState_32fc** ppState, const Ipp32fc*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc64fc_32fc(IppsFIRState64fc_32fc** ppState, const
Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMRInitAlloc_64fc(IppsFIRState_64fc** ppState, const Ipp64fc*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp64fc* pDlyLine);
```

## Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <i>Ipp32s</i> data type (for integer versions only).
<i>downFactor</i>	Downsampling factor.

<i>downPhase</i>	Phase for downsampled signal.
<i>upFactor</i>	Upsampling factor.
<i>upPhase</i>	Phase for upsampled signal.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array $(\text{tapsLen} + \text{upFactor} - 1) / \text{upFactor}$ .
<i>ppState</i>	Pointer to the pointer to the FIR state structure to be created.

## Description

The function `ippsFIRMRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a multi-rate FIR filter state. The initialization functions copy the taps from the `tapsLen`-length array `pTaps` into the state structure `ppState`. To scale integer taps use the `tapsFactor` value. The array `pDlyLine` specifies the delay line values. If the pointer to the array `pDlyLine` is not `NULL`, the array content is copied into the state structure `ppState`, otherwise the delay line values in the state structure are initialized to 0.

If the state is not created, the initialization function returns an error status.

The function initializes the state structure `ppState` of a multi-rate filter; that is, a filter that internally upsamples and/or downsamples using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The parameter `upFactor` is the factor by which the filtered signal is internally upsampled (see description of the function [ippsSampleUp](#) for more details). That is, `upFactor`-1 zeros are inserted between each sample of the input signal.

The parameter `upPhase` is the parameter which determines where a non-zero sample lies within the `upFactor`-length block of upsampled input signal.

The parameter `downFactor` is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the function [ippsSampleDown](#) for more details). That is, `downFactor`-1 output samples are discarded from each `downFactor`-length output block of the upsampled filter response.

The parameter `downPhase` is the parameter which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array `pDlyLine` is non-`NULL`, its length is defined as  $(\text{tapsLen} + \text{upFactor} - 1) / \text{upFactor}$ .

Code [example 6-8](#) shows how to use the function `ippsFIRMRInitAlloc_32f`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pTaps</i> or <i>ppState</i> is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error when <i>upFactor</i> ( <i>downFactor</i> ) is less than or equal to 0.
<code>ippStsFIRMRPhaseErr</code>	Indicates an error when <i>upPhase</i> ( <i>downPhase</i> ) is negative, or greater than or equal to <i>upFactor</i> ( <i>downFactor</i> ).
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRMRStreamInitAlloc

*Allocates memory and initializes a state structure for a multi-rate stream FIR filter.*

---

### Syntax

```
IppStatus ippsFIRMRStreamInitAlloc_16s(IppsFIRState_16s** ppState, const
Ipp16s* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
downFactor, int downPhase, IppRoundMode rndMode);
```

```
IppStatus ippsFIRMRStreamInitAlloc_32f(IppsFIRState_32f** ppState, const
Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase);
```

### Parameters

<i>pTaps</i>	Pointer to the array containing the tap values.
<i>tapsLen</i>	Number of elements in the array <i>pTaps</i> .
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer version only).
<i>downFactor</i>	Downsampling factor.
<i>downPhase</i>	Phase for downsampled signal.
<i>upFactor</i>	Upsampling factor.

<i>upPhase</i>	Phase for upsampled signal.
<i>rndMode</i>	Rounding mode, the following values are possible: <i>ippRndZero</i> Specifies that floating-point values must be truncated toward zero. <i>ippRndNear</i> Specifies that floating-point values must be rounded to the nearest even integer. <i>ippRndFinancial</i> Specifies that floating-point values must be rounded down to the nearest integer if decimal value is less than 0.5, or rounded up to the nearest integer if decimal value is equal or greater than 0.5.
<i>ppState</i>	Double pointer to the stream FIR filter state structure.

## Description

The function `ippsFIRMRStreamInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a state structure for a multi-rate stream FIR filter. The initialization function copies the tap values from the *tapsLen*-length array *pTaps* into the state structure *ppState*. To scale integer taps use the *tapsFactor* value.

If the state is not created, the initialization function returns an error status.

The function initializes the state structure *ppState* of a multi-rate filter; that is, a filter that internally upsamples and/or downsamples using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The parameter *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the `ip psSampleUp` function for more details). That is, *upFactor*-1 zeros are inserted between each sample of input signal.

The parameter *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The parameter *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see description of the `ipp sSampleDown` function for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The parameter *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response.

The parameter *rndMode* sets the rounding mode that is used by the functions `ippsFIR` and `ippsFIROne`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pTaps</i> or <i>ppState</i> is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error when <i>upFactor</i> ( <i>downFactor</i> ) is less than or equal to 0.
<code>ippStsFIRMRPhaseErr</code>	Indicates an error when <i>upPhase</i> ( <i>downPhase</i> ) is negative, or greater than or equal to <i>upFactor</i> ( <i>downFactor</i> ).
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.
<code>ippStsRndModeNotSupportedErr</code>	Indicates an error when the <i>rndMode</i> has an illegal value.

## FIRFree

*Closes a FIR filter state.*

---

### Syntax

```

IppStatus ippsFIRFree_16s(IppsFIRState_16s* pState);
IppStatus ippsFIRFree32s_16s(IppsFIRState32s_16s* pState);
IppStatus ippsFIRFree32f_16s(IppsFIRState32f_16s* pState);
IppStatus ippsFIRFree64f_16s(IppsFIRState64f_16s* pState);
IppStatus ippsFIRFree_32s(IppsFIRState_32s* pState);
IppStatus ippsFIRFree64f_32s(IppsFIRState64f_32s* pState);
IppStatus ippsFIRFree32sc_16sc(IppsFIRState32sc_16sc* pState);
IppStatus ippsFIRFree32fc_16sc(IppsFIRState32fc_16sc* pState);
IppStatus ippsFIRFree64fc_16sc(IppsFIRState64fc_16sc* pState);
IppStatus ippsFIRFree64fc_32sc(IppsFIRState64fc_32sc* pState);
IppStatus ippsFIRFree_32f(IppsFIRState_32f* pState);

```

```

IppStatus ippsFIRFree64f_32f(IppsFIRState64f_32f* pState);
IppStatus ippsFIRFree_64f(IppsFIRState_64f* pState);
IppStatus ippsFIRFree_32fc(IppsFIRState_32fc* pState);
IppStatus ippsFIRFree64fc_32fc(IppsFIRState64fc_32fc* pState);
IppStatus ippsFIRFree_64fc(IppsFIRState_64fc* pState);

```

## Parameters

*pState*                      Pointer to the FIR filter state structure to be closed.

## Description

The function `ippsFIRFree` is declared in the `ipps.h` file. This function closes the FIR filter state by freeing all memory associated with the filter state structure created by `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`. Call `ippsFIRFree` after filtering is completed.

## Return Values

`ippStsNoErr`                Indicates no error.  
`ippStsNullPtrErr`        Indicates an error when one of the specified pointers is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIRInit

*Initializes a single-rate FIR filter state.*

### Syntax

#### Case 1: Operation on integer samples

```

IppStatus ippsFIRInit_16s(IppsFIRState_16s** ppState, const Ipp16s* pTaps,
int tapsLen, int tapsFactor, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
pTaps, int tapsLen, int tapsFactor, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32s_16s32f(IppsFIRState32s_16s** ppState, const Ipp32f*
pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

```

```

IppStatus ippsFIRInit64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
pTaps, int tapsLen, const Ipp16s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_32s(IppsFIRState_32s** ppState, const Ipp32s* pTaps,
int tapsLen, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
pTaps, int tapsLen, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32sc_16sc(IppsFIRState32sc_16sc** ppState, const Ipp32sc*
pTaps, int tapsLen, int tapsFactor, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32sc_16sc32fc(IppsFIRState32sc_16sc** ppState, const
Ipp32fc* pTaps, int tapsLen, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit32fc_16sc(IppsFIRState32fc_16sc** ppState, const Ipp32fc*
pTaps, int tapsLen, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit64fc_16sc(IppsFIRState64fc_16sc** ppState, const Ipp64fc*
pTaps, int tapsLen, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit64fc_32sc(IppsFIRState64fc_32sc** ppState, const Ipp64fc*
pTaps, int tapsLen, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);

```

## Case 2: Operation on floating point samples

```

IppStatus ippsFIRInit_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps,
int tapsLen, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
pTaps, int tapsLen, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_64f(IppsFIRState_64f** ppState, const Ipp64f* pTaps,
int tapsLen, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_32fc(IppsFIRState_32fc** ppState, const Ipp32fc* pTaps,
int tapsLen, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit64fc_32fc(IppsFIRState64fc_32fc** ppState, const Ipp64fc*
pTaps, int tapsLen, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsFIRInit_64fc(IppsFIRState_64fc** ppState, const Ipp64fc* pTaps,
int tapsLen, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

```



## Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer versions only).
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>tapsLen</i> .
<i>ppState</i>	Pointer to the pointer to the FIR state structure to be created.
<i>pBuffer</i>	Pointer to the external buffer for FIR state structure.

## Description

The function `ippsFIRInit` is declared in the `ipps.h` file. This function initializes a single-rate FIR filter state structure in the external buffer. The size of this buffer must be computed previously by calling the functions `ippsFIRGetStateSize`. The initialization function copies the taps from the *tapsLen*-length array *pTaps* into the state structure *ppState* of a single-rate filter. To scale integer taps, use the *tapsFactor* value. The *tapsLen*-length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array contents is copied into the state structure *ppState*, otherwise the delay line values in the state structure are initialized to 0.

Note that the delay line length is different than that for direct FIR filters (where this length is doubled).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.

## FIRStreamInit

*Initializes a single-rate stream FIR filter state.*

---

### Syntax

```

IppStatus ippsFIRStreamInit_16s(IppsFIRState_16s** ppState, const Ipp16s*
pTaps, int tapsLen, int tapsFactor, IppRoundMode rndMode, Ipp8u* pBuffer);

IppStatus ippsFIRStreamInit_32f(IppsFIRState_32f** ppState, const Ipp32f*
pTaps, int tapsLen, Ipp8u* pBuffer);

```

### Parameters

<i>pTaps</i>	Pointer to the array containing the tap values.
<i>tapsLen</i>	Number of elements in the array <i>pTaps</i> .
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer version only).
<i>rndMode</i>	Rounding mode, the following values are possible: <code>ippRndZero</code> Specifies that floating-point values must be truncated toward zero. <code>ippRndNear</code> Specifies that floating-point values must be rounded to the nearest even integer. <code>ippRndFinancial</code> Specifies that floating-point values must be rounded down to the nearest integer if decimal value is less than 0.5, or rounded up to the nearest integer if decimal value is equal or greater than 0.5.
<i>ppState</i>	Double pointer to the FIR state structure to be created.
<i>pBuffer</i>	Pointer to the external buffer for FIR state structure.

### Description

The function `ippsFIRStreamInit` is declared in the `ipps.h` file. This function initializes a single-rate FIR filter state structure in the external buffer . The size of this buffer must be computed by calling the function [ippsFIRStreamGetStateSize](#) beforehand. The initialization function copies the taps from the *tapsLen*-length array *pTaps* into the state structure *ppState* of a single-rate filter. To scale integer taps, use the *tapsFactor* value.

The parameter *rndMode* sets the rounding mode that is used by the functions `ippsFIR` and `ippsFIROne`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsRndModNotSupported</code>	Indicates an error when the <i>rndMode</i> has an illegal value.

## FIRMRInit

*Initializes a multi-rate FIR filter state.*

### Syntax

#### Case 1: Operation on integer samples

```
IppStatus ippsFIRMRInit_16s(IppsFIRState_16s** ppState, const Ipp16s* pTaps,
int tapsLen, int tapsFactor, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit32s_16s(IppsFIRState32s_16s** ppState, const Ipp32s*
pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int downFactor,
int downPhase, const Ipp16s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit32s_16s32f(IppsFIRState32s_16s** ppState, const Ipp32f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp16s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit32f_16s(IppsFIRState32f_16s** ppState, const Ipp32f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp16s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit64f_16s(IppsFIRState64f_16s** ppState, const Ipp64f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp16s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit64f_32s(IppsFIRState64f_32s** ppState, const Ipp64f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp32s* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit32sc_16sc(IppsFIRState32sc_16sc** ppState, const
Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
downFactor, int downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit32sc_16sc32fc(IppsFIRState32sc_16sc** ppState, const
Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit32fc_16sc(IppsFIRState32fc_16sc** ppState, const
Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit64fc_16sc(IppsFIRState64fc_16sc** ppState, const
Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp16sc* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit64fc_32sc(IppsFIRState64fc_32sc** ppState, const
Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);
```

### **Case 2: Operation on floating point samples**

```
IppStatus ippsFIRMRInit_32f(IppsFIRState_32f** ppState, const Ipp32f* pTaps,
int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase, const
Ipp32f* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit64f_32f(IppsFIRState64f_32f** ppState, const Ipp64f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp32f* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit_64f(IppsFIRState_64f** ppState, const Ipp64f* pTaps,
int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase, const
Ipp64f* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit_32fc(IppsFIRState_32fc** ppState, const Ipp32fc*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit64fc_32fc(IppsFIRState64fc_32fc** ppState, const
Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRInit_64fc(IppsFIRState_64fc** ppState, const Ipp64fc*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

## Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer versions only).
<i>downFactor</i>	Downsampling factor.
<i>downPhase</i>	Phase for downsampled signal.
<i>upFactor</i>	Upsampling factor.
<i>upPhase</i>	Phase for upsampled signal.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $(tapsLen + upFactor - 1) / upFactor$ .
<i>ppState</i>	Pointer to the pointer to the FIR state structure.
<i>pBuffer</i>	Pointer to the external buffer for FIR state structure.

## Description

The function `ippsFIRMRInit` is declared in the `ipps.h` file. This function initializes a multi-rate FIR filter state structure in the external buffer. The size of this buffer must be computed previously by calling the functions `ippsFIRMRGetStateSize`. The initialization functions copy the taps from the *tapsLen*-length array *pTaps* into the state structure *ppState*. To scale integer taps, use the *tapsFactor* value. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array contents is copied into the state structure *ppState*, otherwise the delay line values in the state structure are initialized to 0.

The function initializes the state structure *ppState* of a multi-rate filter, that is, a filter that internally upsamples and/or downsamples signals using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The parameter *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the function `ippsSampleUp` for more details). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The parameter *upPhase* is the parameter, which determines where a non-zero sample lies within the *upFactor*-length block of the upsampled input signal.

The parameter *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the function [ippsSample-Down](#) for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

The parameter *downPhase* is the parameter, which determines where non-discarded sample lies within a block of upsampled filter response.

If the delay line array *pDlyLine* is not NULL, its length is defined as  $(\text{tapsLen} + \text{upFactor} - 1) / \text{upFactor}$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pTaps</i> , <i>ppState</i> or <i>pBuffer</i> is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error when <i>upFactor</i> ( <i>downFactor</i> ) is less than or equal to 0.
<code>ippStsFIRMRPhaseErr</code>	Indicates an error when <i>upPhase</i> ( <i>downPhase</i> ) is negative, or greater than or equal to <i>upFactor</i> ( <i>downFactor</i> ).

## FIRMRStreamInit

*Initializes a multi-rate stream FIR filter state.*

---

### Syntax

```
IppStatus ippsFIRMRStreamInit_16s(IppsFIRState_16s** ppState, const Ipp16s*
pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int downFactor,
int downPhase, IppRoundMode rndMode, Ipp8u* pBuffer);
```

```
IppStatus ippsFIRMRStreamInit_32f(IppsFIRState_32f** ppState, const Ipp32f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
Ipp8u* pBuffer);
```

### Parameters

<i>pTaps</i>	Pointer to the array containing the tap values.
<i>tapsLen</i>	Number of elements in the array <i>pTaps</i> .
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer version only).

<i>downFactor</i>	Downsampling factor.
<i>downPhase</i>	Phase for downsampled signal.
<i>upFactor</i>	Upsampling factor.
<i>upPhase</i>	Phase for upsampled signal.
<i>rndMode</i>	Rounding mode, the following values are possible: <i>ippRndZero</i> Specifies that floating-point values must be truncated toward zero. <i>ippRndNear</i> Specifies that floating-point values must be rounded to the nearest even integer. <i>ippRndFinancial</i> Specifies that floating-point values must be rounded down to the nearest integer if decimal value is less than 0.5, or rounded up to the nearest integer if decimal value is equal or greater than 0.5.
<i>ppState</i>	Double pointer to the stream FIR filter state structure.
<i>pBuffer</i>	Pointer to the external buffer for the stream FIR filter state structure.

## Description

The function `ippsFIRMRStreamInit` is declared in the `ipps.h` file. This function initializes a state structure for a multi-rate stream FIR filter in the external buffer. The size of this buffer must be computed by calling the function `ippsFIRMRStreamGetStateSize` beforehand. The initialization function copies the tap values from the `tapsLen`-length array `pTaps` into the state structure `ppState`. To scale integer taps use the `tapsFactor` value.

The function initializes the state structure `ppState` of a multi-rate filter; that is, a filter that internally upsamples and/or downsamples using a polyphase filter structure. It initializes the state structure in the same way as described for single-rate filters, but includes additional information about the required upsampling and downsampling parameters.

The parameter `upFactor` is the factor by which the filtered signal is internally upsampled (see description of the `ippsSampleUp` function for more details). That is, `upFactor-1` zeros are inserted between each sample of input signal.

The parameter `upPhase` is the parameter which determines where a non-zero sample lies within the `upFactor`-length block of upsampled input signal.

The parameter *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal is internally downsampled (see description of the function [ippsSample-Down](#) for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of upsampled filter response.

The parameter *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response.

The parameter *rndMode* sets the rounding mode that is used by the functions `ippsFIR` and `ippsFIROne`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pTaps</i> or <i>ppState</i> is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error when <i>upFactor</i> ( <i>downFactor</i> ) is less than or equal to 0.
<code>ippStsFIRMRPhaseErr</code>	Indicates an error when <i>upPhase</i> ( <i>downPhase</i> ) is negative, or greater than or equal to <i>upFactor</i> ( <i>downFactor</i> ).
<code>ippStsRndModNotSupported</code>	Indicates an error when the <i>rndMode</i> has an illegal value.

## FIRGetStateSize, FIRMRGetStateSize

Returns the length of the FIR filter state structure.

### Syntax

#### Case 1: Single-rate filter

```

IppStatus ippsFIRGetStateSize_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize_64fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32s_16s(int tapsLen, int* pBufferSize);

```



---

```

IppStatus ippsFIRGetStateSize32s_16s32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32sc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32sc_16sc32fc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize32fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32f(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64f_32s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_16sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32sc(int tapsLen, int* pBufferSize);
IppStatus ippsFIRGetStateSize64fc_32fc(int tapsLen, int* pBufferSize);

```

### Case 2: Multi-rate filter

```

IppStatus ippsFIRMRGetStateSize_16s(int tapsLen, int upFactor, int downFactor,
int* pBufferSize);

IppStatus ippsFIRMRGetStateSize_32f(int tapsLen, int upFactor, int downFactor,
int* pBufferSize);

IppStatus ippsFIRMRGetStateSize_32fc(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize_64f(int tapsLen, int upFactor, int downFactor,
int* pBufferSize);

IppStatus ippsFIRMRGetStateSize_64fc(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize32s_16s(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize32s_16s32f(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize32sc_16sc(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize32sc_16sc32fc(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

```

```

IppStatus ippsFIRMRGetStateSize32f_16s(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize32fc_16sc(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize64f_16s(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize64f_32s(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize64f_32f(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize64fc_16sc(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize64fc_32sc(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

IppStatus ippsFIRMRGetStateSize64fc_32fc(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);

```

## Parameters

<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>upFactor</i>	Upsampling factor for the multi-rate filter.
<i>downFactor</i>	Downsampling factor for the multi-rate filter.
<i>pBufferSize</i>	Pointer to the computed buffer size value.

## Description

The functions `ippsFIRGetStateSize` and `ippsFIRMRGetStateSize` are declared in the `ipps.h` file. These functions compute the size of the external buffer for a single-rate or multi-rate FIR filter state, respectively, and store the result in `pBufferSize`.

To compute the size of a buffer for the single-rate FIR filter state, the number of taps *tapsLen* only need be specified. To compute the size of a buffer for the multi-rate FIR filter state, the number of taps *tapsLen* and upsampling/downsampling parameters *upFactor* and *downFactor* must be specified. The parameter *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the function [ippsSampleUp](#) for more details). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The parameter *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the function [ippsSample-Down](#) for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pBufferSize</i> pointer is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error if <i>nxTapsLen</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error if <i>upFactor</i> ( <i>downFactor</i> ) is less than or equal to 0.

## FIRStreamGetStateSize, FIRMRStreamGetStateSize

*Calculates the size of the stream FIR filter state structure.*

### Syntax

#### Case 1: Single-rate filter

```
IppStatus ippsFIRStreamGetStateSize_16s(int tapsLen, int* pBufferSize);
IppStatus ippsFIRStreamGetStateSize_32f(int tapsLen, int* pBufferSize);
```

#### Case 2: Multi-rate filter

```
IppStatus ippsFIRMRStreamGetStateSize_16s(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);
IppStatus ippsFIRMRStreamGetStateSize_32f(int tapsLen, int upFactor, int
downFactor, int* pBufferSize);
```

### Parameters

<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>upFactor</i>	Upsampling factor for the multi-rate filter.
<i>downFactor</i>	Downsampling factor for the multi-rate filter.
<i>pBufferSize</i>	Pointer to the computed value of the buffer size.

## Description

The functions `ippsFIRStreamGetStateSize` and `ippsFIRMRStreamGetStateSize` are declared in the `ipps.h` file. These functions compute the size of the external buffer for a single-rate or multi-rate stream FIR filter state, respectively, and store the result in `pBufferSize`.

To calculate the size of a buffer for the single-rate filter state, the number of taps `tapsLen` only need be specified. To calculate the size of a buffer for the multi-rate FIR filter state, the number of taps `tapsLen` and upsampling/downsampling parameters `upFactor` and `downFactor` must be specified.

The parameter `upFactor` is the factor by which the filtered signal is internally upsampled (see description of the function [ippsSampleUp](#) for more details). That is, `upFactor-1` zeros are inserted between each sample of the input signal.

The parameter `downFactor` is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the function [ippsSampleDown](#) for more details). That is, `downFactor-1` output samples are discarded from each `downFactor`-length output block of the upsampled filter response.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pBufferSize</code> pointer is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error if <code>nxTapsLen</code> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error if <code>upFactor</code> ( <code>downFactor</code> ) is less than or equal to 0.

## FIRGetTaps

*Retrieves the tap values from the FIR filter state structure.*

---

### Syntax

```

IppStatus ippsFIRGetTaps_16s(const IppsFIRState_16s* pState, Ipp16s* pTaps);
IppStatus ippsFIRGetTaps_32s(const IppsFIRState_32s* pState, Ipp32s* pTaps);
IppStatus ippsFIRGetTaps_32f(const IppsFIRState_32f* pState, Ipp32f* pTaps);
IppStatus ippsFIRGetTaps_64f(const IppsFIRState_64f* pState, Ipp64f* pTaps);

```

```

IppStatus ippsFIRGetTaps32f_16s(const IppsFIRState32f_16s* pState, Ipp32f*
pTaps);

IppStatus ippsFIRGetTaps64f_16s(const IppsFIRState64f_16s* pState, Ipp64f*
pTaps);

IppStatus ippsFIRGetTaps64f_32s(const IppsFIRState64f_32s* pState, Ipp64f*
pTaps);

IppStatus ippsFIRGetTaps64f_32f(const IppsFIRState64f_32f* pState, Ipp64f*
pTaps);

IppStatus ippsFIRGetTaps_32fc(const IppsFIRState_32fc* pState, Ipp32fc*
pTaps);

IppStatus ippsFIRGetTaps_64fc(const IppsFIRState_64fc* pState, Ipp64fc*
pTaps);

IppStatus ippsFIRGetTaps32fc_16sc(const IppsFIRState32fc_16sc* pState,
Ipp32fc* pTaps);

IppStatus ippsFIRGetTaps64fc_16sc(const IppsFIRState64fc_16sc* pState,
Ipp64fc* pTaps);

IppStatus ippsFIRGetTaps64fc_32sc(const IppsFIRState64fc_32sc* pState,
Ipp64fc* pTaps);

IppStatus ippsFIRGetTaps64fc_32fc(const IppsFIRState64fc_32fc* pState,
Ipp64fc* pTaps);

IppStatus ippsFIRGetTaps32s_16s32f(const IppsFIRState32s_16s* pState, Ipp32f*
pTaps);

IppStatus ippsFIRGetTaps32sc_16sc32fc(const IppsFIRState32sc_16sc* pState,
Ipp32fc* pTaps);

IppStatus ippsFIRGetTaps32s_16s(const IppsFIRState32s_16s* pState, Ipp32s*
pTaps, int* pTapsFactor);

IppStatus ippsFIRGetTaps32sc_16sc(const IppsFIRState32sc_16sc* pState,
Ipp32sc* pTaps, int* pTapsFactor);

```

## Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pTaps</i>	Pointer to the array containing the tap values.

*pTapsFactor*                      Pointer to the scale factor for the taps of `Ipp32s` data type (for integer versions only).

## Description

The function `ippsFIRGetTaps` is declared in the `ipps.h` file. This function copies the tap values from the initialized FIR state structure *pState* to the *tapsLen*-length array *pTaps*. To scale integer taps, use the *pTapsFactor* value.

Before calling `ippsFIRGetTaps` function, the corresponding filter state structure must be initialized, for example by calling the functions `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

## Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`               Indicates an error when one of the specified pointers is `NULL`.  
`ippStsContextMatchErr`       Indicates an error when the state identifier is incorrect.

## FIRSetTaps

Sets the tap values in the FIR filter state structure.

### Syntax

```
IppStatus ippsFIRSetTaps_16s(const Ipp16s* pTaps, IppsFIRState_16s* pState);
IppStatus ippsFIRSetTaps_32s(const Ipp32s* pTaps, IppsFIRState_32s* pState);
IppStatus ippsFIRSetTaps_32f(const Ipp32f* pTaps, IppsFIRState_32f* pState);
IppStatus ippsFIRSetTaps_64f(const Ipp64f* pTaps, IppsFIRState_64f* pState);
IppStatus ippsFIRSetTaps32f_16s(const Ipp32f* pTaps, IppsFIRState32f_16s*
pState);
IppStatus ippsFIRSetTaps64f_16s(const Ipp64f* pTaps, IppsFIRState64f_16s*
pState);
IppStatus ippsFIRSetTaps64f_32s(const Ipp64f* pTaps, IppsFIRState64f_32s*
pState);
IppStatus ippsFIRSetTaps64f_32f(const Ipp64f* pTaps, IppsFIRState64f_32f*
pState);
```

```

IppStatus ippsFIRSetTaps_32fc(const Ipp32fc* pTaps, IppsFIRState_32fc*
pState);

IppStatus ippsFIRSetTaps_64fc(const Ipp64fc* pTaps, IppsFIRState_64fc*
pState);

IppStatus ippsFIRSetTaps32fc_16sc(const Ipp32fc* pTaps, IppsFIRState32fc_16sc*
pState);

IppStatus ippsFIRSetTaps64fc_16sc(const Ipp64fc* pTaps, IppsFIRState64fc_16sc*
pState);

IppStatus ippsFIRSetTaps64fc_32sc(const Ipp64fc* pTaps, IppsFIRState64fc_32sc*
pState);

IppStatus ippsFIRSetTaps64fc_32fc(const Ipp64fc* pTaps, IppsFIRState64fc_32fc*
pState);

IppStatus ippsFIRSetTaps32s_16s32f(const Ipp32f* pTaps, IppsFIRState32s_16s*
pState);

IppStatus ippsFIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps,
IppsFIRState32sc_16sc* pState);

IppStatus ippsFIRSetTaps32s_16s(const Ipp32s* pTaps, IppsFIRState32s_16s*
pState, int tapsFactor);

IppStatus ippsFIRSetTaps32sc_16sc(const Ipp32sc* pTaps, IppsFIRState32sc_16sc*
pState, int tapsFactor);

```

## Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pTaps</i>	Pointer to the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer versions only).

## Description

The function `ippsFIRSetTaps` is declared in the `ipps.h` file. This function sets new tap values in the previously initialized FIR filter state structure *pState*. New tap values must be specified in the array *pTaps*. The length of the array *pTaps* must be equal to the *tapsLen* parameter value of the initialized filter state.

To scale integer taps, use the *tapsFactor* value.

Before calling `ippsFIRSetTaps` function, the corresponding filter state structure must be initialized by calling the function `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIRGetDlyLine

*Retrieves the delay line contents from the FIR filter state structure.*

---

### Syntax

```

IppStatus ippsFIRGetDlyLine_16s(const IppsFIRState_16s* pState, Ipp16s*
pDlyLine);

IppStatus ippsFIRGetDlyLine_32f(const IppsFIRState_32f* pState, Ipp32f*
pDlyLine);

IppStatus ippsFIRGetDlyLine_64f(const IppsFIRState_64f* pState, Ipp64f*
pDlyLine);

IppStatus ippsFIRGetDlyLine32s_16s(const IppsFIRState32s_16s* pState, Ipp16s*
pDlyLine);

IppStatus ippsFIRGetDlyLine32f_16s(const IppsFIRState32f_16s* pState, Ipp16s*
pDlyLine);

IppStatus ippsFIRGetDlyLine64f_16s(const IppsFIRState64f_16s* pState, Ipp16s*
pDlyLine);

IppStatus ippsFIRGetDlyLine64f_32s(const IppsFIRState64f_32s* pState, Ipp32s*
pDlyLine);

IppStatus ippsFIRGetDlyLine64f_32f(const IppsFIRState64f_32f* pState, Ipp32f*
pDlyLine);

IppStatus ippsFIRGetDlyLine_32fc(const IppsFIRState_32fc* pState, Ipp32fc*
pDlyLine);

IppStatus ippsFIRGetDlyLine_64fc(const IppsFIRState_64fc* pState, Ipp64fc*
pDlyLine);

```



```

IppStatus ippsFIRGetDlyLine32sc_16sc(const IppsFIRState32sc_16sc* pState,
Ipp16sc* pDlyLine);

IppStatus ippsFIRGetDlyLine32fc_16sc(const IppsFIRState32fc_16sc* pState,
Ipp16sc* pDlyLine);

IppStatus ippsFIRGetDlyLine64fc_16sc(const IppsFIRState64fc_16sc* pState,
Ipp16sc* pDlyLine);

IppStatus ippsFIRGetDlyLine64fc_32sc(const IppsFIRState64fc_32sc* pState,
Ipp32sc* pDlyLine);

IppStatus ippsFIRGetDlyLine64fc_32fc(const IppsFIRState64fc_32fc* pState,
Ipp32fc* pDlyLine);

```

### Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pDlyLine</i>	Pointer to the array holding the delay line values.

### Description

The function `ippsFIRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from the state structure *pState* and stores them into *pDlyLine*. The destination array *pDlyLine* contains samples in the reverse order as compared to the order of samples in the source vector.

Before calling `ippsFIRGetDlyLine`, the corresponding filter state structure must be initialized, for example by calling the functions `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRSetDlyLine

*Sets the delay line contents in the FIR filter state structure.*

---

### Syntax

```

IppStatus ippsFIRSetDlyLine_16s(IppsFIRState_16s* pState, const Ipp16s*
pDlyLine);

IppStatus ippsFIRSetDlyLine_32f(IppsFIRState_32f* pState, const Ipp32f*
pDlyLine);

IppStatus ippsFIRSetDlyLine_64f(IppsFIRState_64f* pState, const Ipp64f*
pDlyLine);

IppStatus ippsFIRSetDlyLine32s_16s(IppsFIRState32s_16s* pState, const Ipp16s*
pDlyLine);

IppStatus ippsFIRSetDlyLine32f_16s(IppsFIRState32f_16s* pState, const Ipp16s*
pDlyLine);

IppStatus ippsFIRSetDlyLine64f_16s(IppsFIRState64f_16s* pState, const Ipp16s*
pDlyLine);

IppStatus ippsFIRSetDlyLine64f_32s(IppsFIRState64f_32s* pState, const Ipp32s*
pDlyLine);

IppStatus ippsFIRSetDlyLine64f_32f(IppsFIRState64f_32f* pState, const Ipp32f*
pDlyLine);

IppStatus ippsFIRSetDlyLine_32fc(IppsFIRState_32fc* pState, const Ipp32fc*
pDlyLine);

IppStatus ippsFIRSetDlyLine_64fc(IppsFIRState_64fc* pState, const Ipp64fc*
pDlyLine);

IppStatus ippsFIRSetDlyLine32sc_16sc(IppsFIRState32sc_16sc* pState, const
Ipp16sc* pDlyLine);

IppStatus ippsFIRSetDlyLine32fc_16sc(IppsFIRState32fc_16sc* pState, const
Ipp16sc* pDlyLine);

IppStatus ippsFIRSetDlyLine64fc_16sc(IppsFIRState64fc_16sc* pState, const
Ipp16sc* pDlyLine);

```

```

IppStatus ippsFIRSetDlyLine64fc_32sc(IppsFIRState64fc_32sc* pState, const
Ipp32sc* pDlyLine);

IppStatus ippsFIRSetDlyLine64fc_32fc(IppsFIRState64fc_32fc* pState, const
Ipp32fc* pDlyLine);

```

### Parameters

*pState*                      Pointer to the FIR filter state structure.

*pDlyLine*                  Pointer to the array holding the delay line values.

### Description

The function `ippsFIRSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. The source array *pDlyLine* must contain samples in the reverse order as compared to the order of samples in the source vector.

Before calling `ippsFIRGetDlyLine` the corresponding filter state structure must be initialized by calling the `ippsFIRInitAlloc` or `ippsFIRMRInitAlloc`.

### Return Values

`ippStsNoErr`                Indicates no error.

`ippStsNullPtrErr`        Indicates an error when the *pState* pointer is NULL.

`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIROne

*Filters a single sample through a FIR filter.*

### Syntax

#### Case 1: Integer sample

```

IppStatus ippsFIROne_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, IppsFIRStates_16s*
pState, int scaleFactor);

IppStatus ippsFIROne_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, IppsFIRStates_32s*
pState, int scaleFactor);

IppStatus ippsFIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
IppsFIRState32s_16s* pState, int scaleFactor);

```

```

IppStatus ippsFIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
IppsFIRState32f_16s* pState, int scaleFactor);

IppStatus ippsFIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
IppsFIRState64f_16s* pState, int scaleFactor);

IppStatus ippsFIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
IppsFIRState64f_32s* pState, int scaleFactor);

IppStatus ippsFIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
IppsFIRState32sc_16sc* pState, int scaleFactor);

IppStatus ippsFIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
IppsFIRState32fc_16sc* pState, int scaleFactor);

IppStatus ippsFIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
IppsFIRState64fc_16sc* pState, int scaleFactor);

IppStatus ippsFIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
IppsFIRState64fc_32sc* pState, int scaleFactor);

```

## Case 2: Floating point sample

```

IppStatus ippsFIROne_32f(Ipp32f src, Ipp32f* pDstVal, IppsFIRState_32f*
pState);

IppStatus ippsFIROne_64f(Ipp64f src, Ipp64f* pDstVal, IppsFIRState_64f*
pState);

IppStatus ippsFIROne64f_32f(Ipp32f src, Ipp32f* pDstVal, IppsFIRState64f_32f*
pState);

IppStatus ippsFIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsFIRState_32fc*
pState);

IppStatus ippsFIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal, IppsFIRState_64fc*
pState);

IppStatus ippsFIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal,
IppsFIRState64fc_32fc* pState);

```

## Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>src</i>	Input sample.
<i>pDstVal</i>	Pointer to the output sample.

*scaleFactor*                      Scale factor, refer to [Integer Scaling](#).

## Description

The function `ippsFIROne` is declared in the `ipps.h` file. This function filters a single sample *src* through a single-rate filter and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted  $x(n)$  and the taps are denoted  $h(i)$ .

The return value  $y(n)$  is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Before calling `ippsFIROne`, initialize the corresponding filter state by calling `ippsFIRInitAlloc`. Specify the number of taps *tapsLen*, the tap values in *pTaps*, and the delay line values in *pDlyLine* beforehand.

## Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error when one of the specified pointers is `NULL`.  
`ippStsContextMatchErr`          Indicates an error when the state identifier is incorrect.

## FIR

*Filters a source vector through a single-rate or multi-rate FIR filter.*

### Syntax

#### Case 1: Not-in-place operation on integer samples

```
ippStatus ippsFIR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
IppsFIRState_16s* pState, int scaleFactor);

ippStatus ippsFIR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int numIters,
IppsFIRState_32s* pState, int scaleFactor);
```

```
IppStatus ippsFIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
IppsFIRState32s_16s* pState, int scaleFactor);
```

```
IppStatus ippsFIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
IppsFIRState32f_16s* pState, int scaleFactor);
```

```
IppStatus ippsFIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int numIters,
IppsFIRState64f_16s* pState, int scaleFactor);
```

```
IppStatus ippsFIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int numIters,
IppsFIRState64f_32s* pState, int scaleFactor);
```

```
IppStatus ippsFIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
numIters, IppsFIRState32sc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsFIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
numIters, IppsFIRState32fc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsFIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int
numIters, IppsFIRState64fc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsFIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int
numIters, IppsFIRState64fc_32sc* pState, int scaleFactor);
```

### **Case 2 Not-in-place operation on floating point samples**

```
IppStatus ippsFIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
IppsFIRState_32f* pState);
```

```
IppStatus ippsFIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
IppsFIRState_64f* pState);
```

```
IppStatus ippsFIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
IppsFIRState_32fc* pState);
```

```
IppStatus ippsFIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int numIters,
IppsFIRState_64fc* pState);
```

```
IppStatus ippsFIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
IppsFIRState64f_32f* pState);
```

```
IppStatus ippsFIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int numIters,
IppsFIRState64fc_32fc* pState);
```

### **Case 3 In-place operation on integer samples**

```
IppStatus ippsFIR_16s_ISfs(Ipp16s* pSrcDst, int numIters, IppsFIRState_16s*
pState, int scaleFactor);
```

```

IppStatus ippsFIR_32s_ISfs(Ipp32s* pSrcDst, int numIters, IppsFIRState_32s*
pState, int scaleFactor);

IppStatus ippsFIR32s_16s_ISfs(Ipp16s* pSrcDst, int numIters,
IppsFIRState32s_16s* pState, int scaleFactor);

IppStatus ippsFIR32f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
IppsFIRState32f_16s* pState, int scaleFactor);

IppStatus ippsFIR64f_16s_ISfs(Ipp16s* pSrcDst, int numIters,
IppsFIRState64f_16s* pState, int scaleFactor);

IppStatus ippsFIR64f_32s_ISfs(Ipp32s* pSrcDst, int numIters,
IppsFIRState64f_32s* pState, int scaleFactor);

IppStatus ippsFIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
IppsFIRState32sc_16sc* pState, int scaleFactor);

IppStatus ippsFIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
IppsFIRState32fc_16sc* pState, int scaleFactor);

IppStatus ippsFIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
IppsFIRState64fc_16sc* pState, int scaleFactor);

IppStatus ippsFIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
IppsFIRState64fc_32sc* pState, int scaleFactor);

```

#### **Case 4 In-place operation on floating point samples**

```

IppStatus ippsFIR_32f_I(Ipp32f* pSrcDst, int numIters, IppsFIRState_32f*
pState);

IppStatus ippsFIR_64f_I(Ipp64f* pSrcDst, int numIters, IppsFIRState_64f*
pState);

IppStatus ippsFIR64f_32f_I(Ipp32f* pSrcDst, int numIters, IppsFIRState64f_32f*
pState);

IppStatus ippsFIR_32fc_I(Ipp32fc* pSrcDst, int numIters, IppsFIRState_32fc*
pState);

IppStatus ippsFIR_64fc_I(Ipp64fc* pSrcDst, int numIters, IppsFIRState_64fc*
pState);

IppStatus ippsFIR64fc_32fc_I(Ipp32fc* pSrcDst, int numIters,
IppsFIRState64fc_32fc* pState);

```

## Parameters

<i>pState</i>	Pointer to the FIR filter state structure.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>numIters</i>	Number of elements of the source vector to be filtered.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsFIR` is declared in the `ipps.h` file. This function filters a source vector *pSrc* or *pSrcDst* through a FIR or stream FIR filters, and stores the results in *pDst* or *pSrcDst*, respectively. Both filters can be single-rate or multi-rate. The filter state structure *pState* defines the type of the filter and specifies its parameters. It must be initialized before calling the function `ippsFIR` using one of the initializing functions (`ippsFIRInitAlloc`, `ippsFIRStreamInitAlloc`, `ippsFIRInit`, `ippsFIRStreamInit` for single-rate filters, or `ippsFIRMRInitAlloc`, `ippsFIRMRStreamInitAlloc`, `ippsFIRMRInit`, `ippsFIRMRStreamInit` for multi-rate filters).

The parameter *numIters* specified the number of elements to be filtered. The length of the source and destination vectors depends on this parameter.

For single-rate FIR filters, the length of the *pSrc* (*pSrcDst*) and *pDst* (*pSrcDst*) is equal to *numIters*.

For single-rate stream FIR filters, the length of the *pSrc* (*pSrcDst*) is equal to *numIters*+*tapLen*-1, and length of the *pDst* (*pSrcDst*) is equal to *numIters*.

For multi-rate FIR filters, the length of the *pSrc* is equal to *numIters*\**downFactor*, the length of the *pDst* is equal to *numIters*\**upFactor*.

For multi-rate stream FIR filters, the length of the *pSrc* is equal to *numIters*\**downFactor* + (*tapLen*-1)/ *upFactor*+ 1, the length of the *pDst* is equal to *numIters* \**upFactor*.

In the following definition of the FIR filter, the sample to be filtered is denoted  $x(n)$ , the taps are denoted  $h(i)$ , and the return value is  $y(n)$ .

The return value  $y(n)$  is defined by the formula for a single-rate filter:

For a single-rate filter the return value  $y(n)$  is defined by the following formula:



$$y(n) = \sum_{i=0}^{\text{tapsLen}-1} h(i) \cdot x(n-i), \quad 0 \leq n < \text{numIters}$$

The results are identical to *numIters* consecutive calls to the function `ippsFIROne`.

When the function completes calculation, it updates the delay line values stored in the state structure (for structures with delay line only).

The multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps. Thus, the function does not create an internal (*upFactor\*numIters\*downFactor*)-size buffer to store the upsampling result.

Code examples 6-7 and 6-8 below illustrate single-rate filtering with the function `ippsFIR_32f` and using of multi-rate filter functions for vector interpolation (resizing).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>numIters</i> is less or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## Example 6-7 Single-Rate Filtering with the ippsFIR Function

```
#undef NUMITERS

#define NUMITERS 150

    int n;

    IppStatus status;

    IppsIIRState_32f *ictx;
    IppsFIRState_32f *fctx;
    IppsFIRState_32f *fsctx;

    Ipp32f *x = ippsMalloc_32f(NUMITERS+10),
            *y = ippsMalloc_32f(NUMITERS),
            *z = ippsMalloc_32f(NUMITERS);

    const float taps[] = {
        0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
        0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
    };

    for (n =0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *0.2);

    ippsIIRInitAlloc_32f( &ictx, taps, 10, NULL );
    ippsFIRInitAlloc_32f( &fctx, taps, 11, NULL );
    ippsFIRStreamInitAlloc_32f( &fsctx, taps, 11 );

    status = ippsIIR_32f( x, y, NUMITERS, ictx);
    printf_32f("IIR 32f output + 120 =", y + 120, 5, status);

    ippsIIRFree_32f(ictx);

    status = ippsFIR_32f( x, z, NUMITERS, fctx );
    printf_32f("FIR 32f output + 120 =", z + 120, 5, status);

    ippsFIRFree_32f(fctx);

    status = ippsFIR_32f( x, z, NUMITERS, fsctx );
    printf_32f("FIR 32f output + 110 =", z + 110, 5, status);

    ippsFIRFree_32f(fsctx);
```

```
    ippsFree(z);  
    ippsFree(y);  
    ippsFree(x);  
    return status;  
}
```

Output:

```
IIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
```

```
FIR 32f output + 120 = 0.000000 0.049896 0.030838 -0.030838 -0.049896
```

Matlab\* Analog:

```
>> F = 0.2; N = 150; n = 0:N-1; x = sin(2*pi*n*F);
```

```
y = filter(fir1(10,0.15),1,x); y(121:125)
```

### Example 6-8 Using Multi-Rate FIR Functions

Input  
vector (src) is:

```
x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,...
```

Taps are (tapsLen=8 for example):

```
t0,t1,t2,t3,t4,t5,t6,t7
```

We want to resize the input vector in 2/3 proportion, so upsample factor for MR FIR is equal to 2 and downsample factor is equal to 3:

Upsample operation:

```
x0,0,x1,0,x2,0,x3,0,x4,0,x5,0,x6,0,x7,0,x8,0,x9,0,...
```

Interpolation (filtering):

```
y0 = x0*t0+0*t1+x1*t2+0*t3+x2*t4+0*t5+x3*t6+0*t7 = (optimized) = x0*t0+x1*t2+x2*t4+x3*t6
```

```
y1 = 0*t0+x1*t1+0*t2+x2*t3+0*t4+x3*t5+0*t6+x4*t7 = (optimized) = x1*t1+x2*t3+x3*t5+x4*t7
```

```
y2 = x1*t0+0*t1+x2*t2+0*t3+x3*t4+0*t5+x4*t6+0*t7 = (optimized) = x1*t0+x2*t2+x3*t4+x4*t6
```

```
y3 = 0*t0+x2*t1+0*t2+x3*t3+0*t4+x4*t5+0*t6+x5*t7 = (optimized) = x2*t1+x3*t3+x4*t5+x5*t7
```

```
y4 = x2*t0+0*t1+x3*t2+0*t3+x4*t4+0*t5+x5*t6+0*t7 = (optimized) = x2*t0+x3*t2+x4*t4+x5*t6
```

```
y5 = 0*t0+x3*t1+0*t2+x4*t3+0*t4+x5*t5+0*t6+x6*t7 = (optimized) = x2*t1+x3*t3+x4*t5+x5*t7
```

as a result after this operation y vector is 2x times longer than x

Downsample operation:

```
dst = y0, y3, y6, ... (optimized version doesn't calculate y1,y2,etc.) -
```

---

```

interpolation                so dst will have 2/3 length in comparison with src with
                                that is defined by taps coefficients ( so it can be cubic,
supersampling,                lanczos or any other spline polynomial)

"C" analogue:
#define n 99
IppsFIRState_32f *pState;
Ipp32f pTaps[8] = { t0,t1,t2,t3,t4,t5,t6,t7};
int tapsLen = 8;
int upFactor = 2;
int upPhase = 0;
int downFactor = 3;
int downPhase = 0;
Ipp32f *pDlyLine = NULL;
Ipp32f pSrc[n] = { x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,... };
Ipp32f pDst[2*n/3] ;
int numIters = n;
IppStatus status;

status = ippsFIRMRInitAlloc_32f ( &pState, pTaps, tapsLen, upFactor, upPhase,
                                downFactor, downPhase, pDlyLine);

if( ippStsNoErr != status) return error;

status = ippsFIR_32f( pSrc, pDst, numIters, pState);
// pDst will contain y0, y3, y6,... according to the above formulas;

if( ippStsNoErr != status) return error;

status = ippsFIRFree_32f(
pState );

if( ippStsNoErr != status) return error;

.....

```

## FIROne\_Direct

*Directly filters a single sample through a FIR filter.*

### Syntax

#### Case 1: Not-in-place operation on integer sample

```
IppStatus ippsFIROne_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const Ipp16s*
pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIROne32f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIROne64f_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIROne64f_Direct_32s_Sfs(Ipp32s src, Ipp32s* pDstVal, const
Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIROne32fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIROne64fc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIROne64fc_Direct_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal, const
Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIROne32s_Direct_16s_Sfs(Ipp16s src, Ipp16s* pDstVal, const
Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIROne32sc_Direct_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal, const
Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

**Case 2: Not-in-place operation on floating point sample**

```

IppStatus ippsFIROne_Direct_32f(Ipp32f src, Ipp32f* pDstVal, const Ipp32f*
pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_64f(Ipp64f src, Ipp64f* pDstVal, const Ipp64f*
pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal, const Ipp32fc*
pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIROne_Direct_64fc(Ipp64fc src, Ipp64fc* pDstVal, const Ipp64fc*
pTaps, int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIROne64f_Direct_32f(Ipp32f src, Ipp32f* pDstVal, const Ipp64f*
pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIROne64fc_Direct_32fc(Ipp32fc src, Ipp32fc* pDstVal, const
Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);

```

**Case 3: In-place operation on integer sample**

```

IppStatus ippsFIROne_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp16s* pTaps,
int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32f_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp32f*
pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64f_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp64f*
pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64f_Direct_32s_ISfs(Ipp32s* pSrcDstVal, const Ipp64f*
pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp32fc*
pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp64fc*
pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDstVal, const Ipp64fc*
pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int scaleFactor);

IppStatus ippsFIROne32s_Direct_16s_ISfs(Ipp16s* pSrcDstVal, const Ipp32s*
pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine, int* pDlyLineIndex,
int scaleFactor);

```

```
IppStatus ippsFIROne32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDstVal, const Ipp32sc*
pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int* pDlyLineIndex,
int scaleFactor);
```

## Case 4: In-place operation on floating point sample

```
IppStatus ippsFIROne_Direct_32f_I(Ipp32f* pSrcDstVal, const Ipp32f* pTaps,
int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIROne_Direct_64f_I(Ipp64f* pSrcDstVal, const Ipp64f* pTaps,
int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIROne_Direct_32fc_I(Ipp32fc* pSrcDstVal, const Ipp32fc* pTaps,
int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIROne_Direct_64fc_I(Ipp64fc* pSrcDstVal, const Ipp64fc* pTaps,
int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIROne64f_Direct_32f_I(Ipp32f* pSrcDstVal, const Ipp64f* pTaps,
int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIROne64fc_Direct_32fc_I(Ipp32fc* pSrcDstVal, const Ipp64fc*
pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
```

## Parameters

<i>src</i>	Input sample.
<i>pDstVal</i>	Pointer to the output sample.
<i>pSrcDstVal</i>	Pointer to the input and output sample for in-place operation.
<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>pTapsQ15</i>	Pointer to the array containing the tap values, represented in Q0.15 format. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <i>Ipp32s</i> data type.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * tapsLen$ . Note that the delay line length is different than that for FIR filters using state structure.



<i>pDlyLineIndex</i>	Pointer to the current delay line index.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsFIROne_Direct` is declared in the `ipps.h` file. This function directly filters a single sample *src* or *pSrcDstVal* through a single-rate filter and stores the result in *pDstVal* or *pSrcDstVal*. The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps, the *tapsFactor* value is used. The set of *tapsLen* input samples is copied twice to the  $2 \cdot \textit{tapsLen}$ -length array *pDlyLine*. Double length of the delay line in direct FIR filters is used to improve filter performance by decreasing the number of sample copyings. The current delay line index is specified in the *pDlyLineIndex*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated. In the following definition of the FIR filter, the sample to be filtered is denoted  $x(n)$  and the taps are denoted  $h(i)$ .

The return value  $y(n)$  is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{\textit{tapsLen} - 1} h(i) \cdot x(n - i)$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.

## FIR\_Direct

*Directly filters a source vector through a single-rate FIR filter.*

---

### Syntax

#### Case 1: Not-in-place operation on integer samples

```
IppStatus ippsFIR_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
numIters, const Ipp16s* pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
numIters, const Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
numIters, const Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int
numIters, const Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
int numIters, const Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
int numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine,
int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
int numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc*
pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

**Case 2: Not-in-place operation on floating point samples**

```
IppStatus ippsFIR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
const Ipp32f* pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
const Ipp64f* pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
numIters, const Ipp32fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int*
pDlyLineIndex);
```

```
IppStatus ippsFIR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
numIters, const Ipp64fc* pTaps, int tapsLen, Ipp64fc* pDlyLine, int*
pDlyLineIndex);
```

```
IppStatus ippsFIR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
numIters, const Ipp64f* pTaps, int tapsLen, Ipp32f* pDlyLine, int*
pDlyLineIndex);
```

```
IppStatus ippsFIR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
numIters, const Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int*
pDlyLineIndex);
```

**Case 3: In-place operation on integer samples**

```
IppStatus ippsFIR_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const Ipp16s*
pTapsQ15, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
Ipp32f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
Ipp64f* pTaps, int tapsLen, Ipp16s* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters, const
Ipp64f* pTaps, int tapsLen, Ipp32s* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
Ipp32fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
Ipp64fc* pTaps, int tapsLen, Ipp16sc* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters, const
Ipp64fc* pTaps, int tapsLen, Ipp32sc* pDlyLine, int* pDlyLineIndex, int
scaleFactor);
```

```
IppStatus ippsFIR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
Ipp32s* pTaps, int tapsLen, int tapsFactor, Ipp16s* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

```
IppStatus ippsFIR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters, const
Ipp32sc* pTaps, int tapsLen, int tapsFactor, Ipp16sc* pDlyLine, int*
pDlyLineIndex, int scaleFactor);
```

## Case 4: In-place operation on floating point samples

```
IppStatus ippsFIR_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp32f*
pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIR_Direct_64f_I(Ipp64f* pSrcDst, int numIters, const Ipp64f*
pTaps, int tapsLen, Ipp64f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const Ipp32fc*
pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters, const Ipp64fc*
pTaps, int tapsLen, Ipp64fc* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp64f*
pTaps, int tapsLen, Ipp32f* pDlyLine, int* pDlyLineIndex);
```

```
IppStatus ippsFIR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
Ipp64fc* pTaps, int tapsLen, Ipp32fc* pDlyLine, int* pDlyLineIndex);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the output array.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>numIters</i>	Number of elements in the source vector to be filtered.

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>pTapsQ15</i>	Pointer to the array containing the tap values, represented in Q0.15 format. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * \textit{tapsLen}$ . Note that the delay line length is different than that for FIR filters using state structure.
<i>pDlyLineIndex</i>	Pointer to the current delay line index.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsFIR_Direct` is declared in the `ipps.h` file. This function filters an source vector *pSrc* or *pSrcDst* containing *numIters* samples through a single-rate filter, and stores the resulting *numIters* samples in *pDst* or *pSrcDst*, respectively. The results are identical to *numIters* consecutive calls to `ippsFIROne_Direct`. The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used. The set of *tapsLen* input samples is copied to the  $2 * \textit{tapsLen}$ -length array *pDlyLine*. Double length of the delay line in direct FIR filters is used to improve filter performance by decreasing the number of sample copyings. The current delay line index is specified in the *pDlyLineIndex*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

In the following definition of the FIR filter, the sample to be filtered is denoted  $x(n)$ , the taps are denoted  $h(i)$ , and the return value is  $y(n)$ .

The return value  $y(n)$  is defined by the formula for a single-rate filter:

$$y(n) = \sum_{i=0}^{\textit{tapsLen}-1} h(i) \cdot x(n-i), \quad 0 \leq n < \textit{numIters}$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFIRLenErr</code>	Indicates an error when <code>tapsLen</code> is less than or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <code>numIters</code> is less than or equal to 0.

## FIRMR\_Direct

*Directly filters a source vector through a multi-rate FIR filter.*

---

### Syntax

#### Case 1: Not-in-place operation on integer samples

```
IppStatus ippsFIRMR32f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
numIters, const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int
downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64f_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64f_Direct_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int
numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
downFactor, int downPhase, Ipp32s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR32fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
int numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase,
int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64fc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
int numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64fc_Direct_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst,
int numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase,
int downFactor, int downPhase, Ipp32sc* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR32s_Direct_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
numIters, const Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor,
int upPhase, int downFactor, int downPhase, Ipp16s* pDlyLine, int
scaleFactor);
```

```
IppStatus ippsFIRMR32sc_Direct_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst,
int numIters, const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor,
int upPhase, int downFactor, int downPhase, Ipp16sc* pDlyLine, int
scaleFactor);
```

### Case 2: Not-in-place operation on floating point samples

```
IppStatus ippsFIRMR_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int numIters,
const Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
int downPhase, Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMR64f_Direct_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
numIters, const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int
downFactor, int downPhase, Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMR_Direct_64f(const Ipp64f* pSrc, Ipp64f* pDst, int numIters,
const Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
int downPhase, Ipp64f* pDlyLine);
```

```
IppStatus ippsFIRMR_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
numIters, const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int
downFactor, int downPhase, Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMR64fc_Direct_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
downFactor, int downPhase, Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMR_Direct_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
numIters, const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int
downFactor, int downPhase, Ipp64fc* pDlyLine);
```

### Case 3: In-place operation on integer samples

```
IppStatus ippsFIRMR32f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
Ipp32f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, Ipp16s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64f_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, Ipp16s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64f_Direct_32s_ISfs(Ipp32s* pSrcDst, int numIters, const
Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, Ipp32s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR32fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
const Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64fc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR64fc_Direct_32sc_ISfs(Ipp32sc* pSrcDst, int numIters,
const Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor,
int downPhase, Ipp32sc* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR32s_Direct_16s_ISfs(Ipp16s* pSrcDst, int numIters, const
Ipp32s* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase, int
downFactor, int downPhase, Ipp16s* pDlyLine, int scaleFactor);
```

```
IppStatus ippsFIRMR32sc_Direct_16sc_ISfs(Ipp16sc* pSrcDst, int numIters,
const Ipp32sc* pTaps, int tapsLen, int tapsFactor, int upFactor, int upPhase,
int downFactor, int downPhase, Ipp16sc* pDlyLine, int scaleFactor);
```

## Case 4: In-place operation on floating point samples

```
IppStatus ippsFIRMR_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const Ipp32f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMR64f_Direct_32f_I(Ipp32f* pSrcDst, int numIters, const
Ipp64f* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, Ipp32f* pDlyLine);
```

```
IppStatus ippsFIRMR_Direct_64f_I(Ipp64f* pSrcDst, int numIters, const Ipp64f*
pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int downPhase,
Ipp64f* pDlyLine);
```

```
IppStatus ippsFIRMR_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
Ipp32fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, Ipp32fc* pDlyLine);
```

```
IppStatus ippsFIRMR64fc_Direct_32fc_I(Ipp32fc* pSrcDst, int numIters, const
Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, Ipp32fc* pDlyLine);
```



```
IppStatus ippsFIRMR_Direct_64fc_I(Ipp64fc* pSrcDst, int numIters, const
Ipp64fc* pTaps, int tapsLen, int upFactor, int upPhase, int downFactor, int
downPhase, Ipp64fc* pDlyLine);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>numIters</i>	Parameter associated with the number of samples to be filtered by the function. The $(numIters * downFactor)$ elements of the source vector are filtered and the resulting $(numIters * upFactor)$ samples are stored in the output array.
<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>tapsFactor</i>	Scale factor for the taps of <i>Ipp32s</i> data type.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $(tapsLen + upFactor - 1) / upFactor$ .
<i>upFactor</i>	Factor for upsampling the multi-rate signals.
<i>downFactor</i>	Factor for downsampling the multi-rate signals.
<i>upPhase</i>	Phase for upsampling the multi-rate signals.
<i>downPhase</i>	Phase for downsampling the multi-rate signals.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsFIRMR_Direct` is declared in the `ipps.h` file. This function filters an source vector *pSrc* or *pSrcDst* through a multi-rate filter, and stores the resulting samples in *pDst* or *pSrcDst*, respectively. The values of filter coefficients (taps) are specified in the *tapsLen*-length array *pTaps*. To scale integer taps the *tapsFactor* value is used. The array *pDlyLine* specifies the delay line values. The input array contains  $(numIters * downFactor)$  samples, and the output array stores the resulting  $(numIters * upFactor)$  samples. The

multi-rate filtering is considered as a sequence of three operations: upsampling, filtering with a single-rate FIR filter, and downsampling. The algorithm is implemented as a single operation including the above-mentioned three steps.

The parameter *upFactor* is the factor by which the filtered signal is internally upsampled (see description of the function [ippsSampleUp](#) for more details). That is, *upFactor*-1 zeros are inserted between each sample of the input signal.

The parameter *upPhase* is the parameter which determines where a non-zero sample lies within the *upFactor*-length block of upsampled input signal.

The parameter *downFactor* is the factor by which the FIR response obtained by filtering an upsampled input signal, is internally downsampled (see description of the function [ippsSampleDown](#) for more details). That is, *downFactor*-1 output samples are discarded from each *downFactor*-length output block of the upsampled filter response.

The parameter *downPhase* is the parameter which determines where non-discarded sample lies within a block of upsampled filter response. The length of the delay line array *pDlyLine* is defined as  $(\text{tapsLen} + \text{upFactor} - 1) / \text{upFactor}$ . The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>tapsLen</i> is less than or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>numIters</i> is less than or equal to 0.
<code>ippStsFIRMRFactorErr</code>	Indicates an error when <i>upFactor</i> ( <i>downFactor</i> ) is less than or equal to 0.
<code>ippStsFIRMRPhaseErr</code>	Indicates an error when <i>upPhase</i> ( <i>downPhase</i> ) is negative, or greater than or equal to <i>upFactor</i> ( <i>downFactor</i> ).

## FIRSparseInit

*Initializes a sparse FIR filter structure.*

---

### Syntax

```
IppStatus ippsFIRSparseInit_32f(IppsFIRSparseState_32f** ppState, const
Ipp32f* pNZTaps, const Ipp32s* pNZTapPos, int nzTapsLen, const Ipp32f*
pDlyLine, Ipp8u* pBuffer);
```

## Parameters

<i>pNZTaps</i>	Pointer to the array containing the non-zero tap values. The number of elements in the array is <i>nzTapsLen</i> .
<i>pNZTapPos</i>	Pointer to the array containing positions of the non-zero tap values. The number of elements in the array is <i>nzTapsLen</i> .
<i>nzTapsLen</i>	Number of elements in the array with non-zero tap values.
<i>pDlyLine</i>	Pointer to the array containing the delay line values.
<i>ppState</i>	Double pointer to the sparse FIR state structure.
<i>pBuffer</i>	Pointer to the external buffer for the sparse FIR state structure.

## Description

The functions `ippsFIRSparseInit` is declared in the `ipps.h` file. This function initializes a sparse FIR filter state structure *ppState* in the external buffer *pBuffer*. The size of this buffer must be computed previously by calling the function `FIRSparseGetStateSize`. The initialization function copies the values of filter coefficients from the array *pNZTaps* containing *nzTapsLen* non-zero taps and their positions from the array *pNZTapPos* into the state structure *ppState*. The array *pDlyLine* specifies the delay line values. The number of elements in this array is *pNZTapPos[nzTapsLen - 1]*. If the pointer to the array *pDlyLine* is not NULL, the array contents are copied into the state structure *ppState*, otherwise the delay line values in the state structure are initialized to 0.




---

**CAUTION.** The values of *nzTapsLen* and *pNZTapPos[nzTapsLen - 1]* must be equal to those specified in the function `FIRSparseGetStateSize`.

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers <i>ppState</i> , <i>pNZTaps</i> , <i>pNZTapPos</i> , or <i>pBuffer</i> is NULL.
<code>ippStsFIRLenErr</code>	Indicates an error if <i>nzTapsLen</i> is less than or equal to 0.

`ippStsSparseErr` Indicates an error if positions of the non-zero taps are not in ascending order, or are negative or repetitive.

## FIRSparseGetStateSize

*Computes the size of the external buffer for the sparse FIR filter structure.*

---

### Syntax

```
IppStatus ippFIRSparseGetStateSize_32f(int nzTapsLen, int order, int* pStateSize);
```

### Parameters

<i>nzTapsLen</i>	Number of elements in the array containing the non-zero tap values.
<i>order</i>	Order of the sparse FIR filter.
<i>pStateSize</i>	Pointer to the computed value of the external buffer.

### Description

The function `ippFIRSparseGetStateSize` is declared in the `ipps.h` file. This function computes the size of the external buffer for a sparse FIR filter structure that is required for the function `ippFIRSparseInit`. Computation is based on the specified number of non-zero filter coefficients *nzTapsLen* and filter order *order* that is equal to the number of elements in the delay line `pNZTapPos [nzTapsLen - 1]` (see description of the function `ippFIRSparseInit`). The result value is stored in the *pStateSize*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pStateSize</i> pointer is <code>NULL</code> .
<code>ippStsFIRLenErr</code>	Indicates an error if <i>nzTapsLen</i> or <i>order</i> is less than or equal to 0; or <i>nzTapsLen</i> is more than <i>order</i> .

## FIRSparse

*Filters a source vector through a sparse FIR filter.*

### Syntax

```
IppStatus ippsFIRSparse_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
IppFIRSparseState_32f* pState);
```

### Parameters

<i>pState</i>	Pointer to the sparse FIR filter state structure.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements that are filtered.

### Description

The function `ippsFIRSparse` is declared in the `ipps.h` file. This function applies the sparse FIR filter to the *len* elements of the source vector *pSrc*, and stores the results in *pDst*. The filter parameters - the number of non-zero taps *nzTapsLen*, their values *pNZTaps* and their positions *pNZTapPos*, and the delay line values *pDlyLine* - are specified in the sparse FIR filter structure *pState* that should be previously initialized by calling the function `ippsFIRSparseInit`.

In the following definition of the sparse FIR filter, the sample to be filtered is denoted  $x(n)$ , the non-zero taps are denoted  $pNZTaps(i)$ , their positions are denoted  $pNZTapPos(i)$  and the return value is  $y(n)$ .

The return value  $y(n)$  is defined by the formula for a sparse FIR filter:

$$y(n) = \sum_{i=0}^{nzTapsLen-1} pNZTaps(i) \cdot x(n - pNZTapPos(i)), \quad 0 \leq n < len$$

After the function has performed calculations, it updates the delay line values stored in the state.

Example 6-9 below shows how to use the sparse FIR filter functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>len</code> is less or equal to 0.

## Example 6-9 Using the Sparse FIR Filter Functions

```
int buflen;
Ipp8u *buf;

int nzTapsLen = 5; //number of non-zero taps
Ipp32f nzTaps [] = {0.5, 0.4, 0.3, 0.2, 0.1}; //non-zero taps values
Ipp32s nzTapsPos[] = {0, 10, 20, 30, 40}; //non-zero tap positions
IppsFIRSparseState_32f* firState;
Ipp32f *src, *dst;

/* ..... */

ippsFIRSparseGetStateSize_32f(nzTapsLen, nzTapsPos [nzTapsLen - 1], &buflen);
buf = ippsMalloc_8u(buflen);

ippsFIRSparseInit_32f(&firState, nzTaps, nzTapsPos, nzTapsLen, NULL, buf);

/* .... initializing src somehow .... */

ippsFIRSparse_32f(src, dst, len, firState);

/*dst[i]=src[i]*0.5 + src[i-10]*0.4 + src[i-20]*0.3 + src[i-30]*0.2 + src[i-40]*0.1 */

/* ..... */

ippsFree(buf);
```

## FIR Filter Coefficient Generating Functions

The functions described in this section compute coefficients (tap values) for different FIR filters by windowing the ideal infinite filter coefficients.

## FIRGenLowpass

*Computes lowpass FIR filter coefficients.*

---

### Syntax

```
IppStatus ippsFIRGenLowpass_64f(Ipp64f rFreq, Ipp64f* taps, int tapsLen,
IppWinType winType, IppBool doNormal);
```

### Parameters

<i>rFreq</i>	Normalized cutoff frequency, must be in the range (0, 0.5).
<i>pTaps</i>	Pointer to the array where computed tap values are stored. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values; must be equal or greater than 5.
<i>winType</i>	Specifies what type of window is used in computations. The <i>winType</i> must have one of the following values: <i>ippWinBartlett</i> Bartlett window; <i>ippWinBlackman</i> Blackman window; <i>ippWinHamming</i> Hamming window; <i>ippWinHann</i> Hann window.
<i>doNormal</i>	Specifies normalized or non-normalized sequence of the filter coefficients is computed. The <i>doNormal</i> must have one of the following values: <i>ippTrue</i> The function computes normalized sequence of coefficients. <i>ippFalse</i> The function computes non-normalized sequence of coefficients.

## Description

The function `ippsFIRGenLowpass` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for lowpass FIR filter with the cutoff frequency *rFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see [Windowing Functions](#). The computed coefficients are stored in the array *pTaps*.

Example 6-10 below shows how the function `ippsFIRGenLowpass` can be used.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTaps</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when the <i>tapsLen</i> is less than 5, or <i>rFreq</i> is out of range.

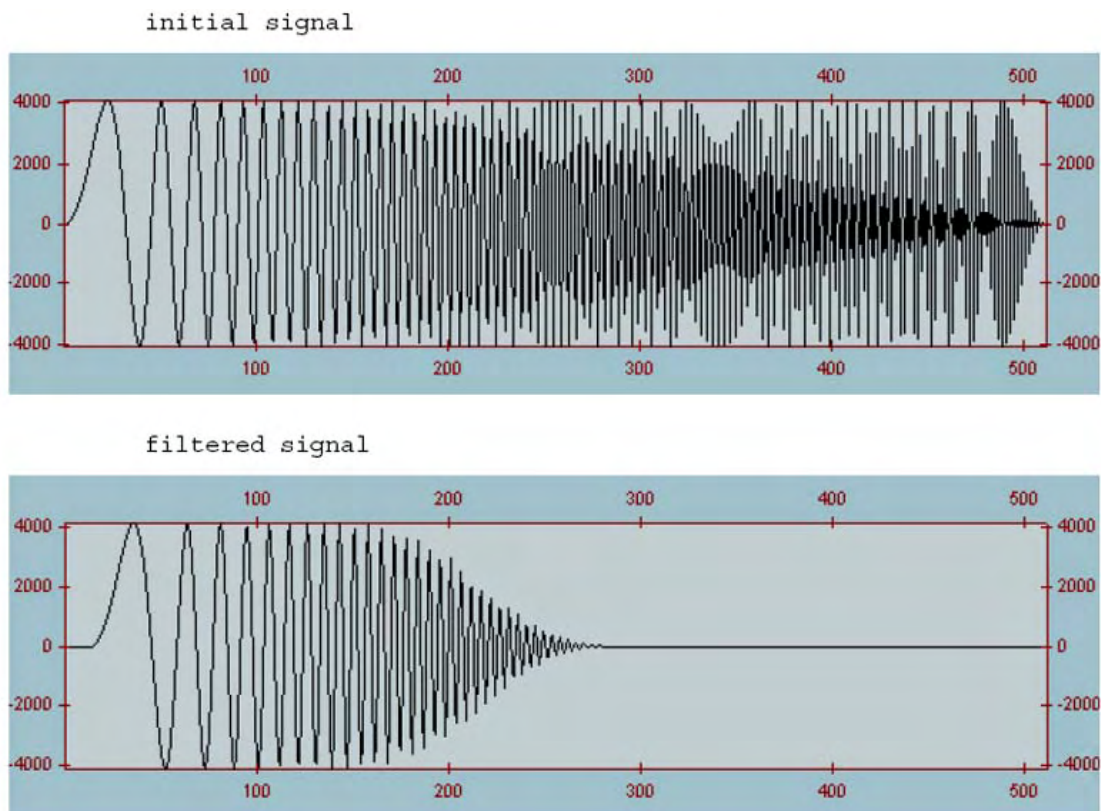
## Example 6-10 Using the Function `ippsFIRGenLowpass`

```
void func_firgenlowpass()  
{  
  
    int len = 512;  
    Ipp64f pDst[512];  
    Ipp64f magn = 4095;  
    Ipp64f rFreq = 0.2;  
    int tapslen = 27;  
    int numIters = 512;
```



```
IppsFIRState_64f* pState;
IppStatus st;
Ipp64f* FIRDst = ippsMalloc_64f(512*sizeof(Ipp64f));
Ipp64f* taps = ippsMalloc_64f(tapslen*sizeof(Ipp64f));
Ipp64f* pDL = ippsMalloc_64f(tapslen*sizeof(Ipp64f));
ippsZero_64f(pDL,tapslen);
//generate source vector
ippsVectorJaehne_64f(pDst, len, magn);// create a Jaehne vector
//computes tapsLen coefficients for lowpass FIR filter
ippsFIRGenLowpass_64f(rFreq, taps, tapslen, ippWinBartlett, ippTrue);
ippsFIRInitAlloc_64f(&pState, taps,tapslen, pDL);
//filter an input vector
ippsFIR_64f(pDst, FIRDst, numIters, pState);
}
```

Result:



## FIRGenHighpass

*Computes highpass FIR filter coefficients.*

---

### Syntax

```
IppStatus ippsFIRGenHighpass_64f(Ipp64f rFreq, Ipp64f* taps, int tapsLen,
IppWinType winType, IppBool doNormal);
```

### Parameters

*rFreq* Normalized cutoff frequency, must be in the range (0, 0.5).

<i>pTaps</i>	Pointer to the array where computed tap values are stored. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values; must be equal or greater than 5.
<i>winType</i>	Specifies what type of window is used in computations. The <i>winType</i> must have one of the following values: <code>ippWinBartlett</code> Bartlett window; <code>ippWinBlackman</code> Blackman window; <code>ippWinHamming</code> Hamming window; <code>ippWinHann</code> Hann window.
<i>doNormal</i>	Specifies normalized or non-normalized sequence of the filter coefficients is computed. The <i>doNormal</i> must have one of the following values: <code>ippTrue</code> The function computes normalized sequence of coefficients. <code>ippFalse</code> The function computes non-normalized sequence of coefficients.

## Description

The function `ippsFIRGenHighpass` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for highpass FIR filter the cutoff frequency *rFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see [Windowing Functions](#). The computed coefficients are stored in the array *pTaps*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTaps</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when the <i>tapsLen</i> is less than 5, or <i>rFreq</i> is out of the range.

## FIRGenBandpass

*Computes bandpass FIR filter coefficients.*

---

### Syntax

```
IppStatus ippSFIRGenBandpass_64f(Ipp64f rLowFreq, Ipp64f rHighFreq, Ipp64f*
pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

### Parameters

<i>rLowFreq</i>	Normalized low cutoff frequency, must be in the range (0, 0.5) and less than <i>rHighFreq</i> .
<i>rHighFreq</i>	Normalized high cutoff frequency, must be in the range (0, 0.5) and greater than <i>rLowFreq</i> .
<i>pTaps</i>	Pointer to the array where computed tap values are stored. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values; should be equal or greater than 5.
<i>winType</i>	Specifies what type of window is used in computations. The <i>winType</i> must have one of the following values: <i>ippWinBartlett</i> Bartlett window; <i>ippWinBlackman</i> Blackman window; <i>ippWinHamming</i> Hamming window; <i>ippWinHann</i> Hann window.
<i>doNormal</i>	Specifies normalized or non-normalized sequence of the filter coefficients is computed. The <i>doNormal</i> must have one of the following values: <i>ippTrue</i> The function computes normalized sequence of coefficients. <i>ippFalse</i> The function computes non-normalized sequence of coefficients.

## Description

The function `ippsFIRGenBandpass` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for bandpass FIR filter with the cutoff frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see [Windowing Functions](#). The computed coefficients are stored in the array *pTaps*.

Example 6-11 below shows how the function `ippsFIRGenBandpass` can be used.

## Return Values

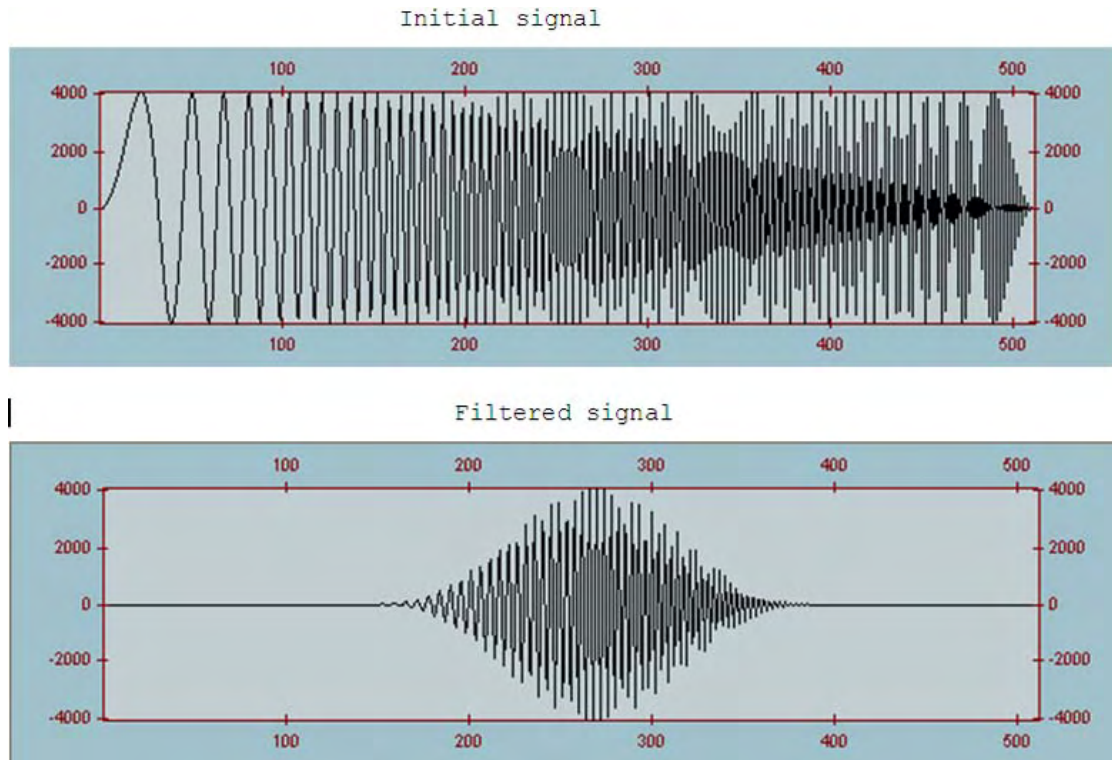
<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTaps</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when the <i>tapsLen</i> is less than 5, or <i>rLowFreq</i> is greater than or equal to <i>rHighFreq</i> , or one of the frequency parameters <i>rLowFreq</i> and <i>rHighFreq</i> is out of the range.

## Example 6-11 Using the Function `ippsFIRGenBandpass`

```
void func_BandPass()
{
    int len = 512;
    Ipp64f pDst[512];
    Ipp64f magn = 4095;
    Ipp64f rLowFreq = 0.2;
    Ipp64f rHighFreq = 0.3;
    int tapslen = 27;
    int numIters = 512;
```

```
IppsFIRState_64f* pState;
IppStatus st;
Ipp64f* FIRDst = ippsMalloc_64f(512*sizeof(Ipp64f));
Ipp64f* taps = ippsMalloc_64f(tapslen*sizeof(Ipp64f));
Ipp64f* pDL = ippsMalloc_64f(tapslen*sizeof(Ipp64f));

ippsZero_64f(pDL,tapslen);
ippsVectorJaehne_64f(pDst, len, magn);/// //generate source vector
//computes tapsLen coefficients for bandstop FIR filter
ippsFIRGenBandpass_64f(rLowFreq, rHighFreq, taps, tapslen, ippWinHamming, ippTrue);
////filter an input vector
ippsFIRInitAlloc_64f(&pState, taps,tapslen, pDL);
ippsFIR_64f(pDst, FIRDst, numIters, pState);
}
Result:
```



## FIRGenBandstop

*Computes bandstop FIR filter coefficients.*

### Syntax

```
IppStatus ippsFIRGenBandstop_64f(Ipp64f rLowFreq, Ipp64f rHighFreq, Ipp64f*
pTaps, int tapsLen, IppWinType winType, IppBool doNormal);
```

### Parameters

*rLowFreq* Normalized low cutoff frequency, must be in the range (0, 0.5) and less than *rHighFreq*.

<i>rHighFreq</i>	Normalized high cutoff frequency, must be in the range (0, 0.5) and greater than <i>rLowFreq</i> .
<i>pTaps</i>	Pointer to the array where computed tap values are stored. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values, must be equal or greater than 5.
<i>winType</i>	Specifies what type of window is used in computations. The <i>winType</i> must have one of the following values: ippWinBartlett Bartlett window; ippWinBlackman Blackman window; ippWinHamming Hamming window; ippWinHann Hann window.
<i>doNormal</i>	Specifies normalized or non-normalized sequence of the filter coefficients is computed. The <i>doNormal</i> must have one of the following values: ippTrue The function computes normalized sequence of coefficients. ippFalse The function computes non-normalized sequence of coefficients.

## Description

The function `ippsFIRGenBandstop` is declared in the `ipps.h` file. This function computes *tapsLen* coefficients for bandstop FIR filter with the cutoff frequencies *rLowFreq* and *rHighFreq* by windowing the ideal infinite filter coefficients. The parameter *winType* specifies the type of the window. For more information on window types used by the function, see [Windowing Functions](#). The computed coefficients are stored in the array *pTaps*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTaps</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when the <i>tapsLen</i> is less than 5, or <i>rLowFreq</i> is greater than or equal to <i>rHighFreq</i> , or one of the frequency parameters <i>rLowFreq</i> and <i>rHighFreq</i> is out of the range.



## Single-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a single-rate FIR LMS filter
- get and set the delay line values
- get the filter coefficients (taps) values
- perform filtering
- free dynamic memory allocated for the functions state

To use the single-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call `ippsFIRLMSInitAlloc` to allocate memory and initialize a single-rate FIR LMS filter.
2. Call `ippsFIRLMSOne_Direct` to make one iteration of FIR filter taps fitting with one input sample and/or call `ippsFIRLMS` to fit taps with a block of consecutive input samples.
3. Call `ippsFIRLMSMRGetTaps` to get the filter coefficients (taps). Call `ippsFIRLMSMRGetDlyLine` and `ippsFIRLMSMRSetDlyLine` to get and set the values in the delay line.
4. Call `ippsFIRLMSMRFree` to release dynamic memory associated with the FIR LMS filter.

## FIRLMSInitAlloc

*Allocates memory and initializes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.*

---

### Syntax

```
IppStatus ippsFIRLMSInitAlloc_32f(IppsFIRLMSState_32f** ppState, const Ipp32f* pTaps, int tapsLen, const Ipp32f* pDlyLine, int dlyLineIndex);
```

```
IppStatus ippsFIRLMSInitAlloc32f_16s(IppsFIRLMSState32f_16s** ppState, const Ipp32f* pTaps, int tapsLen, const Ipp16s* pDlyLine, int dlyLineIndex);
```

### Parameters

<i>pTaps</i>	Pointer to the multi-rate FIR LMS filter state structure to be closed.
<i>tapsLen</i>	Number of elements in the array containing the tap values.

<i>pDlyLine</i>	Pointer to the array holding the delay line values. The number of elements in the array is $2 * tapsLen$ .
<i>dlyLineIndex</i>	Current index of the delay line.
<i>ppState</i>	Pointer to the pointer to the state structure.

## Description

The function `ippsFIRLMSInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a single-rate FIR LMS filter state. The function `ippsFIRLMSInitAlloc` copies the taps from the *tapsLen*-length array *pTaps* into the state structure *ppState*. The  $(2 * tapsLen)$ -length array *pDlyLine* specifies the delay line values. The current index of the delay line *pDlyLine* is defined by *dlyLineIndex*. If the pointer to the array *pDlyLine* or *pTaps* is `NULL`, then the corresponding values of the state structure are initialized to 0.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSFree

*Closes an adaptive FIR filter that uses the least mean squares (LMS) algorithm.*

---

### Syntax

```
IppStatus ippsFIRLMSFree_32f(IppsFIRLMSState_32f* pState);
IppStatus ippsFIRLMSFree32f_16s(IppsFIRLMSState32f_16s* pState);
```

### Parameters

<i>pState</i>	Pointer to the FIR LMS filter state structure to be closed.
---------------	---

### Description

The function `ippsFIRLMSFree` is declared in the `ipps.h` file. This function closes the FIR LMS filter state by freeing all memory associated with a filter state created by the function `ippsFIRLMSInitAlloc`. Call `ippsFIRLMSFree` after filtering is completed.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIRLMSGetTaps

*Retrieves the tap values from the FIR LMS filter.*

### Syntax

```
IppStatus ippFIRLMSGetTaps_32f(const IppsFIRLMSState_32f* pState, Ipp32f* pOutTaps);  
  
IppStatus ippFIRLMSGetTaps32f_16s(const IppsFIRLMSState32f_16s* pState, Ipp32f* pOutTaps);
```

### Parameters

*pState* Pointer to the FIR LMS filter state structure.  
*pOutTaps* Pointer to the array holding copies of the taps.

### Description

The function `ippFIRLMSGetTaps` is declared in the `ipps.h` file. This function copies the taps from the state structure *pState* to the *tapsLen*-length array *pOutTaps*. To set new taps in the state structure, create a new state using the function `ippFIRLMSInitAlloc`.

Before calling the function `ippFIRLMSGetTaps` the filter state structure must be initialized by the function `ippFIRLMSInitAlloc`.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIRLMSGetDlyLine

*Retrieves the delay line contents from the FIR LMS filter.*

---

### Syntax

```
IppStatus ippsFIRLMSGetDlyLine_32f(const IppsFIRLMSState_32f* pState, Ipp32f*
pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIRLMSGetDlyLine32f_16s(const IppsFIRLMSState32f_16s* pState,
Ipp16s* pDlyLine, int* pDlyLineIndex);
```

### Parameters

<i>pState</i>	Pointer to the FIR LMS filter state structure.
<i>pDlyLine</i>	Pointer to the <i>tapsLen</i> -length array holding the delay line values.
<i>pDlyLineIndex</i>	Pointer to the array to store the current delay line index copied from the filter state structure.

### Description

The function `ippsFIRLMSGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pDlyLine* and *pDlyLineIndex*, respectively.

Before calling the function `ippsFIRLMSGetDlyLine` the filter state structure must be initialized by the function `ippsFIRLMSInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSSetDlyLine

Sets the delay line contents in the FIR LMS filter.

### Syntax

```
IppStatus ippsFIRLMSSetDlyLine_32f(const IppsFIRLMSSState_32f* pState, Ipp32f*
pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIRLMSSetDlyLine32f_16s(const IppsFIRLMSSState32f_16s* pState,
Ipp16s* pDlyLine, int* pDlyLineIndex);

IppStatus ippsFIRLMSSetDlyLine_32f(IppsFIRLMSSState_32f* pState, const Ipp32f*
pDlyLine, int dlyLineIndex);

IppStatus ippsFIRLMSSetDlyLine32f_16s(IppsFIRLMSSState32f_16s* pState, const
Ipp16s* pDlyLine, int dlyLineIndex);
```

### Parameters

<i>pState</i>	Pointer to the FIR LMS filter state structure.
<i>pDlyLine</i>	Pointer to the <i>tapsLen</i> -length array holding the delay line values.
<i>dlyLineIndex</i>	Initial index of the delay line to be stored in the filter state structure <i>pState</i> .

### Description

The function `ippsFIRLMSSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from *pDlyLine*, and the current delay line index from *dlyLineIndex*, and stores them into the state structure *pState*.

Before calling the function `ippsFIRLMSSetDlyLine` the filter state structure must be initialized by the function `ippsFIRLMSSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMS

*Filters a vector through the FIR LMS filter.*

---

### Syntax

```
IppStatus ippsFIRLMS_32f(const Ipp32f* pSrc, const Ipp32f* pRef, Ipp32f*
pDst, int len, float mu, IppsFIRLMSState_32f* pState);
```

```
IppStatus ippsFIRLMS32f_16s(const Ipp16s* pSrc, const Ipp16s* pRef, Ipp16s*
pDst, int len, float mu, IppsFIRLMSState32f_16s* pState);
```

### Parameters

<i>pState</i>	Pointer to the FIR LMS filter state structure.
<i>pSrc</i>	Pointer to the source vector .
<i>pRef</i>	Pointer to the reference signal
<i>pDst</i>	Pointer to the output signal
<i>len</i>	Number of elements in the vector.
<i>mu</i>	Adaptation step.

### Description

The function `ippsFIRLMS` is declared in the `ipps.h` file. This function filters a source vector *pSrc* using an adaptive FIR LMS filter.

Each of *len* iterations performed by the function consists of two main procedures. First, `ippsLMS` filters the current element of the source vector *pSrc* and stores the result in *pDst*. Next, the function updates the current taps using the reference signal *pRef*, the computed result signal *pDst*, and the adaptation step *mu*.

The filtering procedure can be described as a FIR filter operation:

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n-i)$$

Here the input sample to be filtered is denoted by  $x(n)$ , the taps are denoted by  $h(i)$ , and  $y(n)$  is the return value.

The function updates the filter coefficients that are stored in the filter state structure *pState*. Updated filter coefficients are defined as  $h_{n+1}(i) = h_n(i) + 2 * \mu * errVal * x(n-i)$ ,

where  $h_{n+1}(i)$  denotes new taps,  $h_n(i)$  denotes initial taps, *mu* and *errVal* are the adaptation step and adaptation error value, respectively. An adaptation error value *errVal* is computed inside the function as the difference between the output and reference signals.

Before using `ippsFIRLMS`, initialize the *pState* structure by calling the function `ippsFIRLM-  
SInitAlloc`.

Code example 6-12 below demonstrates how the use of the functions `ippsFIRLMS_32f` to filter a signal sample.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## Example 6-12 Filtering with the Function ippsFIRLMS

```
IppStatus firlms(void) {
    IppStatus st;
    Ipp32f taps = 0, x[LEN], y[LEN], mu = 0.03f;
    IppsFIRLMSState_32f* ctx;
    int i;
    /// no taps and no delay line from outside
    ippsFIRLMSInitAlloc_32f( &ctx, 0, 1, 0, 0 );
    /// make a const signal of amplitude 1 and noise it
    for(i=0; i<LEN; ++i) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
    st = ippsFIRLMS_32f( x, x+1, y, LEN-1, mu, ctx);
    /// get FIR LMS tap, it must be near to 1
    ippsFIRLMSGetTaps_32f(ctx, &taps);
    ippsFIRLMSFree_32f(ctx);
    printf_32f("FIR LMS tap fitted =", &taps, 1, st);
    return st;
}
```

Output:

```
FIR LMS tap adapted = 0.986842
```

## FIRLMSOne\_Direct

*Filters a single sample through a FIR LMS filter.*

---

### Syntax

```
IppStatus ippsFIRLMSOne_Direct_32f(Ipp32f src, Ipp32f refVal, Ipp32f* pDstVal,
Ipp32f* pTapsInv, int tapsLen, float mu, Ipp32f* pDlyLine, int*
pDlyLineIndex);

IppStatus ippsFIRLMSOne_Direct32f_16s(Ipp16s src, Ipp16s refVal, Ipp16s*
pDstVal, Ipp32f* pTapsInv, int tapsLen, float mu, Ipp16s* pDlyLine, int*
pDlyLineIndex);
```



```
IppStatus ippsFIRLMSOne_DirectQ15_16s(Ipp16s src, Ipp16s refVal, Ipp16s*
pDstVal, Ipp32s* pTapsInv, int tapsLen, int muQ15, Ipp16s* pDlyLine, int*
pDlyLineIndex);
```

## Parameters

<i>src</i>	Input sample to be filtered.
<i>pDstVal</i>	Pointer to the output sample.
<i>refVal</i>	Reference signal sample.
<i>pTapsInv</i>	Pointer to the array containing the FIR filter taps to be adapted. The tap values are stored in the array in the inverse order.
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>pDlyLine</i>	Pointer to the array holding the delay line values.
<i>pDlyLineIndex</i>	Pointer to the current index of the delay line.
<i>mu</i>	Adaptation step.
<i>muQ15</i>	Integer version adaptation step.

## Description

The function `ippsFIRLMSOne_Direct` is declared in the `ipps.h` file. This function performs directly a single iteration of FIR filter taps adaptation. The *tapsLen*-length array *pTapsInv* contains the FIR filter taps in the inverse order. The  $(2 * \textit{tapsLen})$ -length array *pDlyLine* specifies the delay line values. The *pDlyLineIndex* array specifies the current index of the delay line. The output signal is stored in *pDstVal*.




---

**NOTE.** The adaptation error value can be computed as follows:  $\textit{err}[n] = \textit{refVal}[n] - *pDstVal$ .

---

The function `ippsFIRLMSOne_Direct` performs a single iteration of FIR filter taps adaptation with the *mu* step value. The taps are floating-point numbers.

Set the taps to zero or to values close to calculated to speed up the process.

The function `ippsLMSOne_Direct` is to be called within cycle with the number of iterations equal to the number of input samples. You can decide what data is to be saved as a result of adaptation: either the filtered output signal or the adaptation error.

The function `ippsFIRLMSOne_DirectQ15` performs a single iteration of FIR filter taps adaptation with the `muQ15` step value. The taps are integer numbers. The adaptation step `muQ15` can be computed as follows:

```
muQ15 = (int)(mu *(1<<15)+0.5f)
```

Example 6-13 illustrates the use of the function `ippsFIRLMSOne_Direct` to adapt the FIR filter taps. After adaptation the taps are close to 1.0.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>tapsLen</code> is less or equal to 0.

## Example 6-13 Using the Function `ippsLMSOne_Direct`

```
IppStatus firlmsone(void) {
#define LEN 200

    IppStatus st = ippStsNoErr;
    Ipp32f taps = 0, dly[2] = {0};
    Ipp32f x[LEN], y[LEN], mu = 0.05f;
    int i, indx = 0;

    /// make a const signal of amplitude 1 and noise it
    for( i=0; i<LEN; ++i ) x[i] = 0.4f * rand()/RAND_MAX + 0.8f;
    for( i=0; i<LEN-1 && ippStsNoErr==st; ++i )
        st=ippsFIRLMSOne_Direct_32f( x[i], x[i+1], y+1+i, &taps, 1, mu,
            dly, &indx );
    printf_32f("FIRLMSOne tap adapted =", &taps, 1, st );
    return st;
}
```

Output:

```
FIRLMSOne tap adapted = 0.993872
```

## Multi-Rate FIR LMS Filter Functions

The functions described in this section perform the following tasks:

- initialize a multi-rate FIR LMS filter
- get and set the delay line values
- get and set the filter coefficients (taps) values
- set the adaptation step value
- perform filtering
- update the filter coefficients using the result of the filter operation
- free dynamic memory allocated for the functions state

To use the multi-rate FIR LMS adaptive filter functions, follow this general scheme:

1. Call `ippsFIRLMSMRInitAlloc` to initialize a multi-rate FIR LMS filter.
2. Call `ippsFIRLMSMRPutVal` a required number of times to place the input values in the delay line.
3. Call `ippsFIRLMSMROne` to filter the samples in the delay line.
4. Call `ippsFIRLMSMRUpdateTaps` to update the taps using the value of adaptation error that is based on comparison between filtered and reference signals.
5. Call `ippsFIRLMSMRGetTaps` and `ippsFIRLMSMRSetTaps` to get and set the filter coefficients (taps). Call `ippsFIRLMSMRGetDlyLine` and `ippsFIRLMSMRSetDlyLine` to get and set the values in the delay line.
6. Call `ippsFIRLMSMRFree` to release dynamic memory associated with the FIR LMS filter.

## FIRLMSMRInitAlloc

*Allocates memory and initializes an adaptive multi-rate FIR filter that uses the least mean squares (LMS) algorithm.*

### Syntax

```
IpStatus ippsFIRLMSMRInitAlloc32s_16s(IppsFIRLMSMRState32s_16s** ppState,
const Ipp32s* pTaps, int tapsLen, const Ipp16s* pDlyLine, int dlyLineIndex,
int dlyStep, int updateDly, int mu);
```

```
IpStatus ippsFIRLMSMRInitAlloc32sc_16sc(IppsFIRLMSMRState32sc_16sc** ppState,
const Ipp32sc* pTaps, int tapsLen, const Ipp16sc* pDlyLine, int dlyLineIndex,
int dlyStep, int updateDly, int mu);
```

## Parameters

<i>pTaps</i>	Pointer to the array containing the tap values. The number of elements in the array is <i>tapsLen</i> .
<i>tapsLen</i>	Number of elements in the array containing the tap values.
<i>pDlyLine</i>	Pointer to the array holding the delay line values. The number of elements in the array is <i>tapsLen*dlyStep</i> + <i>updateDly</i> .
<i>dlyLineIndex</i>	Current index of the delay line.
<i>dlyStep</i>	Multi-rate down factor applied to delay line values.
<i>updateDly</i>	Value of adaptation delay in samples.
<i>mu</i>	Adaptation step.
<i>ppState</i>	Pointer to the pointer to the filter state structure.

## Description

The function `ippsFIRLMSMRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes a multi-rate FIR LMS filter state. The function copies the filter coefficients from *tapsLen*-length array *pTaps* into the state structure *ppState*. The array *pDlyLine* specifies the delay line values. The structure is initialized by the downsampling factor over the delay line *dlyStep*, by the adaptation delay value *updateDly*, and by the adaptation step value *mu*. The function returns the pointer in the output parameter *ppState* and also the operation status value.

The function uses copies of the tap values during the filtering procedure. The initial values passed to the function may be obtained from the previous filtering procedure. If the pointer *pTaps* is `NULL`, then the filter coefficient values are set to zero.

Copies of the input samples are used in the filtering procedure. Values in the delay line represent data in the same format as the input data to be filtered. If the pointer *pDlyLine* is `NULL`, then the filter delay line values are set to zero.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppState</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSMRFree

*Closes an adaptive multi-rate FIR filter that uses the least mean squares algorithm.*

---

### Syntax

```
IppStatus ippsFIRLMSMRFree32s_16s(IppsFIRLMSMRState32s_16s* pState);
IppStatus ippsFIRLMSMRFree32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState);
```

### Parameters

*pState*                      Pointer to the multi-rate FIR LMS filter state structure to be closed.

### Description

The function `ippsFIRLMSMRFree` is declared in the `ipps.h` file. This function closes the multi-rate FIR LMS filter state by freeing all memory associated with a filter state created by the function [ippsFIRLMSMRInitAlloc](#). Call `ippsFIRLMSMRFree` after filtering is completed.

### Return Values

`ippStsNoErr`                Indicates no error.  
`ippStsNullPtrErr`        Indicates an error when the *pState* pointer is NULL.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIRLMSMRSetMu

*Sets the adaptation step.*

---

### Syntax

```
IppStatus ippsFIRLMSMRSetMu32s_16s(IppsFIRLMSMRState32s_16s* pState, const
int mu);
IppStatus ippsFIRLMSMRSetMu32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
const int mu);
```

### Parameters

*mu*                              New adaptation step

*pState* Pointer to the filter state structure.

## Description

The function `ippsFIRLMSMRSetMu` is declared in the `ipps.h` file. This function updates the adaptation step stored in *pState* with the new value *mu*.

Before calling the function `ippsFIRLMSMRSetMu` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the *pState* pointer is NULL.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIRLMSMRUpdateTaps

*Updates the filter coefficients using the adaptation error value.*

---

## Syntax

```

IppStatus ippsFIRLMSMRUpdateTaps32s_16s(Ipp32s errVal,
IppsFIRLMSMRState32s_16s* pState);

IppStatus ippsFIRLMSMRUpdateTaps32sc_16sc(Ipp32scerrVal,
IppsFIRLMSMRState32sc_16sc* pState);

```

## Parameters

*pState* Pointer to the filter state structure.  
*errVal* Adaptation error value.

## Description

The function `ippsFIRLMSMRUpdateTaps` is declared in the `ipps.h` file. This function updates the filter coefficients that are stored in the filter state structure *pState*. It is assumed that the filter operation was performed and the adaptation error value *errVal* was computed before calling this function. The adaptation error value is computed outside the function as the difference between output and reference signals. Updated filter coefficients are defined as  $h_{n+1}(i) = h_n(i) + mu * errVal * x(n - (i * dlyStep) - updateDly)$ ,

where  $h_{n+1}(i)$  denotes new taps,  $h_n(i)$  denotes initial taps, and  $\mu$ ,  $errVal$  and  $updateDly$  are the adaptation step, adaptation error value and adaptation delay, respectively. Before calling the function `ippsFIRLMSMRUpdateTaps` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSMRGetTaps

*Retrieves tap values from the multi-rate FIR LMS filter.*

---

### Syntax

```
IppStatus ippsFIRLMSMRGetTaps32s_16s(IppsFIRLMSMRState32s_16s* pState, Ipp32s*
pOutTaps);

IppStatus ippsFIRLMSMRGetTaps32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
Ipp32sc* pOutTaps);
```

### Parameters

<code>pState</code>	Pointer to the filter state structure.
<code>pOutTaps</code>	Pointer to the array holding copies of the taps.

### Description

The function `ippsFIRLMSMRGetTaps` is declared in the `ipps.h` file. This function copies the values of the filter coefficients stored in the filter state structure `pState` to the array pointed by the `pOutTaps` pointer.

Before calling the function `ippsFIRLMSMRGetTaps` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.

`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIRLMSMRSetTaps

*Sets tap values in the multi-rate FIR LMS filter.*

---

### Syntax

```

IppStatus ippSFIRLMSMRSetTaps32s_16s(IppsFIRLMSMRState32s_16s* pState, const
Ipp32s* pInTaps);

IppStatus ippSFIRLMSMRSetTaps32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
const Ipp32sc* pInTaps);

```

### Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pInTaps</i>	Pointer to the array holding new tap values.

### Description

The function `ippSFIRLMSMRSetTaps` is declared in the `ipps.h` file. This function sets the filter coefficients stored in the filter state structure *pState* to the new values stored in an array pointed by the *pInTaps* pointer. If the pointer is `NULL`, then the filter coefficients values are set to zero.

Before calling the function `ippSFIRLMSMRSetTaps` the filter state structure must be initialized by the function `ippSFIRLMSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.



## FIRLMSMRGetTapsPointer

*Returns the pointer to the filter coefficients.*

### Syntax

```
IppStatus ippsFIRLMSMRGetTapsPointer32s_16s(IppsFIRLMSMRState32s_16s* pState,
Ipp32s** ppTaps);
```

```
IppStatus ippsFIRLMSMRGetTapsPointer32sc_16sc(IppsFIRLMSMRState32sc_16sc*
pState, Ipp32sc** ppTaps);
```

### Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>ppTaps</i>	Pointer to the variable that contains the pointer to the tap values.

### Description

The function `ippsFIRLMSMRGetTapsPointer` is declared in the `ipps.h` file. This function writes the pointer to the filter coefficients stored in the filter state structure *pState* to the variable pointed by *ppTaps*.



**CAUTION.** To get the pointer to tap values directly is faster than to copy them using the the function `ippsFIRLMSMRGetTaps` function, but this operation may be error-prone.

Before calling the function `ippsFIRLMSMRGetTapsPointer` the filter state structure must be initialized by the the function `ippsFIRLMSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSMRGetDlyLine

*Retrieves the delay line contents from the multi-rate FIR LMS filter state.*

---

### Syntax

```
IppStatus ippsFIRLMSMRGetDlyLine32s_16s(IppsFIRLMSMRState32s_16s* pState,
Ipp16s* pOutDlyLine, int* pOutDlyLineIndex);

IppStatus ippsFIRLMSMRGetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
Ipp16sc* pOutDlyLine, int* pOutDlyLineIndex);
```

### Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pOutDlyLine</i>	Pointer to the array containing the copies of delay line values. The number of elements in the array is <i>tapsLen*dlyStep+updateDly</i> .
<i>pOutDlyLineIndex</i>	Pointer to the array to store the current delay line index copied from <i>pState</i> .

### Description

The function `ippsFIRLMSMRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values and the current delay line index from the state structure *pState*, and stores them into *pOutDlyLine* and *pOutDlyLineIndex*, respectively.

Before calling the function `ippsFIRLMSMRGetDlyLine` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSMRSetDlyLine

*Gets and sets the delay line contents of a multi-rate FIR LMS filter state.*

---

### Syntax

```
IppStatus ippsFIRLMSMRSetDlyLine32s_16s(IppsFIRLMSMRState32s_16s* pState,
const Ipp16s* pInDlyLine, int dlyLineIndex);
```

```
IppStatus ippsFIRLMSMRSetDlyLine32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
const Ipp16sc* pInDlyLine, int dlyLineIndex);
```

### Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pInDlyLine</i>	Pointer to the array containing the new delay line values. The number of elements in the array is <i>tapsLen*dlyStep+updateDly</i> .
<i>dlyLineIndex</i>	Initial index of the delay line to be stored in <i>pState</i> .

### Description

The function `ippsFIRLMSMRSetDlyLine` is declared in the `ipps.h` file. This function copies the new values stored in the *pInDlyLine* and corresponding delay line index from *dlyLineIndex* and stores them into the state structure *pState*. If the pointer *pInDlyLine* is `NULL`, the delay line values are set to zero.

Before calling the function `ippsFIRLMSMRSetDlyLine` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSMRGetDlyVal

*Gets one delay line values from the specified position.*

---

### Syntax

```
IppStatus ippsFIRLMSMRGetDlyVal32s_16s(IppsFIRLMSMRState32s_16s* pState,
Ipp16s* pOutVal, int index);

IppStatus ippsFIRLMSMRGetDlyVal32sc_16sc(IppsFIRLMSMRState32sc_16sc* pState,
Ipp16sc* pOutVal, int index);
```

### Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pOutVal</i>	Pointer to the copied delay line value.
<i>index</i>	Index of the required delay line value.

### Description

The function `ippsFIRLMSMRGetDlyVal` is declared in the `ipps.h` file. This function copies from the filter state structure *pState* one value to the *pOutVal*. The position of this sample in the delay line is specified by *index* - it means that *index* iterations ago the sample was placed into the delay line.

Before calling the function `ippsFIRLMSMRGetDlyVal` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSMRPutVal

*Places the input value in the delay line.*

---

### Syntax

```

IppStatus ippsFIRLMSMRPutVal32s_16s(Ipp16s val, IppsFIRLMSMRState32s_16s*
pState);

IppStatus ippsFIRLMSMRPutVal32sc_16sc(Ipp16sc val, IppsFIRLMSMRState32sc_16sc*
pState);

```

### Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>val</i>	Value of the input sample.

### Description

The function `ippsFIRLMSMRPutVal` is declared in the `ipps.h` file. This function places the value of the input sample *val* into the delay line, thus preparing the filter with the state structure *pState* for the filtering procedure.

Before calling the function `ippsFIRLMSMRPutVal` the filter state structure must be initialized by the function `ippsFIRLMSMRInitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## FIRLMSMROne

*Filters data placed in the delay line.*

---

### Syntax

```

IppStatus ippsFIRLMSMROne32s_16s(Ipp32s* pDstVal, IppsFIRLMSMRState32s_16s*
pState);

IppStatus ippsFIRLMSMROne32sc_16sc(Ipp32sc* pDstVal,
IppsFIRLMSMRState32sc_16sc* pState);

```

## Parameters

*pState* Pointer to the filter state structure.  
*pDstVal* Pointer to the output signal value.

## Description

The function `FIRLMSMROne` is declared in the `ipps.h` file. This function filters the samples placed in the delay line using the filter coefficients stored in the filter state structure *pState*. The resulting value is placed in the *pDstVal*. The downsampling factor *dlyStep* defines the number of samples that are filtered. The filter coefficients are not updated.

The filtering procedure can be described as a FIR filter operation (here the input sample to be filtered is denoted  $x(n)$ , the taps are denoted  $h(i)$ , and the return value is  $y(n)$ ):

$$y(n) = \sum_{i=0}^{tapsLen-1} h(i) \cdot x(n - (i \cdot dlyStep))$$

Note that the function operates with values stored in the delay line that are copies of the input samples. Before calling the function `ippsFIRLMSMROne`, initialize the filter state by calling the function `ippsFIRLMSMRInitAlloc`.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## FIRLMSMROneVal

*Filters one input value.*

---

### Syntax

```
ippStatus ippsFIRLMSMROneVal32s_16s(ipp16s val, ipp32s* pDstVal,
ippFIRLMSMRState32s_16s* pState);
```

```
IppStatus ippsFIRLMSMROneVal32sc_16sc(Ipp16sc val, Ipp32sc* pDstVal,  
IppFIRLMSMRState32sc_16sc* pState);
```

### Parameters

<i>pState</i>	Pointer to the filter state structure.
<i>pDstVal</i>	Pointer to the output signal value.
<i>val</i>	Value of the input signal sample.

### Description

The function `ippsFIRLMSMROneVal` is declared in the `ipps.h` file. This function places one input sample *val* into the delay line and filters it using the filter coefficients specified in the filter state structure *pState*. The result value is stored in the *pDstVal*.

Before calling the function `ippsFIRLMSMROneVal`, initialize the filter state by calling the function `ippsFIRLMSMRInitAlloc`.

### Return Values

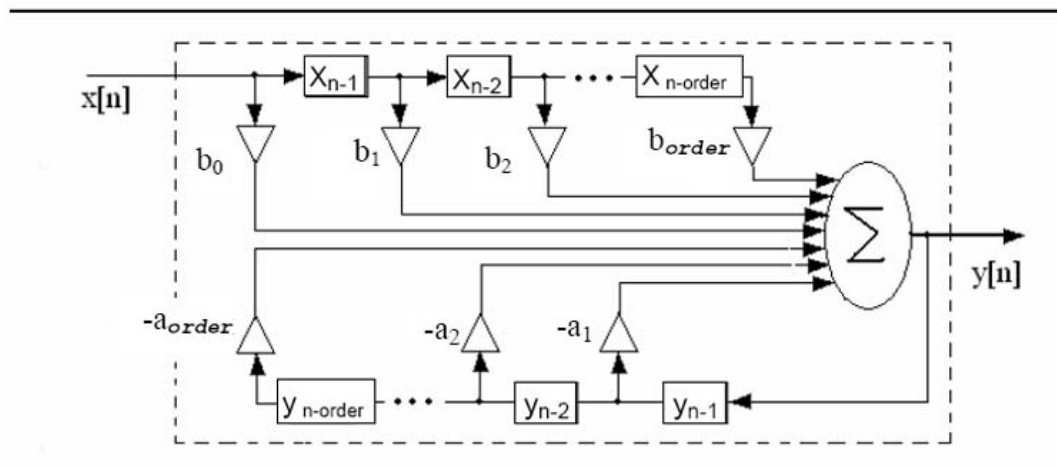
<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## IIR Filter Functions

The functions described in this section initialize an infinite impulse response (IIR) filter and perform filtering. Intel IPP supports two types of filters: arbitrary order filter and biquad filter.

Figure 6-1below shows the structure of an arbitrary order IIR filter.

**Figure 6-1 Structure of an Arbitrary Order Filter**



Here  $x[n]$  is a sample of the input signal,  $y[n]$  is a sample of the output signal,  $order$  is the filter order, and  $b_0, b_1, \dots, b_{order}, a_1, \dots, a_{order}$  are the reduced filter coefficients.

The output signal is computed by the following formula:

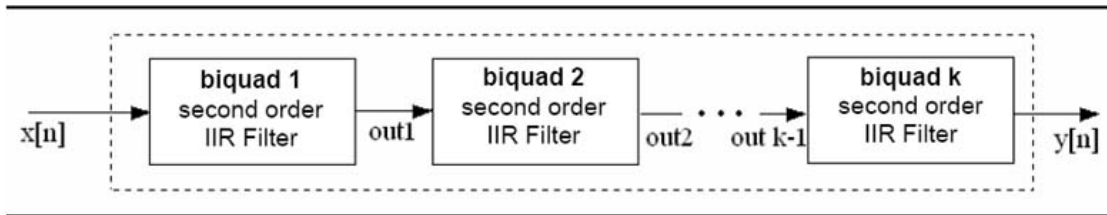
$$y[n] = \sum_{k=0}^{order} b_k \cdot x(n-k) - \sum_{k=1}^{order} a_k \cdot y(n-k)$$

Reduced coefficients are calculated as  $a_k = A_k/A_0$  and  $b_k = B_k/A_0$  where  $A_0, A_1, \dots, A_{order}, B_0, B_1, \dots, B_{order}$  are initial filter coefficients (taps).



A biquad IIR filter is a cascade of second-order filters. Figure 6-2 below illustrates the structure of the biquad filter with  $k$  cascades of second-order filters.

**Figure 6-2 Structure of a BiQuad IIR Filter**



To initialize and use an IIR filter, follow this general scheme:

1. Call `ippsIIRInitAlloc` to allocate memory and initialize the filter as an arbitrary order IIR filter, or call `ippsIIRInitAlloc_BiQuad` to allocate memory and initialize the filter as a cascade of biquads. Or alternatively call `ippsIIRInit` to initialize the filter as an arbitrary order IIR filter in the external buffer, or `ippsIIRInit_BiQuad` to initialize the filter as a cascade of biquads in the external buffer. Size of the buffer can be computed by calling the functions `ippsIIRGetStateSize` or `ippsIIRGetStateSize_BiQuad`, respectively.
2. Call `ippsIIROne` repeatedly to filter a single sample through an IIR filter or call `ippsIIR` to filter consecutive samples at once.
3. Call `ippsIIRGetDlyLine` and `ippsIIRSetDlyLine` to get and set the delay line values in the IIR state structure.
4. Call `ippsIIRSetTaps` to set new tap values in the previously initialized filter state structure.
5. After all filtering is complete, call `ippsIIRFree` to release dynamic memory associated with the filter state structure created by `ippsIIRInitAlloc` or `ippsIIRInitAlloc_BiQuad`.

Alternatively, you may use the direct version of the functions. These functions perform filtering without initializing the filter state structure. All required parameters are directly set in the function.

## IIRInitAlloc

*Allocates memory and initializes the state structure for an arbitrary IIR filter.*

---

### Syntax

#### Case 1: Operation on integer samples

```
IppStatus ippsIIRInitAlloc32s_16s(IppsIIRState32s_16s** ppState, const Ipp32s* pTaps, int order, int tapsFactor, const Ipp32s* pDlyLine);

IppStatus ippsIIRInitAlloc32s_16s32f(IppsIIRState32s_16s** ppState, const Ipp32f* pTaps, int order, const Ipp32s* pDlyLine);

IppStatus ippsIIRInitAlloc32f_16s(IppsIIRState32f_16s** ppState, const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine);

IppStatus ippsIIRInitAlloc64f_16s(IppsIIRState64f_16s** ppState, const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc64f_32s(IppsIIRState64f_32s** ppState, const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_16sc(IppsIIRState32sc_16sc** ppState, const Ipp32sc* pTaps, int order, int tapsFactor, const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_16sc32fc(IppsIIRState32sc_16sc** ppState, const Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32fc_16sc(IppsIIRState32fc_16sc** ppState, const Ipp32fc* pTaps, int order, const Ipp32fc* pDlyLine);

IppStatus ippsIIRInitAlloc64fc_16sc(IppsIIRState64fc_16sc** ppState, const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc64fc_32sc(IppsIIRState64fc_32sc** ppState, const Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);
```

#### Case 2: Operation on floating point samples

```
IppStatus ippsIIRInitAlloc_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps, int order, const Ipp32f* pDlyLine);

IppStatus ippsIIRInitAlloc64f_32f(IppsIIRState64f_32f** ppState, const Ipp64f* pTaps, int order, const Ipp64f* pDlyLine);
```

```

IppStatus ippsIIRInitAlloc_64f(IppsIIRState_64f** ppState, const Ipp64f*
pTaps, int order, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc_32fc(IppsIIRState_32fc** ppState, const Ipp32fc*
pTaps, int order, const Ipp32fc* pDlyLine);

IppStatus ippsIIRInitAlloc64fc_32fc(IppsIIRState64fc_32fc** ppState, const
Ipp64fc* pTaps, int order, const Ipp64fc* pDlyLine);

IppStatus ippsIIRInitAlloc_64fc(IppsIIRState_64fc** ppState, const Ipp64fc*
pTaps, int order, const Ipp64fc* pDlyLine);
    
```

### Parameters

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $2 * (order + 1)$ .
<i>tapsFactor</i>	Scale factor for the taps of integer data type.
<i>order</i>	Order of the IIR filter.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> .
<i>ppState</i>	Pointer to the pointer to the IIR state structure.

### Description

The function `ippsIIRInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes an arbitrary IIR filter state. The initialization functions copy the taps from the array *pTaps* into the state structure *ppState*. The *tapsFactor* is used to scale integer tap values. The array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The  $2 * (order + 1)$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_0, B_1, \dots, B_{order}, A_0, A_1, \dots, A_{order}$$

$$A_0 \neq 0$$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. [Example 6-15](#) shows how to convert floating-point taps into integer data type.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pTaps</i> or <i>ppState</i> is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>order</i> is less than or equal to 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when $A_0$ is equal to 0.

## IIRInitAlloc\_BiQuad

*Allocates memory and initializes the state structure for the biquad IIR filter.*

---

### Syntax

#### Case 1: Operation on integer samples

```

IppStatus ippsIIRInitAlloc32s_BiQuad_16s(IppsIIRState32s_16s** ppState, const
Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine);

IppStatus ippsIIRInitAlloc32s_BiQuad_16s32f(IppsIIRState32s_16s** ppState,
const Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine);

IppStatus ippsIIRInitAlloc32f_BiQuad_16s(IppsIIRState32f_16s** ppState, const
Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);

IppStatus ippsIIRInitAlloc64f_BiQuad_16s(IppsIIRState64f_16s** ppState, const
Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc64f_BiQuad_32s(IppsIIRState64f_32s** ppState, const
Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);

IppStatus ippsIIRInitAlloc64f_BiQuad_DF1_32s(IppsIIRState64f_32s** ppState,
const Ipp64f* pTaps, int numBq, const Ipp32s* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc(IppsIIRState32sc_16sc** ppState,
const Ipp32sc* pTaps, int numBq, int tapsFactor, const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc**
ppState, const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine);

IppStatus ippsIIRInitAlloc32fc_BiQuad_16sc(IppsIIRState32fc_16sc** ppState,
const Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);

```

```
IppStatus ippsIIRInitAlloc64fc_BiQuad_16sc(IppsIIRState64fc_16sc** ppState,
const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc64fc_BiQuad_32sc(IppsIIRState64fc_32sc** ppState,
const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

### Case 2: Operation on floating point samples

```
IppStatus ippsIIRInitAlloc_BiQuad_32f(IppsIIRState_32f** ppState, const
Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc_BiQuad_DF1_32f(IppsIIRState_32f** ppState, const
Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc64f_BiQuad_32f(IppsIIRState64f_32f** ppState, const
Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc_BiQuad_64f(IppsIIRState_64f** ppState, const
Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc_BiQuad_32fc(IppsIIRState_32fc** ppState, const
Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc64fc_BiQuad_32fc(IppsIIRState64fc_32fc** ppState,
const Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

```
IppStatus ippsIIRInitAlloc_BiQuad_64fc(IppsIIRState_64fc** ppState, const
Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine);
```

### Parameters

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $6 \cdot \text{numBq}$ .
<i>tapsFactor</i>	Scale factor for the taps of integer data type.
<i>numBq</i>	Number of cascades of biquads.
<i>pDlyLine</i>	Pointer to the array containing the delay line values.
<i>ppState</i>	Pointer to the pointer to the biquad IIR state structure.

### Description

The function `ippsIIRInitAlloc_BiQuad` is declared in the `ipps.h` file. This function allocates memory and initializes a biquad (BQ) IIR filter state. The initialization functions copy the taps from the array *pTaps* into the state structure *ppState*. The *tapsFactor* is used to scale integer tap values. The array *pDlyLine* specifies the delay line values. The number of elements in the

array *pDlyLine* is  $4 * numBq$  for the function flavor `ippsIIRInitAlloc_BiQuad_DF1`, and  $2 * numBq$  for all other flavors. If the pointer to the array *pDlyLine* is not NULL, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

The function flavor `ippsIIRInitAlloc_BiQuad_DF1` initializes the filter with the biquad section in the direct form I (DF1) [Opp75]. In this case the delay line values are arranged in the array as follows:

$x_{0,-2}, x_{0,-1}, B_{0,2}, y_{0,-2}, y_{0,-1}, x_{1,-2}, x_{1,-1}, y_{1,-2}, y_{1,-1}, \dots, x_{numBq-1,-2}, x_{numBq-1,-1}, y_{numBq-1,-2}, y_{numBq-1,-1}$ .

A biquad IIR filter is defined by a cascade of biquads. The number of cascades of biquads is specified by the *numBq* value. The  $6 * numBq$  -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \dots A_{numBq-1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. [Example 6-15](#) shows how to convert floating-point taps into integer data type.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pTaps</i> or <i>ppState</i> is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>numBq</i> is less than or equal to 0.
<code>ippStsDivByZeroErr</code>	Indicates an error when $A_0$ , $A_{n,0}$ or $B_{n,0}$ is equal to 0.

## IIRFree

*Closes an IIR filter state.*

---

### Syntax

```
IppStatus ippsIIRFree_32f(IppsIIRState_32f* pState);
```

```

IppStatus ippsIIRFree_64f(IppsIIRState_64f* pState);
IppStatus ippsIIRFree_32fc(IppsIIRState_32fc* pState);
IppStatus ippsIIRFree_64fc(IppsIIRState_64fc* pState);
IppStatus ippsIIRFree32s_16s(IppsIIRState32s_16s* pState);
IppStatus ippsIIRFree32sc_16sc(IppsIIRState32sc_16sc* pState);
IppStatus ippsIIRFree32f_16s(IppsIIRState32f_16s* pState);
IppStatus ippsIIRFree32fc_16sc(IppsIIRState32fc_16sc* pState);
IppStatus ippsIIRFree64f_16s(IppsIIRState64f_16s* pState);
IppStatus ippsIIRFree64f_32s(IppsIIRState64f_32s* pState);
IppStatus ippsIIRFree64f_32f(IppsIIRState64f_32f* pState);
IppStatus ippsIIRFree64fc_16sc(IppsIIRState64fc_16sc* pState);
IppStatus ippsIIRFree64fc_32sc(IppsIIRState64fc_32sc* pState);
IppStatus ippsIIRFree64fc_32fc(IppsIIRState64fc_32fc* pState);

```

### Parameters

*pState*                      Pointer to an IIR filter state structure to be closed.

### Description

The function `ippsIIRFree` is declared in the `ipps.h` file. This function closes the IIR filter state by freeing all memory associated with a filter state created by `ippsIIRInitAlloc` or `ippsIIRInitAlloc_BiQuad`. Call `ippsIIRFree` after filtering is completed.

### Return Values

`ippStsNoErr`                Indicates no error.  
`ippStsNullPtrErr`        Indicates an error when the *pState* pointer is NULL.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## IIRInit

*Initializes an arbitrary IIR filter state.*

---

### Syntax

#### Case 1: Operation on integer samples

```

IppStatus ippsIIRInit32s_16s(IppsIIRState32s_16s** ppState, const Ipp32s*
pTaps, int order, int tapsFactor, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32s_16s32f(IppsIIRState32s_16s** ppState, const Ipp32f*
pTaps, int order, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32f_16s(IppsIIRState32f_16s** ppState, const Ipp32f*
pTaps, int order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64f_16s(IppsIIRState64f_16s** ppState, const Ipp64f*
pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64f_32s(IppsIIRState64f_32s** ppState, const Ipp64f*
pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32sc_16sc(IppsIIRState32sc_16sc** ppState, const Ipp32sc*
pTaps, int order, int tapsFactor, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32sc_16sc32fc(IppsIIRState32sc_16sc** ppState, const
Ipp32fc* pTaps, int order, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32fc_16sc(IppsIIRState32fc_16sc** ppState, const Ipp32fc*
pTaps, int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_16sc(IppsIIRState64fc_16sc** ppState, const Ipp64fc*
pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_32sc(IppsIIRState64fc_32sc** ppState, const Ipp64fc*
pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

```

#### Case 2: Operation on floating point samples

```

IppStatus ippsIIRInit_32f(IppsIIRState_32f** ppState, const Ipp32f* pTaps,
int order, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64f_32f(IppsIIRState64f_32f** ppState, const Ipp64f*
pTaps, int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

```



```

IppStatus ippsIIRInit_64f(IppsIIRState_64f** ppState, const Ipp64f* pTaps,
int order, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_32fc(IppsIIRState_32fc** ppState, const Ipp32fc* pTaps,
int order, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_32fc(IppsIIRState64fc_32fc** ppState, const Ipp64fc*
pTaps, int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_64fc(IppsIIRState_64fc** ppState, const Ipp64fc* pTaps,
int order, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
    
```

### Parameters

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $2 \cdot (\text{order} + 1)$ .
<i>tapsFactor</i>	Scale factor for the taps of integer data type.
<i>order</i>	Order of the IIR filter.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> .
<i>ppState</i>	Pointer to the pointer to the arbitrary IIR state structure to be created.
<i>pBuffer</i>	Pointer to the external buffer.

### Description

The function `ippsIIRInit` is declared in the `ipps.h` file. This function initializes an arbitrary IIR filter state in the external buffer. The size of this buffer must be computed previously by calling the function `IIRGetStateSize`. The initialization functions copy the taps from the array *pTaps* into the state structure *pState*. The *tapsFactor* is used to scale integer tap values. The *order*-length array *pDlyLine* specifies the delay line values. If the pointer to the array *pDlyLine* is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The  $2 \cdot (\text{order} + 1)$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_0, B_1, \dots, B_{\text{order}}, A_0, A_1, \dots, A_{\text{order}}$$

$$A_0 \neq 0$$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. [Example 6-15](#) shows how to convert floating-point taps into integer data type.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsDivByZeroErr</code>	Indicates an error when $A_0$ is equal to 0.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>order</i> is less than or equal to 0.

## IIRInit\_BiQuad

*Initializes an IIR filter state.*

---

### Syntax

#### Case 1: Operation on integer samples

```

IppStatus ippSIIRInit32s_BiQuad_16s(IppsIIRState32s_16s** ppState, const
Ipp32s* pTaps, int numBq, int tapsFactor, const Ipp32s* pDlyLine, Ipp8u*
pBuffer);

IppStatus ippSIIRInit32s_BiQuad_16s32f(IppsIIRState32s_16s** ppState, const
Ipp32f* pTaps, int numBq, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippSIIRInit32f_BiQuad_16s(IppsIIRState32f_16s** ppState, const
Ipp32f* pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippSIIRInit64f_BiQuad_16s(IppsIIRState64f_16s** ppState, const
Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippSIIRInit64f_BiQuad_32s(IppsIIRState64f_32s** ppState, const
Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippSIIRInit64f_BiQuad_DF1_32s(IppsIIRState64f_32s** ppState, const
Ipp64f* pTaps, int numBq, const Ipp32s* pDlyLine, Ipp8u* pBuffer);

IppStatus ippSIIRInit32sc_BiQuad_16sc(IppsIIRState32sc_16sc** ppState, const
Ipp32sc* pTaps, int numBq, int tapsFactor, const Ipp32sc* pDlyLine, Ipp8u*
pBuffer);

```

```
IppStatus ippsIIRInit32sc_BiQuad_16sc32fc(IppsIIRState32sc_16sc** ppState,
const Ipp32fc* pTaps, int numBq, const Ipp32sc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit32fc_BiQuad_16sc(IppsIIRState32fc_16sc** ppState, const
Ipp32fc* pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_BiQuad_16sc(IppsIIRState64fc_16sc** ppState, const
Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_BiQuad_32sc(IppsIIRState64fc_32sc** ppState, const
Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

### Case 2: Operation on floating point samples

```
IppStatus ippsIIRInit_BiQuad_32f(IppsIIRState_32f** ppState, const Ipp32f*
pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_DF1_32f(IppsIIRState_32f** ppState, const Ipp32f*
pTaps, int numBq, const Ipp32f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64f_BiQuad_32f(IppsIIRState64f_32f** ppState, const
Ipp64f* pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_64f(IppsIIRState_64f** ppState, const Ipp64f*
pTaps, int numBq, const Ipp64f* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_32fc(IppsIIRState_32fc** ppState, const Ipp32fc*
pTaps, int numBq, const Ipp32fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit64fc_BiQuad_32fc(IppsIIRState64fc_32fc** ppState, const
Ipp64fc* pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);

IppStatus ippsIIRInit_BiQuad_64fc(IppsIIRState_64fc** ppState, const Ipp64fc*
pTaps, int numBq, const Ipp64fc* pDlyLine, Ipp8u* pBuffer);
```

### Parameters

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $6 \cdot \text{numBq}$ .
<i>tapsFactor</i>	Scale factor for the taps of integer data type.
<i>numBq</i>	Number of cascades of biquads.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 \cdot \text{numBq}$ .
<i>ppState</i>	Pointer to the pointer to the biquad IIR state structure.
<i>pBuffer</i>	Pointer to the external buffer.

## Description

The function `ippsIIRInit_BiQuad` are declared in the `ipps.h` file. This function initializes a biquad (BQ) IIR filter state in the external buffer. The size of this buffer must be computed previously by calling the corresponding function `ippsIIRGetStateSize_BiQuad`. The initialization function copies the taps from the array `pTaps` into the state structure `ppState`. The `tapsFactor` is used to scale integer tap values. The array `pDlyLine` specifies the delay line values. The number of elements in the array `pDlyLine` is  $4 * numBq$  for the function flavor `ippsIIRInit_BiQuad_DF1`, and  $2 * numBq$  for all other flavors.

If the pointer to the array `pDlyLine` is not `NULL`, the array content is copied into the context structure, otherwise the delay values of the state structure are set to 0.

The function flavor `ippsIIRInit_BiQuad_DF1` operates with the delay line values that are arranged in the array as follows:

$X_{0,-2}, X_{0,-1}, B_{0,2}, Y_{0,-2}, Y_{0,-1}, X_{1,-2}, X_{1,-1}, Y_{1,-2}, Y_{1,-1}, \dots, X_{numBq-1,-2}, X_{numBq-1,-1}, Y_{numBq-1,-2}, Y_{numBq-1,-1}$ .

A biquad IIR filter is defined by a cascade of biquads. The number of cascades of biquads is specified by the `numBq` value. The  $6 * numBq$ -length array `pTaps` specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \dots A_{numBq-1,2}$

$A_{n,0} \neq 0, B_{n,0} \neq 0$

If the state is not created, the initialization function returns an error status.

The initialization functions with the `32s_32f` suffixes called with floating-point taps automatically convert the taps into integer data type.

In all cases the data is converted into integer type with scaling for better precision. [Example 6-15](#) shows how to convert floating-point taps into integer data type.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsDivByZeroErr</code>	Indicates an error when $A_0$ , $A_{n,0}$ or $B_{n,0}$ is equal to 0.
<code>ippStsIIROrderErr</code>	Indicates an error when <code>numBq</code> is less than or equal to 0.

## IIRGetStateSize

*Computes the length of the external buffer for the arbitrary IIR filter state structure.*

---

### Syntax

```
IppStatus ippsIIRGetStateSize32s_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32s_16s32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_16s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32s(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32sc_16sc32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize32fc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_16sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32sc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64f_32f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64f(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize64fc_32fc(int order, int* pBufferSize);
IppStatus ippsIIRGetStateSize_64fc(int order, int* pBufferSize);
```

### Parameters

<i>order</i>	Order of the IIR filter.
<i>pBufferSize</i>	Pointer to the computed buffer size value.

### Description

The function `ippsIIRGetStateSize` is declared in the `ipps.h` file. This function computes the size of the external buffer for an arbitrary IIR filter state, and stores the result in *pBufferSize*.

To compute a size of the buffer, the filter order parameter *order* must be specified.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pBufferSize</i> pointer is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>order</i> is less than or equal to 0.

## IIRGetStateSize\_BiQuad

*Computes the length of the external buffer for the biquad IIR filter state structure.*

---

### Syntax

```

IppStatus ippSIIRGetStateSize32s_BiQuad_16s(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize32s_BiQuad_16s32f(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize32f_BiQuad_16s(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize64f_BiQuad_16s(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize64f_BiQuad_32s(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize64f_BiQuad_DF1_32s(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize32sc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize32sc_BiQuad_16sc32fc(int numBq, int*
pBufferSize);
IppStatus ippSIIRGetStateSize32fc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize64fc_BiQuad_16sc(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize64fc_BiQuad_32sc(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize_BiQuad_DF1_32f(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize64f_BiQuad_32f(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize_BiQuad_64f(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize_BiQuad_32fc(int numBq, int* pBufferSize);
IppStatus ippSIIRGetStateSize64fc_BiQuad_32fc(int numBq, int* pBufferSize);

```

```
IppStatus ippsIIRGetStateSize_BiQuad_64fc(int order, int* pBufferSize);
```

### Parameters

*numBq*                      Number of cascades of biquads.  
*pBufferSize*              Pointer to the computed buffer size value.

### Description

The function `ippsIIRGetStateSize_BiQuad` is declared in the `ipps.h` file. This function computes the size of the external buffer for a corresponding biquad IIR filter state, and stores the result in *pBufferSize*.

To compute a size of the buffer, the number of cascades of biquads *numBq* must be specified.

### Return Values

`ippStsNoErr`              Indicates no error.  
`ippStsNullPtrErr`       Indicates an error when *pBufferSize* pointer is NULL.  
`ippStsIIROrderErr`      Indicates an error when *numBq* is less than or equal to 0.

## IIRSetTaps

*Sets the taps in an IIR filter state.*

---

### Syntax

```
IppStatus ippsIIRSetTaps_32f(const Ipp32f* pTaps, IppsIIRState_32f* pState);  

IppStatus ippsIIRSetTaps_32fc(const Ipp32fc* pTaps, IppsIIRState_32fc*  

pState);  

IppStatus ippsIIRSetTaps_64f(const Ipp64f* pTaps, IppsIIRState_64f* pState);  

IppStatus ippsIIRSetTaps_64fc(const Ipp64fc* pTaps, IppsIIRState_64fc*  

pState);  

IppStatus ippsIIRSetTaps32s_16s(const Ipp32s* pTaps, IppsIIRState32s_16s*  

pState, int tapsFactor);  

IppStatus ippsIIRSetTaps32s_16s32f(const Ipp32f* pTaps, IppsIIRState32s_16s*  

pState);
```

```

IppStatus ippsIIRSetTaps32sc_16sc(const Ipp32sc* pTaps, IppsIIRState32sc_16sc*
pState, int tapsFactor);

IppStatus ippsIIRSetTaps32sc_16sc32fc(const Ipp32fc* pTaps,
IppsIIRState32sc_16sc* pState);

IppStatus ippsIIRSetTaps32f_16s(const Ipp32f* pTaps, IppsIIRState32f_16s*
pState);

IppStatus ippsIIRSetTaps32fc_16sc(const Ipp32fc* pTaps, IppsIIRState32fc_16sc*
pState);

IppStatus ippsIIRSetTaps64f_16s(const Ipp64f* pTaps, IppsIIRState64f_16s*
pState);

IppStatus ippsIIRSetTaps64f_32s(const Ipp64f* pTaps, IppsIIRState64f_32s*
pState);

IppStatus ippsIIRSetTaps64f_32f(const Ipp64f* pTaps, IppsIIRState64f_32f*
pState);

IppStatus ippsIIRSetTaps64fc_16sc(const Ipp64fc* pTaps, IppsIIRState64fc_16sc*
pState);

IppStatus ippsIIRSetTaps64fc_32sc(const Ipp64fc* pTaps, IppsIIRState64fc_32sc*
pState);

IppStatus ippsIIRSetTaps64fc_32fc(const Ipp64fc* pTaps, IppsIIRState64fc_32fc*
pState);

```

## Parameters

<i>pTaps</i>	Pointer to the array containing the tap values.
<i>pState</i>	Pointer to the IIR filter state structure.
<i>tapsFactor</i>	Scale factor for the taps of <code>Ipp32s</code> data type (for integer versions only).

## Description

The function `ippsIIRSetTaps` is declared in the `ipps.h` file. This function sets new tap values in the previously initialized IIR filter state structure *pState*. New tap values must be specified in the array *pTaps*. To scale integer taps use the *tapsFactor* value.

The filter state must be initialized before calling the function `ippsIIRSetTaps`. The length of the array *pTaps* must be equal to the *tapsLen* parameter of the initialized filter state.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsDivByZeroErr</code>	Indicates an error when $A_0$ is equal to 0 for an arbitrary IIR filter state, or $A_0$ , $A_{n,0}$ or $B_{n,0}$ is equal to 0 for a biquad IIR filter state.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## IIRGetDlyLine

*Retrieves the delay line contents from the IIR filter state.*

---

### Syntax

```

IppStatus ippSIIRGetDlyLine32s_16s(IppsIIRState32s_16s* pState, const Ipp32s*
pDlyLine);

IppStatus ippSIIRGetDlyLine32f_16s(IppsIIRState32f_16s* pState, const Ipp32f*
pDlyLine);

IppStatus ippSIIRGetDlyLine64f_16s(IppsIIRState64f_16s* pState, const Ipp64f*
pDlyLine);

IppStatus ippSIIRGetDlyLine64f_32s(IppsIIRState64f_32s* pState, const Ipp64f*
pDlyLine);

IppStatus ippSIIRGetDlyLine64f_DF1_32s(IppsIIRState64f_32s* pState, const
Ipp32s* pDlyLine);

IppStatus ippSIIRGetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState, const
Ipp32sc* pDlyLine);

IppStatus ippSIIRGetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState, const
Ipp32fc* pDlyLine);

IppStatus ippSIIRGetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState, const
Ipp64fc* pDlyLine);

IppStatus ippSIIRGetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState, const
Ipp64fc* pDlyLine);

IppStatus ippSIIRGetDlyLine_32f(IppsIIRState_32f* pState, const Ipp32f*
pDlyLine);

```

```

IppStatus ippsIIRGetDlyLine64f_32f(IppsIIRState64f_32f* pState, const Ipp64f*
pDlyLine);

IppStatus ippsIIRGetDlyLine_64f(IppsIIRState_64f* pState, const Ipp64f*
pDlyLine);

IppStatus ippsIIRGetDlyLine_32fc(IppsIIRState_32fc* pState, const Ipp32fc*
pDlyLine);

IppStatus ippsIIRGetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState, const
Ipp64fc* pDlyLine);

IppStatus ippsIIRGetDlyLine_64fc(IppsIIRState_64fc* pState, const Ipp64fc*
pDlyLine);

```

## Parameters

<i>pState</i>	Pointer to the IIR filter state structure.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> for arbitrary filters and $2 * numBq$ for BQ filters.

## Description

The function `ippsIIRGetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from the corresponding state structure *pState* and stores them into the arrays *pDlyLine*. If the pointer is `NULL`, then the delay line values in the state structure are initialized to zero.

The corresponding filter state must be previously initialized by the one of the initialization functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## IIRSetDlyLine

Sets the delay line contents in an IIR filter state.

### Syntax

```

IppStatus ippsIIRSetDlyLine32s_16s(IppsIIRState32s_16s* pState, const Ipp32s*
pDlyLine);

IppStatus ippsIIRSetDlyLine32f_16s(IppsIIRState32f_16s* pState, const Ipp32f*
pDlyLine);

IppStatus ippsIIRSetDlyLine64f_16s(IppsIIRState64f_16s* pState, const Ipp64f*
pDlyLine);

IppStatus ippsIIRSetDlyLine64f_32s(IppsIIRState64f_32s* pState, const Ipp64f*
pDlyLine);

IppStatus ippsIIRSetDlyLine64f_DF1_32s(IppsIIRState64f_32s* pState, const
Ipp32s* pDlyLine);

IppStatus ippsIIRSetDlyLine32sc_16sc(IppsIIRState32sc_16sc* pState, const
Ipp32sc* pDlyLine);

IppStatus ippsIIRSetDlyLine32fc_16sc(IppsIIRState32fc_16sc* pState, const
Ipp32fc* pDlyLine);

IppStatus ippsIIRSetDlyLine64fc_16sc(IppsIIRState64fc_16sc* pState, const
Ipp64fc* pDlyLine);

IppStatus ippsIIRSetDlyLine64fc_32sc(IppsIIRState64fc_32sc* pState, const
Ipp64fc* pDlyLine);

IppStatus ippsIIRSetDlyLine_32f(IppsIIRState_32f* pState, const Ipp32f*
pDlyLine);

IppStatus ippsIIRSetDlyLine64f_32f(IppsIIRState64f_32f* pState, const Ipp64f*
pDlyLine);

IppStatus ippsIIRSetDlyLine_64f(IppsIIRState_64f* pState, const Ipp64f*
pDlyLine);

IppStatus ippsIIRSetDlyLine_32fc(IppsIIRState_32fc* pState, const Ipp32fc*
pDlyLine);

IppStatus ippsIIRSetDlyLine64fc_32fc(IppsIIRState64fc_32fc* pState, const
Ipp64fc* pDlyLine);

```

```
IppStatus ippsIIRSetDlyLine_64fc(IppsIIRState_64fc* pState, const Ipp64fc*
pDlyLine);
```

## Parameters

<i>pState</i>	Pointer to the IIR filter state structure.
<i>pDlyLine</i>	Pointer to the array holding the delay line values. The number of elements in the array is <i>order</i> for arbitrary filters and $2 \cdot \text{numBq}$ for BQ filters. If the pointer is <code>NULL</code> , then the delay line values in the state structure are initialized to zero.

## Description

The function `ippsIIRSetDlyLine` is declared in the `ipps.h` file. This function copies the delay line values from *pDlyLine* and stores them into the state structure *pState*. If the pointer is `NULL`, then the delay line values in the state structure are initialized to zero.

The filter state must be previously initialized by the one of the initialization functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## IIROne

*Filters a single sample through an IIR filter.*

---

### Syntax

```
IppStatus ippsIIROne_32f(Ipp32f src, Ipp32f* pDstVal, IppsIIRState_32f*
pState);
```

```
IppStatus ippsIIROne_64f(Ipp64f src, Ipp64f* pDstVal, IppsIIRState_64f*
pState);
```

```
IppStatus ippsIIROne64f_32f(Ipp32f src, Ipp32f* pDstVal, IppsIIRState64f_32f*
pState);
```

```
IppStatus ippsIIROne_32fc(Ipp32fc src, Ipp32fc* pDstVal, IppsIIRState_32fc*
pState);
```

```
IppStatus ippsIIROne_64fc(Ipp64fc src, Ipp64fc* pDstVal, IppsIIRState_64fc*
pState);

IppStatus ippsIIROne64fc_32fc(Ipp32fc src, Ipp32fc* pDstVal,
IppsIIRState64fc_32fc* pState);

IppStatus ippsIIROne32s_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
IppsIIRState32s_16s* pState, int scaleFactor);

IppStatus ippsIIROne32f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
IppsIIRState32f_16s* pState, int scaleFactor);

IppStatus ippsIIROne64f_16s_Sfs(Ipp16s src, Ipp16s* pDstVal,
IppsIIRState64f_16s* pState, int scaleFactor);

IppStatus ippsIIROne64f_32s_Sfs(Ipp32s src, Ipp32s* pDstVal,
IppsIIRState64f_32s* pState, int scaleFactor);

IppStatus ippsIIROne32sc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
IppsIIRState32sc_16sc* pState, int scaleFactor);

IppStatus ippsIIROne32fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
IppsIIRState32fc_16sc* pState, int scaleFactor);

IppStatus ippsIIROne64fc_16sc_Sfs(Ipp16sc src, Ipp16sc* pDstVal,
IppsIIRState64fc_16sc* pState, int scaleFactor);

IppStatus ippsIIROne64fc_32sc_Sfs(Ipp32sc src, Ipp32sc* pDstVal,
IppsIIRState64fc_32sc* pState, int scaleFactor);
```

## Parameters

<i>pState</i>	Pointer to the IIR filter state structure.
<i>src</i>	Input sample.
<i>pDstVal</i>	Pointer to the output filtered sample.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsIIROne` is declared in the `ipps.h` file. This function filters a single sample *src* through an IIR filter with real taps, and stores the result in *pDstVal*. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

The filter state must be initialized before calling the function `ippsFIROne` function. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the state identifier is incorrect.

## IIR

*Filters a source vector through an IIR filter.*

---

### Syntax

#### Case 1: Not-in-place operation on integer samples

```
IppStatus ippsIIR32s_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
IppsIIRState32s_16s* pState, int scaleFactor);
```

```
IppStatus ippsIIR32f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
IppsIIRState32f_16s* pState, int scaleFactor);
```

```
IppStatus ippsIIR64f_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
IppsIIRState64f_16s* pState, int scaleFactor);
```

```
IppStatus ippsIIR64f_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, int len,
IppsIIRState64f_32s* pState, int scaleFactor);
```

```
IppStatus ippsIIR32sc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
IppsIIRState32sc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsIIR32fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
IppsIIRState32fc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsIIR64fc_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, int len,
IppsIIRState64fc_16sc* pState, int scaleFactor);
```

```
IppStatus ippsIIR64fc_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, int len,
IppsIIRState64fc_32sc* pState, int scaleFactor);
```

**Case 2: Not-in-place operation on floating point samples**

```

IppStatus ippsIIR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
IppsIIRState_32f* pState);

IppStatus ippsIIR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
IppsIIRState_64f* pState);

IppStatus ippsIIR64f_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
IppsIIRState64f_32f* pState);

IppStatus ippsIIR_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
IppsIIRState_32fc* pState);

IppStatus ippsIIR_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
IppsIIRState_64fc* pState);

IppStatus ippsIIR64fc_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
IppsIIRState64fc_32fc* pState);

```

**Case 3: In-place operation on integer samples**

```

IppStatus ippsIIR32s_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState32s_16s*
pState, int scaleFactor);

IppStatus ippsIIR32f_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState32f_16s*
pState, int scaleFactor);

IppStatus ippsIIR64f_16s_ISfs(Ipp16s* pSrcDst, int len, IppsIIRState64f_16s*
pState, int scaleFactor);

IppStatus ippsIIR64f_32s_ISfs(Ipp32s* pSrcDst, int len, IppsIIRState64f_32s*
pState, int scaleFactor);

IppStatus ippsIIR32fc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
IppsIIRState32fc_16sc* pState, int scaleFactor);

IppStatus ippsIIR32sc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
IppsIIRState32sc_16sc* pState, int scaleFactor);

IppStatus ippsIIR64fc_16sc_ISfs(Ipp16sc* pSrcDst, int len,
IppsIIRState64fc_16sc* pState, int scaleFactor);

IppStatus ippsIIR64fc_32sc_ISfs(Ipp32sc* pSrcDst, int len,
IppsIIRState64fc_32sc* pState, int scaleFactor);

```

**Case 4: In-place operation on floating point samples**

```

IppStatus ippsIIR_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState_32f* pState);

```

```

IppStatus ippsIIR_64f_I(Ipp64f* pSrcDst, int len, IppsIIRState_64f* pState);
IppStatus ippsIIR64f_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState64f_32f*
pState);
IppStatus ippsIIR_32fc_I(Ipp32fc* pSrcDst, int len, IppsIIRState_32fc*
pState);
IppStatus ippsIIR_64fc_I(Ipp64fc* pSrcDst, int len, IppsIIRState_64fc*
pState);
IppStatus ippsIIR64fc_32fc_I(Ipp32fc* pSrcDst, int len, IppsIIRState64fc_32fc*
pState);

```

## Case 4: Operation with specified number of vector.

```

IppStatus ippsIIR_32f_P(const Ipp32f** ppSrc, Ipp32f** ppDst, int len, int
nChannels, IppsIIRState_32f** ppState);
IppStatus ippsIIR_32f_IP(Ipp32f** ppSrcDst, int len, int nChannels,
IppsIIRState_32f** ppState);

```

## Parameters

<i>pState</i>	Pointer to the IIR filter state structure.
<i>ppState</i>	Pointer to the array of the pointers to the IIR filter state structures.
<i>pSrc</i>	Pointer to the source vector.
<i>ppSrc</i>	Pointer to the array of pointers to the source vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operations.
<i>ppSrcDst</i>	Pointer to the array of pointers to the source and destination vectors for the in-place operations.
<i>len</i>	Number of elements of the vector to be filtered.
<i>nChannels</i>	Number of vectors to be filtered.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .



## Description

The function `ippsIIR` is declared in the `ipps.h` file. This function filters *len* elements of the source vector *pSrc* or *pSrcDst* through an IIR filter, and stores the results in *pDst* or *pSrcDst*, respectively. The filter parameters are specified in *pState*. The output of the integer sample is scaled according to *scaleFactor* and can be saturated.

Do not modify the *scaleFactor* value unless the state structure is changed.

The filter state must be initialized before calling the function `ippsIIR`. Specify the number of taps *tapsLen*, the tap values in *pTaps*, the delay line values in *pDlyLine*, and the *order* or *numBq* value beforehand.

Function flavors described in the **Case 4** filter simultaneously the *nChannels* source vectors. Each vector must have the *len* elements and is filtered with its own state structure. These state structures must be initialized beforehand.

Examples 6-14 - 6-16 below show how the Intel IPP functions for IIR filtering can be used.

Example 6-14 illustrates using `ippsIIR_32f` function.

Example 6-15 demonstrates how to use the function `ippsIIR` to filter a sample. The function `ippsConvert_64f32s_Sfs` converts floating-point taps into integer data type before calling `ippsIIRInitAlloc_32s`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.
<code>ippStsChannelErr</code>	Indicates an error when <i>nChannels</i> is less or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## Example 6-14 Using the ippsIIR\_32f Function to Suppress a 60 Hz Signal

```

IppStatus iir( void )
{
    #undef NUMITERS
    #define NUMITERS 150

    int n;

    IppStatus status;

    IppsIIRState_32f *ctx;

    Ipp32f *x = ippsMalloc_32f( NUMITERS ), *y = ippsMalloc_32f( NUMITERS );

    /// A second-order notch filter having notch freq at 60 Hz
    const float taps[] = {
        0.940809f,-1.105987f,0.940809f,1,-1.105987f,0.881618f
    };

    /// generate a signal having 60 Hz freq sampled with 400 Hz freq
    for(n=0;n<NUMITERS;++n)x[n]=(float)sin(IPP_2PI *n *60 /400);

    ippsIIRInitAlloc_32f( &ctx, taps, 2, NULL );

    status = ippsIIR_32f( x, y, NUMITERS, ctx );

    printf_32f( " IIR 32f output+120 =", y+120, 5, status );

    ippsIIRFree_32f( ctx );

    ippsFree( y );

    ippsFree( x );

    return status;
}

```

Output:

```
IIR 32f output + 120 = -0.000094 0.000339 0.000458 0.000208 -0.000173
```

Matlab\* Analog:

```

>> B = [0.940809,-1.105987,0.940809]; A = [1,-1.105987,0.881618];
n = 0:150; x = sin(2*pi*n*60/400); y = filter(B,A,x); y(121:125)

```

**Example 6-15 Using the ippsIIR Function to Filter a Sample**

```

IppStatus iirl16s( void ) {
#undef NUMITERS
#define NUMITERS 150

    int n, tapsfactor = 30;

    IppStatus status;

    IppsIIRState32s_16s *ctx;

    Ipp16s *x = ippsMalloc_16s( NUMITERS ), *y = ippsMalloc_16s( NUMITERS );

    /// A second-order notch filter having notch freq at 60 Hz

    Ipp64f taps[6] = {
        0.940809f, -1.105987f, 0.940809f, 1, -1.105987f, 0.881618f
    };

    Ipp32s taps32s[6];
    Ipp64f tmax, tmp[6];
    ippsAbs_64f( taps, tmp, 6 );
    ippsMax_64f( tmp, 6, &tmax );
    tapsfactor = 0;
    if( tmax > IPP_MAX_32S )
        while( (tmax/=2) > IPP_MAX_32S ) ++tapsfactor;
    else
        while( (tmax*=2) < IPP_MAX_32S ) --tapsfactor;
    if( tapsfactor > 0 )
        ippsDivC_64f_I( (float)(1<<(++tapsfactor)), taps, 6 );
    else if( tapsfactor < 0 )
        ippsMulC_64f_I( (float)(1<<(-(tapsfactor))), taps, 6 );
    ippsConvert_64f32s_Sfs( taps, taps32s, 6, ippRndNear, 0 );

    /// generate a signal of 60 Hz freq that is sampled with 400 Hz freq
    for(n=0; n<NUMITERS; ++n) x[n] = (Ipp16s)(1000*sin(IPP_2PI*n*60/400));
}

```

```

    ippsIIRInitAlloc32s_16s( &ctx, taps32s, 2, tapsfactor, NULL );
    status = ippsIIR32s_16s_Sfs( x, y, NUMITERS, ctx, 0 );
    printf_16s( " IIR 32s output+120 =", y+120, 5, status );
    ippsIIRFree32s_16s( ctx );
    ippsFree( y );
    ippsFree( x );
    return status;
}

```

Output:

IIR 32s output + 120 = 0 0 0 0 0

## Example 6-16 Using the function ippsIIR for Low Pass Filtering

```

#include <ipp.h>

/*
y[n] = ( 1 * x[n- 2])
+ ( 2 * x[n- 1])
+ ( 1 * x[n- 0])
+ ( -0.1958157127 * y[n- 2])
+ ( -0.3695273774 * y[n- 1])
*/

#define NZEROS 2
#define NPOLES 2
#define GAIN 2.555350342e+00f

static float xv[NZEROS+1], yv[NPOLES+1];

static void filterloop( const float* x, float* y, int len )
{ for (int i=0; i<len; i++ )
  { xv[0] = xv[1]; xv[1] = xv[2];

```

---

```
xv[2] = x[i] / GAIN;
yv[0] = yv[1]; yv[1] = yv[2];

yv[2] = (xv[0] + xv[2]) + 2 * xv[1]
+ ( -0.1958157127f * yv[0]) + ( -0.3695273774f * yv[1]);
y[i] = yv[2];
}
}

const int LEN = 100;

int main() {
float x[LEN], yu[LEN], yi[LEN];
for(int i=0; i<LEN; i++) x[i] = i-3;
/// customer will compute
/// -1.1740073 -2.6968584 -1.9042342 -0.33358926
```

```

filterloop(x,yu,LEN);

/// matlab gives
///>> b=[1 2 1];a=[1 0.3695273774 0.1958157127 ];x=[-3:1:15]/2.555350342;filter(b,a,x)
/// -1.1740 -2.6969 -1.9042 -0.3336
/// IPP will compute
/// -1.1740074 -2.6968584 -1.9042342 -0.33358920

const int order = 2;

float taps[6] = {1.0f, 2.0f, 1.0f, 1.0f, 0.3695273774f, 0.1958157127f};

IppsIIRState_32f *ctx;

ippsIIRInitAlloc_32f( &ctx, taps, order, (Ipp32f*)0 );

ippsDivC_32f_I( GAIN, x, LEN);

ippsIIR_32f( x, yi, LEN, ctx );

ippsIIRFree_32f( ctx );

return 0;

}

```

## IROne\_Direct

*Directly filters a single sample through the arbitrary IIR filter.*

---

### Syntax

```

IppStatus ippsIIROne_Direct_16s(Ipp16s src, Ipp16s* pDstVal, const Ipp16s*
pTaps, int order, Ipp32s* pDlyLine);

IppStatus ippsIIROne_Direct_16s_I(Ipp16s* pSrcDstVal, const Ipp16s* pTaps,
int order, Ipp32s* pDlyLine);

```

### Parameters

<i>src</i>	Input sample.
<i>pDstVal</i>	Pointer to the output filtered sample.
<i>pSrcDstVal</i>	Pointer to the input and output sample for in-place operation.

<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $2*(order + 1)$ .
<i>order</i>	Order of the arbitrary IIR filter.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> .

## Description

The function `ippsIIROne_Direct` is declared in the `ipps.h` file. This function filters a single sample *src* (*pSrcDstVal* for in-place flavor) through an arbitrary order IIR filter and stores the result in *pDstVal* (*pSrcDstVal*). The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The *order*-length array *pDlyLine* specifies the delay line values. The  $2*(order + 1)$ -length array *pTaps* specifies the taps arranged in the array as follows:

$B_0, B_1, \dots, B_{order}, A_0, A_1, \dots, A_{order}$

The value  $A_0 \geq 0$  holds the scale factor (and not a divisor) for all the other taps.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsIIROrderErr</code>	Indicates an error when <i>order</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when $A_0$ is less than 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect.

## IIROne\_BiQuadDirect

*Directly filters a single sample through the biquad IIR filter.*

---

### Syntax

```

IppStatus ippsIIROne_BiQuadDirect_16s(Ipp16s src, Ipp16s* pDtsVal, const
Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);

IppStatus ippsIIROne_BiQuadDirect_16s_I(Ipp16s* pSrcDtsVal, const Ipp16s*
pTaps, int numBq, Ipp32s* pDlyLine);

```

## Parameters

<i>src</i>	Input sample.
<i>pDstVal</i>	Pointer to the output filtered sample.
<i>pSrcDstVal</i>	Pointer to the input and output sample for in-place operation.
<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $6 * numBq$ .
<i>numBq</i>	Number of cascades of biquads.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * numBq$ .

## Description

The function `ippsIIROne_BiQuadDirect` is declared in the `ipps.h` file. This function filters a single sample *src* (*pSrcDstVal* for in-place flavors) through a biquad IIR filter and stores the result in *pDstVal* (*pSrcDstVal*). The number of cascades of biquads is defined by the *numBq* value. The  $2 * numBq$ -length array *pDlyLine* specifies the delay line values. The  $6 * numBq$ -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \dots A_{numBq-1,2}$

Values  $A_{n,0} \geq 0$ ,  $B_{n,0} \geq 0$  hold the scale factors (and not divisors) for all the other taps of each section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsIIROrderErr</code>	Indicates an error when <i>numBq</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when $A_{n,0}$ is less than 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier is incorrect



## IIR\_Direct

*Directly filters a source vector through the arbitrary IIR filter.*

---

### Syntax

```
IppStatus ippsIIR_Direct_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, const
Ipp16s* pTaps, int order, Ipp32s* pDlyLine);

IppStatus ippsIIR_Direct_16s_I(Ipp16s* pSrcDst, int len, const Ipp16s* pTaps,
int order, Ipp32s* pDlyLine);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements to be filtered.
<i>pTaps</i>	Pointer to the array containing the taps. The number of elements in the array is $2 * (order + 1)$ .
<i>order</i>	Order of the arbitrary IIR filter.
<i>pDlyLine</i>	Pointer to the array containing the delay line values. The number of elements in the array is <i>order</i> .

### Description

The function `ippsIIR_Direct` is declared in the `ipps.h` file. This function filters *len* elements of the source vector *pSrc* or *pSrcDst* through an arbitrary IIR filter and stores the results in *pDst* or *pSrcDst*, respectively. The filter order is defined by the *order* value which is equal to 0 for zero-order filters. The *order*-length array *pDlyLine* specifies the delay line values. The  $2 * (order + 1)$ -length array *pTaps* specifies the taps arranged in the array as follows:

$$B_0, B_1, \dots, B_{order}, A_0, A_1, \dots, A_{order}$$

The value  $A_0 \geq 0$  holds the scale factor (and not a divisor) for all the other taps.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less or equal to 0.
<code>ippStsIIROrderErr</code>	Indicates an error when <code>order</code> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when $A_0$ is less than 0.

## IIR\_BiQuadDirect

*Directly filters a source vector through the biquad IIR filter.*

---

### Syntax

```
IppStatus ippSIIR_BiQuadDirect_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
len, const Ipp16s* pTaps, int numBq, Ipp32s* pDlyLine);

IppStatus ippSIIR_BiQuadDirect_16s_I(Ipp16s* pSrcDst, int len, const Ipp16s*
pTaps, int numBq, Ipp32s* pDlyLine);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>pSrcDst</code>	Pointer to the source and destination vector for the in-place operation.
<code>len</code>	Number of elements to be filtered.
<code>pTaps</code>	Pointer to the array containing the taps. The number of elements in the array is $6 * numBq$ for BQ filters.
<code>numBq</code>	Number of cascades of biquads.
<code>pDlyLine</code>	Pointer to the array containing the delay line values. The number of elements in the array is $2 * numBq$ for BQ filters.

## Description

The function `ippsIIR_BiQuadDirect` is declared in the `ipps.h` file. This function filters *len* elements in the source vector *pSrc* or *pSrcDst* through a biquad IIR filter and stores the result in *pDst* or *pSrcDst*, respectively. The number of cascades of biquads is defined by the *numBq* value. The  $2 * \text{numBq}$ -length array *pDlyLine* specifies the delay line values. The  $6 * \text{numBq}$ -length array *pTaps* specifies the taps arranged in the array as follows:

$B_{0,0}, B_{0,1}, B_{0,2}, A_{0,0}, A_{0,1}, A_{0,2}; B_{1,0}, B_{1,1}, B_{1,2}, A_{1,0}, A_{1,1}, A_{1,2}; \dots A_{\text{numBq}-1,2}$

Values  $A_{n,0} \geq 0, B_{n,0} \geq 0$  hold the scale factors (and not divisors) for all the other taps of each section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsIIROrderErr</code>	Indicates an error when <i>numBq</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when $A_{n,0}$ is less than 0.

## IIRSparseInit

*Initializes a sparse IIR filter structure.*

### Syntax

```
IppStatus ippsIIRSparseInit_32f(IppsIIRSparseState_32f** ppState, const
Ipp32f* pNZTaps, const Ipp32s* pNZTapPos, int nzTapsLen1, int nzTapsLen2,
const Ipp32f* pDlyLine, Ipp8u* pBuffer);
```

### Parameters

<i>pNZTaps</i>	Pointer to the array containing the non-zero tap values.
<i>pNZTapPos</i>	Pointer to the array containing positions of the non-zero tap values. The number of elements in the array is <i>nzTapsLen</i> .

<i>nzTapsLen1</i> , <i>nzTapsLen2</i>	Pointer to the destination vector.
<i>pDlyLine</i>	Pointer to the array containing the delay line values.
<i>ppState</i>	Double pointer to the sparse IIR state structure.
<i>pBuffer</i>	Pointer to the external buffer for the sparse IIR state structure.

## Description

The function `ippsIIRSparseInit` is declared in the `ipps.h` file. This function initializes a sparse IIR filter state structure *ppState* in the external buffer *pBuffer*. The size of this buffer must be computed previously by calling the function `ippsIIRSparseGetStateSize`.

The  $(nzTapsLen1 + nzTapsLen2)$ -length array *pNZTaps* specifies the non-zero taps arranged in the array as follows:

$B_0, B_1, \dots, B_{nzTapsLen1-1}, A_0, A_1, \dots, A_{nzTapsLen2-1}$ .

The  $(nzTapsLen1 + nzTapsLen2)$ -length array *pNZTapPos* specifies the non-zero tap positions arranged in the array as follows:

$BP_0, BP_1, \dots, BP_{nzTapsLen1-1}, AP_0, AP_1, \dots, AP_{nzTapsLen2-1}, AP_0 \neq 0$

The initialization function copies the values of filter coefficients from the array *pNZTaps* containing non-zero taps and their positions from the array *pNZTapPos* into the state structure *ppState*. The array *pDlyLine* contains the delay line values. The number of elements in this array is  $pNZTapPos[nzTapsLen1-1] + pNZTapPos[nzTapsLen1 + nzTapsLen2-1]$ . If the pointer to the array *pDlyLine* is not NULL, the array contents is copied into the state structure *ppState*, otherwise the delay line values in the state structure are initialized to 0.



**CAUTION.** The values of *nzTapsLen1*, *nzTapsLen2*, *pNZTapPos[nzTapsLen1-1]*, and *pNZTapPos[nzTapsLen1 + nzTapsLen2-1]* must be equal to those specified in the function `ippsIIRSparseGetStateSize`.

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.
<code>ippStsIIROrderErr</code>	Indicates an error if <i>nzTapsLen1</i> is less than or equal to 0, or <i>nzTapsLen2</i> is less than 0.

`ippStsSparseErr` Indicates an error if positions of the non-zero taps are not in ascending order, or are negative or repetitive; or `pNZTapPos[nzTapsLen1]` is equal to 0.

## IIRSparseGetStateSize

*Computes the size of the external buffer for the sparse IIR filter structure.*

---

### Syntax

```
IppStatus ippsIIRSparseGetStateSize_32f(int nzTapsLen1, int nzTapsLen2, int
order1, int order2, int* pStateSize);
```

### Parameters

<code>nzTapsLen1,</code> <code>nzTapsLen2</code>	Number of elements in the array containing the non-zero tap values.
<code>order1, order2</code>	Order of the sparse IIR filter.
<code>pStateSize</code>	Pointer to the computed value of the external buffer.

### Description

The function `ippsIIRSparseGetStateSize` is declared in the `ipps.h` file. This function computes the size in bytes of the external buffer for a sparse IIR filter state that is required for the function `ippsIIRSparseInit`. The computations are based on the specified number of non-zero filter coefficients `nzTapsLen1`, `nzTapsLen2` and filter orders `order1`, `order2`. `order1 = pNZTapPos[nzTapsLen1 - 1]`, `order2 = pNZTapPos[nzTapsLen1 + nzTapsLen2 - 1]` (see description of the function `ippsIIRSparseInit` for more details). The result value is stored in the `pStateSize`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <code>pStateSize</code> pointer is <code>NULL</code> .
<code>ippStsIIROrderErr</code>	Indicates an error if <code>nzTapsLen1</code> is less than or equal to 0, or <code>nzTapsLen2</code> is less than 0.
<code>ippStsSparseErr</code>	Indicates an error if <code>order1</code> or <code>order2</code> is less than 0.

## IIRSparse

Filters a source vector through a sparse IIR filter.

### Syntax

```
IppStatus ippsIIRSparse_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
IppIIRSparseState_32f* pState);
```

### Parameters

<i>pState</i>	Pointer to the sparse IIR filter state structure.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements that will be filtered.

### Description

The function `ippsIIRSparse` is declared in the `ipps.h` file. This function applies the sparse IIR filter to the *len* elements of the source vector *pSrc*, and stores the results in *pDst*. The filter parameters - the number of non-zero taps *nzTapsLen1*, *nzTapsLen2*, their values *pNZTaps* and their positions *pNZTapPos*, and the delay line values *pDlyLine* - are specified in the sparse IIR filter structure *pState* that should be previously initialized the function `ippsIIRSparseInit`.

In the following definition of the sparse IIR filter, the sample to be filtered is denoted  $x(n)$ , the non-zero taps are denoted  $B_i$  and  $A_i$ , their positions are denoted  $BP_i$  and  $AP_i$ .

The non-zero taps are arranged in the array as follows:

$B_0, B_1, \dots, B_{nzTapsLen1-1}, A_0, A_1, \dots, A_{nzTapsLen2-1}$ .

The non-zero tap positions are arranged in the array as follows:

$BP_0, BP_1, \dots, BP_{nzTapsLen1-1}, AP_0, AP_1, \dots, AP_{nzTapsLen2-1}, AP_0 \neq 0$

The return value is  $y(n)$  is defined by the formula for a sparse IIR filter:

$$y(n) = \sum_{k=0}^{nzTapsLen1-1} B_k \cdot x(n-BP_k) + \sum_{k=1}^{nzTapsLen2-1} A_k \cdot y(n-AP_k) \quad 0 \leq n < len$$

After the function has performed calculations, it updates the delay line values stored in the filter state structure.

Example 6-17 below shows how to use the sparse IIR filter functions.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>len</code> is less than or equal to 0.

## Example 6-17 Using the Sparse IIR Filter Functions

```
int buflen;

Ipp8u *buf;

int nzTapsLen1 = 5; //number of non-zero taps in the FIR part of the formula
int nzTapsLen2 = 3; //number of non-zero taps in the IIR part of the formula
Ipp32f nzTaps [] = {0.5, 0.4, 0.3, 0.2, 0.1, 0.8, 0.7, 0.6};
                                     //non-zero taps values (FIR+IIR)
Ipp32s nzTapsPos[] = {0, 10, 20, 30, 40, 1, 5, 15};

                                     //non-zero tap positions (FIR+IIR)
IppsIIRSparseState_32f* iirState;
Ipp32f *src, *dst;

/* ..... */

ippsIIRSparseGetStateSize_32f(nzTapsLen1, nzTapsLen2, nzTapsPos [nzTapsLen1 - 1],
                             nzTapsPos [nzTapsLen1 + nzTapsLen2 - 1], &buflen);

buf = ippsMalloc_8u(buflen);
ippsIIRSparseInit_32f(&iirState, nzTaps, nzTapsPos, nzTapsLen1, nzTapsLen2,
                     NULL, buf);

/* . . . . initializing src somehow . . . . */
ippsIIRSparse_32f(src, dst, len, iirState);

/* dst[i] = src[i] * 0.5 + src[i-10] * 0.4 + src[i-20] * 0.3 + src[i-30] * 0.2
   + src[i-40] * 0.1 + dst[i-1] * 0.8 + dst[i-5] * 0.7 + dst[i-15] * 0.6 */

/* ..... */

ippsFree(buf);
```

## IIRGenLowpass, IIRGenHighpass

*Computes lowpass and highpass IIR filter coefficients.*

---

### Syntax

```
IppStatus ippsIIRGenLowpass_64f(Ipp64f rFreq, Ipp64f ripple, int order,
Ipp64f* pTaps, IppsIIRFilterType filterType);
```



```
IppStatus ippsIIRGenHighpass_64f(Ipp64f rFreq, Ipp64f ripple, int order,
Ipp64f* pTaps, IppsIIRFilterType filterType);
```

## Parameters

<i>rFreq</i>	Cutoff frequency, should be in the range (0, 0.5).
<i>ripple</i>	Possible ripple in pass band for <code>ippChebyshev1</code> type of filter.
<i>order</i>	Order of the filter [1, 12]
<i>pTaps</i>	Pointer to the array where computed tap values are stored.
<i>filterType</i>	Type of the IIR filter, possible values: <code>ippButterworth</code> , <code>ippChebyshev1</code> .

## Description

The functions `ippsIIRGenLowpass` and `ippsIIRGenHighpass` are declared in the `ipps.h` file. These functions compute coefficients for lowpass or highpass IIR filters, respectively, with the cutoff frequency *rFreq*. The parameter *filterType* specifies the type of the filter. The computed coefficients are stored in the array *pTaps*. Its length must be at least  $2 \times (\text{order} + 1)$  and the taps arranged in the array as follows:

$B_0, B_1, \dots, B_{\text{order}}, A_0, A_1, \dots, A_{\text{order}}$

## Application Notes

*Butterworth filters* are characterized by a magnitude response that is at most flat in the passband and monotonic overall. Butterworth filters sacrifice rolloff steepness for monotonicity in the passband. Unless the smoothness of the Butterworth filter is needed, *Chebyshev1 filter* can generally provide steeper rolloff characteristics with a lower filter order. Chebyshev1 type filters are equiripple in the passband and monotonic in the stopband. For `ippButterworth` filter cutoff frequency is the frequency where the magnitude response of the filter is  $2^{-1/2}$ . For `ippChebyshev1` filter cutoff frequency is the frequency at which the magnitude response of the filter is  $(-ripple)$  dB. For the functions `ippsIIRGenLowpass` and `ippsIIRGenHighpass`, the normalized cutoff frequency *rFreq* must be a number between 0 and 0.5, where 0.5 corresponds to the Nyquist frequency,  $\pi$  radians per sample. The correspondence between MATLAB's *Wn* and Intel IPP *rFreq* is very simple:  $Wn = 2 \times rFreq$ .

Examples:

1) For data sampled at 1000 Hz, create a 9th-order highpass Butterworth filter with cutoff frequency at 300 Hz.

Intel IPP:

```
Ipp64f pTaps[2*(9+1)];

status = ippsIIRGenHighpass( 300.0/1000.0, 0, 9, pTaps, ippButterworth );
```

MATLAB:

```
[b,a] = butter(9,300/500,'high');
```

2) For data sampled at 1000 Hz, create a 9th-order lowpass Chebyshev1 filter with ripple in the passband of 0.5 dB and a cutoff frequency at 300 Hz.

Intel IPP:

```
Ipp64f pTaps[2*(9+1)];

status = ippsIIRGenLowpass( 300.0/1000.0, 0.5, 9, pTaps, ippChebyshev1 );
```

MATLAB:

```
[b,a] = cheby1(9, 0.5, 300/500);
```

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pTaps</i> pointer is NULL.
<code>ippStsIIRGenOrderErr</code>	Indicates an error when the <i>order</i> is less than 1 or greater than 12.
<code>ippStsFilterFrequencyErr</code>	Indicates an error when the <i>rFreq</i> is out of the range.

## Median Filter Functions

Median filters are nonlinear rank-order filters based on replacing each element of the source vector with the median value, taken over the fixed neighborhood (mask) of the processed element. These filters are extensively used in image and signal processing applications. Median filtering removes impulsive noise, while keeping the signal blurring to the minimum. Typically mask size (or window width) is set to odd value which ensures simple function implementation and low output signal bias. In the Intel IPP median function implementation, the mask is always centered at the input element for which the median value is computed. You can use an even mask size in function calls as well, but internally it will be changed to odd by subtracting 1. Another specific feature of the median function implementation in Intel IPP is that elements outside the source vector, which are needed to determine the median value for “border” elements, are set to be equal to the corresponding edge element of the source vector.

## FilterMedian

*Computes median values for each source vector element.*

---

### Syntax

```
IppStatus ippsFilterMedian_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int maskSize);

IppStatus ippsFilterMedian_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, int maskSize);

IppStatus ippsFilterMedian_32s(const Ipp32s* pSrc, Ipp32s* pDst, int len, int maskSize);

IppStatus ippsFilterMedian_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, int maskSize);

IppStatus ippsFilterMedian_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, int maskSize);

IppStatus ippsFilterMedian_8u_I(Ipp8u* pSrcDst, int len, int maskSize);

IppStatus ippsFilterMedian_16s_I(Ipp16s* pSrcDst, int len, int maskSize);

IppStatus ippsFilterMedian_32s_I(Ipp32s* pSrcDst, int len, int maskSize);

IppStatus ippsFilterMedian_32f_I(Ipp32f* pSrcDst, int len, int maskSize);

IppStatus ippsFilterMedian_64f_I(Ipp64f* pSrcDst, int len, int maskSize);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vector.
<i>maskSize</i>	Median mask size, must be a positive integer. If an even value is specified, the function subtracts 1 and uses the odd value of the filter mask for median filtering.

## Description

The function `ippsFilterMedian` is declared in the `ipps.h` file. This function computes median values for each element of the source vector `pSrc` or `pSrcDst`, and stores the result in `pDst` or `pSrcDst`, respectively.



**NOTE.** The value of a non-existent point is equal to the last point value, for example: `x[- 1]=x[0]`, or `x[len] = x [len - 1]`.

Example 6-18 below illustrates using `ippsFilterMedian_16s_I` for single-rate filtering.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsEvenMedianMaskSize</code>	Indicates a warning when the median mask length is even.

## Example 6-18 Single-Rate Filtering with the `ippsFilterMedian` Function

```
void median(void) {
    Ipp16s x[8] = {1,2,127,4,5,0,7,8};
    IppStatus status = ippsFilterMedian_16s_I(x, 8, 3);
    printf_16s("median =", x,8, status);
}
```

Output:

```
median =  1 2 4 5 4 5 7 8
```

Matlab\* Analog:

```
>> x = [1 2 127 4 5 0 7 8]; medfilt1(x)
```

# Transform Functions

This chapter describes the Intel<sup>®</sup> IPP functions that perform Fourier and discrete cosine transforms (DCT), as well as Hartley, Hilbert, Walsh-Hadamard and wavelet transforms of signals.

The full list of functions in this group is given in Table 7-1 below.

**Table 7-1 Intel IPP Transform Functions**

Function Base Name	Operation
Support Functions	
ConjPack	Converts the data in <code>Pack</code> format to complex data format.
ConjPerm	Converts the data in <code>Perm</code> format to complex data format.
ConjCcs	Converts the data in <code>CCS</code> format to complex data format.
MulPack	Multiplies the elements of two vectors stored in <code>Pack</code> format.
MulPerm	Multiplies the elements of two vectors stored in <code>Perm</code> format.
MulPackConj	Multiplies elements of a vector by the elements of a complex conjugate vector stored in <code>Pack</code> format.
Fast Fourier Transform Functions	
FFTInitAlloc_R, FFTInitAlloc_C	Allocates memory and initializes a fast Fourier transform specification structure for real and complex signals.
FFTFree_R, FFTFree_C	Closes a fast Fourier transform structure for real and complex signals.
FFTInit_R, FFTInit_C	Initializes a fast Fourier transform specification structure for real and complex signals.

Function Base Name	Operation
FFTGetSize_R, FFTGetSize_C	Computes sizes of the FFT specification structure and required working buffers.
FFTGetBufSize_R, FFTGetBufSize_C	Computes the size of the FFT work buffer.
FFTFwd_CToC	Computes the forward fast Fourier transform (FFT) of a complex signal.
FFTInv_CToC	Computes the inverse fast Fourier transform (FFT) of a complex signal.
FFTFwd_RToPack, FFTFwd_RToPerm, FFTFwd_RToCCS	Computes the forward fast Fourier transform (FFT) of a real signal.
FFTInv_PackToR, FFTInv_PermToR, FFTInv_CCSToR	Computes the inverse fast Fourier transform (FFT) of a real signal.
Discrete Fourier Transform Functions	
DFTInitAlloc_R, DFTInitAlloc_C	Initializes the discrete Fourier transform structure for real and complex signals.
DFTFree_R, DFTFree_C	Closes the discrete Fourier transform structure for real and complex signals.
DFTGetBufSize_R, DFTGetBufSize_C	Computes the size of the discrete Fourier transform work buffer.
DFTFwd_CToC	Computes the forward discrete Fourier transform of a complex signal.
DFTInv_CToC	Computes the inverse discrete Fourier transform of a complex signal.
DFTFwd_RToPack, DFTFwd_RToPerm, DFTFwd_RToCCS	Computes the forward discrete Fourier transform of a real signal.
DFTInv_PackToR, DFTInv_PermToR, DFTInv_CCSToR	Computes the inverse discrete Fourier transform of a real signal.
DFTOutOrdInitAlloc_C	Initializes the out-of-order discrete Fourier transform structure.

Function Base Name	Operation
DFTOutOrdFree_C	Closes the out-of-order discrete Fourier transform structure.
DFTOutOrdGetBufSize_C	Computes the size of the work buffer for the out-of-order discrete Fourier transform.
DFTOutOrdFwd_CToC	Computes the forward out-of-order discrete Fourier transform.
DFTOutOrdInv_CToC	Computes the inverse out-of-order discrete Fourier transform.
Goertz	Computes the discrete Fourier transform for a given frequency for a single complex signal.
GoertzTwo	Computes two discrete Fourier transforms for a given frequency for a single complex signal.
Discrete Hartley Transform Functions Hartley	Performs the discrete Hartley transform of a real signal.
Discrete Walsh-Hadamard Transform Functions WHTGetBufferSize	Computes the size of the working buffer for Walsh-Hadamard transform.
WHT	Performs Walsh-Hadamard transform of a real signal.
Discrete Cosine Transform Functions DCTFwdInitAlloc	Allocates memory and initializes the forward discrete cosine transform structure.
DCTInvInitAlloc	Allocates memory and initializes the inverse discrete cosine transform structure.
DCTFwdFree	Closes the forward discrete cosine transform structure.

Function Base Name	Operation
DCTInvFree	Closes the inverse discrete cosine transform structure.
DCTFwdGetBufSize	Computes the size of the forward DCT work buffer.
DCTInvGetBufSize	Computes the size of the inverse DCT work buffer.
DCTFwdInit	Initializes the forward discrete cosine transform structure.
DCTInvInit	Initializes the inverse discrete cosine transform structure.
DCTFwdGetSize	Computes the size of all buffers required for the forward DCT.
DCTInvGetSize	Computes the size of all buffers required for the inverse DCT.
DCTFwd	Computes the forward discrete cosine transform (DCT) of a signal.
DCTInv	Computes the inverse discrete cosine transform (DCT) of a signal.
DCT4InitAlloc	Allocates memory and initializes the DCT-IV specification structure.
DCT4Free	Closes the DCT-IV specification structure.
DCT4Init	Initializes the DCT-IV specification structure.
DCT4GetSize	Computes the size of the DCT-IV specification structure.
DCT4	Computes the type-IV discrete cosine transform (DCT-IV) of a signal.
Hilbert Transform Functions	
HilbertInitAlloc	Initializes the Hilbert transform structure.
HilbertFree	Closes a Hilbert transform structure.



Function Base Name	Operation
<code>Hilbert</code>	Computes an analytic signal using the Hilbert transform.
Wavelet Transform Functions	
<code>WTHaarFwd</code> , <code>WTHaarInv</code>	Performs forward or inverse single-level discrete wavelet Haar transforms.
<code>WTFwdInitAlloc</code> , <code>WTInvInitAlloc</code>	Initializes the wavelet transform structure.
<code>WTFwdFree</code> , <code>WTInvFree</code>	Closes a wavelet transform structure.
<code>WTFwd</code>	Computes the forward wavelet transform.
<code>WTFwdSetDlyLine</code> , <code>WTFwdGetDlyLine</code>	Sets and gets the delay lines of the forward wavelet transform.
<code>WTInv</code>	Computes the inverse wavelet transform.
<code>WTInvSetDlyLine</code> , <code>WTInvGetDlyLine</code>	Sets and gets the delay lines of the inverse wavelet transform.

## Fourier Transform Functions

The functions described in this section perform the fast Fourier transform (FFT), the discrete Fourier transform (DFT) of signal samples. It also includes variations of the basic functions to support different application requirements.

### Special Arguments

This section describes the flag and hint arguments used by the Fourier transform functions.

The Fourier transform functions require you to specify the *flag* and *hint* arguments.

The *flag* argument specifies the result normalization method. The following table 7-2 lists the possible values for the *flag* argument. Specify one and only one of the represented values in the *flag* argument. The **A** and **B** factors are multipliers used in the DFT computation.

**Table 7-2 . Flag Arguments for Fourier Transform Functions**

Value	A	B	Description
IPP_FFT_DIV_FWD_BY_N	1/N	1	Forward transform is done with the 1/N normalization.
IPP_FFT_DIV_INV_BY_N	1	1/N	Inverse transform is done with the 1/N normalization.
IPP_FFT_DIV_BY_SQRTN	1/N <sup>1/2</sup>	1/N <sup>1/2</sup>	Forward and inverse transform is done with the 1/ N <sup>1/2</sup> normalization.
IPP_FFT_NODIV_BY_ANY	1	1	Forward or inverse transform is done without the 1/ N or 1/N <sup>1/2</sup> normalization.

The *hint* argument suggests using special code, faster but less accurate calculation, or more accurate but slower calculation. The following table 7-3 lists values you can enter for the *hint* argument.

Note that the *hint* argument is also used for some other operations (for example, for tone generation, logarithmic addition).

**Table 7-3. Hint Arguments for Fourier Transform Functions**

Value	Description
ippAlgHintNone	No suggestions. The function selects an algorithm
ippAlgHintFast	Fast algorithm is suggested for computation.
ippAlgHintAccurate	Accurate algorithm is suggested for computation.

## Packed Formats

This section describes the main packed formats *Perm*, *Pack*, and *CCS* used by the Fourier transform functions.

### Pack Format

The *Pack* format is a convenient, compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real Fourier transform algorithms (“natural” in the sense that bit-reversed order is natural for radix-2

complex Fourier transforms). In `Pack` format, the output samples of the Fourier transform are arranged as shown in Tables 7-4 and 7-5below. The output signal can be unpacked to a complex signal using the function `ippsConjPack`.

Perm Format

The `Perm` format stores the values in the order in which the Fourier transform algorithms use them. This is the most natural way of storing values for the Fourier transform algorithms. The `Perm` format is an arbitrary permutation of the `Pack` format. An important characteristic of the `Perm` format is that the real and imaginary parts of a given sample need not be adjacent.

In `Perm` format, the output samples of the Fourier transform are arranged as shown in Tables 7-4 and 7-5below. The output signal can be unpacked to a complex signal using the function `ippsConjPerm`.



**NOTE.** For input signal of odd length the perm and pack format are identical.

CCS Format

The `CCS` format stores the values of the first half of the output complex signal resulted from the forward Fourier transform.

In `CCS` format, the output samples of the Fourier transform are arranged as shown in Tables 7-4 and 7-5below. Note that the signal stored in `CCS` format is one complex element longer. The output signal can be unpacked to a complex signal using the function `ippsConjCcs`.

Table 7-4 Arrangement of Forward Fourier Transform Results in Packed Formats - Even Length

Index	0	1	2	3	...	N-2	N-1	N	N+1
Pack	$R_0$	$R_1$	$I_1$	$R_2$	...	$I_{(N-1)/2}$	$R_{N/2}$		
Perm	$R_0$	$R_{N/2}$	$R_1$	$I_1$	...	$R_{N/2-1}$	$I_{N/2-1}$		
CCS	$R_0$	0	$R_1$	$I_1$	...	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0

Table 7-5 Forward Fourier transform Result Representation in Packed Formats - Odd Length

Index	0	1	2	3	...	N-2	N-1	N
Pack	$R_0$	$R_1$	$I_1$	$R_2$	...	$R_{(N-1)/2}$	$I_{(N-1)/2}$	
Perm	$R_0$	$R_1$	$I_1$	$R_2$	...	$R_{(N-1)/2}$	$I_{(N-1)/2}$	
CCS	$R_0$	0	$R_1$	$I_1$	...	$I_{(N-1)/2-1}$	$R_{(N-1)/2}$	$I_{(N-1)/2}$

## Format Conversion Functions

The following functions `ippsConjPack`, `ippsConjPerm`, and `ippsConjCcs` convert data from the packed formats to a usual complex data format using the FFT symmetry property for transforming real data. The output data is complex, the output array length is defined by the number of complex elements in the output vector. Note that the output array size is two times as big as the input array size. The data stored in CCS format require a bigger array than the other formats. Even and odd length arrays have some specific features discussed for each function separately.

### ConjPack

*Converts the data in Pack format to complex data format.*

---

#### Syntax

```

IppStatus ippsConjPack_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPack_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPack_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPack_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPack_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPack_64fc_I(Ipp64fc* pSrcDst, int lenDst);

```

#### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>lenDst</i>	Number of elements in the vector.

#### Description

The function `ippsConjPack` is declared in the `ipps.h` file. This function converts the data in format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjPack` converts the data in `Pack` format in the vector `pSrcDst` to complex data format and stores the results in `pSrcDst`.

Table 7-6below shows the examples of unpack from the `Pack` format. The `Data` column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the `Packed` column. The output result is the complex data vector in the `Extended` column. The number of vector elements is in the `Length` column.

**Table 7-6. Examples of Unpack from the Pack Format**

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -2, 7, -7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Code example 7-1below shows how to use the function `ippsConjPack`.

Return Values

- ippStsNoErr
- Indicates no error.
- ippStsNullPtrErr
- Indicates an error when the `pSrcDst`, `pDst`, or `pSrc` pointer is NULL.
- ippStsSizeErr
- Indicates an error when `lenDst` is less than or equal to 0.

Example 7-1 Using the ippsConjPack Function

```
void ConjPack(void) {
    Ipp16s x[8] = {1,2,3,5,6,7,8,9};
    Ipp16sc zero={0,0}, y[6];
    IppStatus st;
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjPack_16sc( x, y, 6 );
}
```

```

    printf_16sc("pack 6", y, 6, st );
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjPack_16sc( x, y, 5 );
    printf_16sc("pack 5", y, 5, st );
}

```

Output:

```

Pack 6: {1,0} {2,3} {5,6} {7,0} {5,-6} {2,-3}
Pack 5: {1,0} {2,3} {5,6} {5,-6} {2,-3}

```

## ConjPerm

*Converts the data in Perm format to complex data format.*

---

### Syntax

```

IppStatus ippsConjPerm_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjPerm_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjPerm_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjPerm_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjPerm_64fc_I(Ipp64fc* pSrcDst, int lenDst);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>lenDst</i>	Number of elements in the vector.

### Description

The function `ippsConjPerm` is declared in the `ipps.h` file. This function converts the data in [Perm](#) format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjPerm` converts the data in `Perm` format in the vector `pSrcDst` to complex data format and stores the results in `pSrcDst`.

The following table 7-7 shows the examples of unpack from the `Perm` format. The `Data` column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the `Packed` column. The output result is the complex data vector in the `Extended` column. The number of vector elements is in the `Length` column.

**Table 7-7 Examples of Packed Data Obtained by FFT**

Data	Packed	Extended	Length
FFT([1])	1	{1, 0}	1
FFT([1 2])	3, -1	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, -7, -2, 7	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Code example 7-2 below shows how to use the function `ippsConjPerm`

Return Values

- ippsStsNoErr
- Indicates no error.
- ippsStsNullPtrErr
- Indicates an error when the `pSrcDst`, `pDst`, or `pSrc` pointer is NULL.
- ippsStsSizeErr
- Indicates an error when `lenDst` is less than or equal to 0.

**Example 7-2 Using the ippsConjPerm Function**

```
void ConjPerm(void) {
    Ipp16s x[8] = {1,2,3,5,6,7,8,9};
    Ipp16sc zero={0,0}, y[6];
    IppStatus st;
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjPerm_16sc( x, y, 6 );
    printf_16sc("Perm 6:", y, 6, st );
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjPerm_16sc( x, y, 5 );
    printf_16sc("Perm 5:", y, 5, st );
}
```

Output:

```
Perm 6:  {1,0} {3,5} {6,7} {2,0} {6,-7} {3,-5}
Perm 5:  {1,0} {2,3} {5,6} {5,-6} {2,-3}
```

**ConjCcs**

*Converts the data in CCS format to complex data format.*

---

**Syntax**

```
IppStatus ippsConjCcs_16sc(const Ipp16s* pSrc, Ipp16sc* pDst, int lenDst);
IppStatus ippsConjCcs_32fc(const Ipp32f* pSrc, Ipp32fc* pDst, int lenDst);
IppStatus ippsConjCcs_64fc(const Ipp64f* pSrc, Ipp64fc* pDst, int lenDst);
IppStatus ippsConjCcs_16sc_I(Ipp16sc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_32fc_I(Ipp32fc* pSrcDst, int lenDst);
IppStatus ippsConjCcs_64fc_I(Ipp64fc* pSrcDst, int lenDst);
```

**Parameters**

*pSrc*                                      Pointer to the source vector.



<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>lenDst</i>	Number of elements in the vector.

Description

The function `ippsConjCcs` is declared in the `ipps.h` file. This function converts the data in **CCS** format in the vector *pSrc* to complex data format and stores the results in *pDst*.

The in-place function `ippsConjCcs` converts the data in **CCS** format in the vector *pSrcDst* to complex data format and stores the results in *pSrcDst*.

The following tble 7-8 shows the examples of unpack from the **CCS** format. The **Data** column contains the real input data to be converted by the forward FFT transform to the packed data. The packed real data are in the **Packed** column. The output result is the complex data vector in the **Extended** column. The number of vector elements is in the **Length** column.The data stored in **CCS** format are two real elements longer.

Table 7-8 Examples of Unpack from the CCS Format

Data	Packed	Extended	Length
FFT([1])	1, 0	{1, 0}	1
FFT([1 2])	3, 0, -1, 0	{3, 0}, {-1, 0}	2
FFT([1 2 3])	6, 0, -1.5, 0.86	{6, 0}, {-1.5, 0.86}, {-1.5, -0.86}	3
FFT([1 2 3 9])	15, 0, -2, 7, -7, 0	{15, 0}, {-2, 7}, {-7, 0}, {-2, -7}	4

Example 7-3 below shows how to use the function `ippsConjCcs`

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> , <i>pDst</i> , or <i>pSrc</i> pointer is <i>NULL</i> .
<code>ippStsSizeErr</code>	Indicates an error when <i>lenDst</i> is less than or equal to 0.

## Example 7-3 Using the ippsConjCcs Function

```
void ConjCcs(void) {
    Ipp16s x[8] = {1,2,3,5,6,7,8,9};
    Ipp16sc zero={0,0}, y[6];
    IppStatus st;
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjCcs_16sc( x, y, 6 );
    printf_16sc("CCS 6:", y, 6, st );
    ippsSet_16sc( zero, y, 6 );
    st = ippsConjCcs_16sc( x, y, 5 );
    printf_16sc("CCS 5:", y, 5, st );
}
```

Output:

```
CCS 6:  {1,2} {3,5} {6,7} {8,9} {6,-7} {3,-5}
CCS 5:  {1,2} {3,5} {6,7} {6,-7} {3,-5}
```

## Functions for Packed Data Multiplication

The functions described in this section perform the element-wise complex multiplication of vectors stored in [Pack](#) or [Perm](#) formats. These functions are used with the function `ippsFFTFwd` and `ippsFFTInv` to perform fast convolution on real signals.

The standard vector multiplication function `ippsMul` can not be used to multiply `Pack` or `Perm` format vectors because:

- Two real samples are stored in `Pack` format.
- The `Perm` format might not pair the real parts of a signal with their corresponding imaginary parts.




---

**NOTE.** The vectors stored in [CCS](#) format can be multiplied using the standard function for complex data multiplication.

---

## MulPack

*Multiply the elements of two vectors stored in Pack format.*

---

### Syntax

#### Case 1: Not-in-place operation

```
IppStatus ippsMulPack_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
```

```
IppStatus ippsMulPack_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, int len);
```

```
IppStatus ippsMulPack_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2, Ipp16s* pDst, int len, int scaleFactor);
```

#### Case 2: In-place operation

```
IppStatus ippsMulPack_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippsMulPack_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

```
IppStatus ippsMulPack_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len, int scaleFactor);
```

### Parameters

<i>pSrc1, pSrc2</i>	Pointers to the vectors whose elements are to be multiplied together.
<i>pDst</i>	Pointer to the destination vector which stores the result of the multiplication $pSrc1[n] * pSrc2[n]$ .
<i>pSrc</i>	Pointer to the vector whose elements are to be multiplied by the elements of <i>pSrcDst</i> in-place.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsMulPack` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors `ippsMulPack` multiply the elements of the vector `pSrc` by the elements of the vector `pSrcDst`, and store the result in `pSrcDst`.

The functions multiply the packed data according to their packed format. The data in [Pack](#) packed format include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

For the functions with the `Sfs` suffixes scaling is performed in accordance with the `scaleFactor` value. When the output value exceeds the data range, the result may become saturated.

Example 7-4 below shows how to use the function `ippsMulPack_32f_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> , <code>pDst</code> , <code>pSrc1</code> , <code>pSrc2</code> , or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

### Example 7-4 Using the ippsMulPack Function

```
void mulpack( void ) {
    Ipp32f x[8], X[8], h[8]={1.0f/3,1.0f/3,1.0f/3,0,0,0,0,0}, H[8];
    IppStatus st;
    IppsFFTSpec_R_32f* spec;
    st = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
        ippsAlgHintNone);
    ippsSet_32f( 3, x, 8 );
    x[3] = 5;
    st = ippsFFTFwd_RToPack_32f( x, X, spec, NULL );
    st = ippsFFTFwd_RToPack_32f( h, H, spec, NULL );
    ippsMulPack_32f_I( H, X, 8 );
    st = ippsFFTInv_PackToR_32f( X, x, spec, NULL );
    printf_32f("filtered =", x, 8, st );
    ippsFFTFree_R_32f( spec );
}
```

Output:

```
filtered =  3.0 3.0 3.0 3.666667 3.666667 3.666667 3.0 3.0
```

Matlab\* analog:

```
>> x=3*ones(1,8); x(4)=5;h=zeros(1,8); h(1:3)=1/3;
real(ifft(fft(x).*fft(h)))
```

## MulPerm

*Multiply the elements of two vectors stored in Perm format.*

### Syntax

#### Case 1: Not-in-place operation

```
IppStatus ippsMulPerm_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, int len);
```

```
IppStatus ippsMulPerm_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, int len);
```

```
IppStatus ippsMulPerm_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
Ipp16s* pDst, int len, int scaleFactor);
```

### Case 2: In-place operation

```
IppStatus ippsMulPerm_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsMulPerm_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsMulPerm_16s_ISfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
int scaleFactor);
```

### Parameters

<i>pSrc1, pSrc2</i>	Pointers to the vectors whose elements are to be multiplied together.
<i>pDst</i>	Pointer to the destination vector which stores the result of the multiplication <i>pSrc1</i> [n] * <i>pSrc2</i> [n].
<i>pSrc</i>	Pointer to the vector whose elements are to be multiplied by the elements of <i>pSrcDst</i> in-place.
<i>pSrcDst</i>	Pointer to the source and destination vector (for the in-place operation).
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsMulPerm` is declared in the `ipps.h` file. This function multiplies the elements of the vector *pSrc1* by the elements of the vector *pSrc2*, and stores the result in *pDst*.

The in-place flavors of `ippsMulPerm` multiply the elements of the vector *pSrc* by the elements of the vector *pSrcDst*, and store the result in *pSrcDst*.

The function multiplies the packed data according to their packed format. The data in [Perm](#) packed formats include several real values, the rest are complex. Thus, the function performs several real multiplication operations on real elements and complex multiplication operations on complex data. Such kind of packed data multiplication is usually used for signals filtering with the FFT transform when the element-wise multiplication is performed in the frequency domain.

For the functions with the *Sfs* suffixes scaling is performed in accordance with the *scaleFactor* value. When the output value exceeds the data range, the result may become saturated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> , <code>pDst</code> , <code>pSrc1</code> , <code>pSrc2</code> , or <code>pSrc</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## MulPackConj

*Multiplies elements of a vector by the elements of a complex conjugate vector stored in `Pack` format.*

### Syntax

```
IppStatus ippMulPackConj_32f_I(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
```

```
IppStatus ippMulPackConj_64f_I(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
```

### Parameters

<code>pSrc</code>	Pointer to the first source vector.
<code>pSrcDst</code>	Pointer to the second source and destination vector.
<code>len</code>	Number of elements in the vector.

### Description

The function `ippMulPackConj` is declared in the `ipps.h` file. This function multiplies the elements of a source vector `pSrc` by elements of the vector that is complex conjugate to the source vector `pSrcDst` and stores the results in `pSrcDst`. The function performs only in-place operations on data stored in `Pack` format.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Fast Fourier Transform Functions

The functions described in this section compute the forward and inverse fast Fourier transform of real and complex signals. The FFT is similar to the discrete Fourier transform (DFT) but is significantly faster. The length of the vector transformed by the FFT must be a power of 2.

To use the FFT functions, initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms. The amount of prior calculations is thus reduced and the overall performance increased.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method.

The FFT specification structure can be initialized by the functions `ippsFFTInitAlloc_R` or `ippsFFTInitAlloc_CR` that allocate memory and initialize the structure. Alternatively the structure can be initialized by the functions `ippsFFTInit_R`, `ippsFFTInit_C`. In this case the user must allocate memory for the FFT specification structure whose size should be computed previously using `ippsFFTGetSize_R`, `ippsFFTGetSize_C` functions.

The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in Perm, Pack, or CCS format.

You can speed up the FFT by using an external buffer. The external buffer increases performance because it allows to avoid allocation and deallocation of internal buffers and to store data in cache. If the FFT structure is initialized by the functions `FFTInitAlloc_R`, `FFTInitAlloc_C`, then the size of the external buffer must be previously computed by either `ippsFFTGetBufSize_R`, `ippsFFTGetBufSize_C`. If the FFT structure is initialized by the functions `ippsFFTInit_R`, `ippsFFTInit_C`, then the size of the external buffer is returned by these functions.

If the external buffer is not specified (correspondent parameter is set to `NULL`), then the FFT function itself allocates the memory needed for operation.

## FFTInitAlloc\_R, FFTInitAlloc\_C

*Allocates memory and initializes an FFT specification structure for real and complex signals.*

### Syntax

#### Case 1: Operation on real signal

```
IppStatus ippsFFTInitAlloc_R_16s(IppsFFTSpec_R_16s** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);
```



```
IppStatus ippsFFTInitAlloc_R_32s(IppsFFTSpec_R_32s** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_R_32f(IppsFFTSpec_R_32f** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_R_64f(IppsFFTSpec_R_64f** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_R_16s32s(IppsFFTSpec_R_16s32s** ppFFTSpec, int
order, int flag, IppHintAlgorithm hint);
```

### Case 2: Operation on complex signal

```
IppStatus ippsFFTInitAlloc_C_16s(IppsFFTSpec_C_16s** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_32s(IppsFFTSpec_C_32s** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_32f(IppsFFTSpec_C_32f** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_64f(IppsFFTSpec_C_64f** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_16sc(IppsFFTSpec_C_16sc** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_32sc(IppsFFTSpec_C_32sc** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_32fc(IppsFFTSpec_C_32fc** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);

IppStatus ippsFFTInitAlloc_C_64fc(IppsFFTSpec_C_64fc** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint);
```

### Parameters

<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .

<i>order</i>	FFT order. The input signal length is $N=2^{order}$ .
<i>ppFFTSpec</i>	Double pointer to the FFT specification structure to be created.

## Description

The functions `ippsFFTInitAlloc_R` and `ippsFFTInitAlloc_C` are declared in the `ipps.h` file. These functions allocate memory, create, and initialize the FFT specification structure *ppFFTSpec* with the following parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The *order* argument defines the transform length. Thus the input and output signals are  $2^{order}$ -length arrays.

**ippsFFTInitAlloc\_R.** This function allocates memory and initializes the real FFT specification structure.

**ippsFFTInitAlloc\_C.** This function allocates memory and initializes the complex FFT specification structure.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppFFTSpec</i> pointer is <code>NULL</code> .
<code>ippStsFftOrderErr</code>	Indicates an error when the <i>order</i> value is incorrect.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## FFTFree\_R, FFTFree\_C

*Closes an FFT specification structure for real and complex signals.*

---

### Syntax

#### Case 1: Operation on real signal

```

IppStatus ippsFFTFree_R_16s( IppsFFTSpec_R_16s* pFFTSpec);
IppStatus ippsFFTFree_R_32s( IppsFFTSpec_R_32s* pFFTSpec);
IppStatus ippsFFTFree_R_32f( IppsFFTSpec_R_32f* pFFTSpec);
IppStatus ippsFFTFree_R_64f( IppsFFTSpec_R_64f* pFFTSpec);

```

```
IppStatus ippsFFTFree_R_16s32s(IppsFFTSpec_R_16s32s* pFFTSpec);
```

### Case 2: Operation on complex signal

```
IppStatus ippsFFTFree_C_16s( IppsFFTSpec_C_16s* pFFTSpec);
```

```
IppStatus ippsFFTFree_C_32s(IppsFFTSpec_C_32s* pFFTSpec);
```

```
IppStatus ippsFFTFree_C_32f( IppsFFTSpec_C_32f* pFFTSpec);
```

```
IppStatus ippsFFTFree_C_64f( IppsFFTSpec_C_64f* pFFTSpec);
```

```
IppStatus ippsFFTFree_C_16sc( IppsFFTSpec_C_16sc* pFFTSpec);
```

```
IppStatus ippsFFTFree_C_32sc(IppsFFTSpec_C_32sc* pFFTSpec);
```

```
IppStatus ippsFFTFree_C_32fc( IppsFFTSpec_C_32fc* pFFTSpec);
```

```
IppStatus ippsFFTFree_C_64fc( IppsFFTSpec_C_64fc* pFFTSpec);
```

### Parameters

*pFFTSpec*                      Pointer to the FFT specification structure.

### Description

The function `ippsFFTFree` is declared in the `ipps.h` file. This function closes the FFT specification structure *pFFTSpec* initialized by the function `ippsFFTInitAlloc_C` or `ippsFFTInitAlloc_R`. Call `ippsFFTFree` after the transform is completed.

**ippsFFTFree\_C.** The function `ippsFFTFree_C` closes the complex FFT specification structure.

**ippsFFTFree\_R.** The function `ippsFFTFree_R` closes the real FFT specification structure.

### Return Values

`ippStsNoErr`                      Indicates no error.

`ippStsNullPtrErr`               Indicates an error when the *pFFTSpec* pointer is NULL.

`ippStsContextMatchErr`       Indicates an error when the specification identifier *pFFTSpec* is incorrect.

## FFTInit\_R, FFTInit\_C

*Initializes an FFT specification structure for real and complex signals.*

---

### Syntax

#### Case 1: Operation on real signal

```
IppStatus ippsFFTInit_R_16s(IppsFFTSpec_R_16s** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_R_32s(IppsFFTSpec_R_32s** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_R_32f(IppsFFTSpec_R_32f** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_R_64f(IppsFFTSpec_R_64f** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_R_16s32s(IppsFFTSpec_R_16s32s** ppFFTSpec, int order,
int flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

#### Case 2: Operation on complex signal

```
IppStatus ippsFFTInit_C_16s(IppsFFTSpec_C_16s** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_C_32s(IppsFFTSpec_C_32s** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_C_32f(IppsFFTSpec_C_32f** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_C_64f(IppsFFTSpec_C_64f** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_C_16sc(IppsFFTSpec_C_16sc** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_C_32sc(IppsFFTSpec_C_32sc** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsFFTInit_C_32fc(IppsFFTSpec_C_32fc** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

```
IppStatus ippsFFTInit_C_64fc(IppsFFTSpec_C_64fc** ppFFTSpec, int order, int
flag, IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);
```

## Parameters

<i>order</i>	FFT order. The input signal length is $N = 2^{\text{order}}$ .
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>ppFFTSpec</i>	Double pointer to the FFT specification structure to be created.
<i>pSpec</i>	Pointer to the area for the FFT specification structure.
<i>pSpecBuffer</i>	Pointer to the work buffer.

## Description

The functions `ippsFFTInit_R` and `ippsFFTInit_C` are declared in the `ipps.h` file. These functions initialize the FFT specification structure `ppFFTSpec` with the following parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The *order* argument defines the transform length. Thus the input and output signals are  $2^{\text{order}}$ -length arrays.

Before calling these functions the memory must be allocated for the FFT specification structure and the work buffer (if it is required). The size of the FFT specification structure and the work buffer must be computed by the functions `ippsFFTGetSize_R` or `ippsFFTGetSize_C`.

If the work buffer is not used, the parameter `pSpecBuffer` can be `NULL`. If the working buffer is used, the parameter `pSpecBuffer` can not be `NULL`.

**`ippsFFTInit_C`.** This function initializes the complex FFT specification structure.

**`ippsFFTInit_R`.** This function initializes the real FFT specification structure.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFftOrderErr</code>	Indicates an error when the <i>order</i> value is incorrect.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.

## FFTGetSize\_R, FFTGetSize\_C

*Computes sizes of the FFT specification structure and required working buffers.*

---

### Syntax

#### Case 1: Operation on real signal

```
IppStatus ippsFFTGetSize_R_16s(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_R_32s(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_R_32f(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_R_64f(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_R_16s32s(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

#### Case 2: Operation on complex signal

```
IppStatus ippsFFTGetSize_C_16s(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_32s(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_32f(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_64f(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_16sc(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_32sc(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_32fc(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

```
IppStatus ippsFFTGetSize_C_64fc(int order, int flag, IppHintAlgorithm hint,
int* pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

## Parameters

<i>order</i>	FFT order. The input signal length is $N = 2^{\text{order}}$ .
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>pSpecSize</i>	Pointer to the FFT specification structure size value.
<i>pSpecBufferSize</i>	Pointer to the buffer size value for FFT initialization function.
<i>pBufferSize</i>	Pointer to the size value of the FFT external work buffer.

## Description

The functions `ippsFFTGetSize_R` and `ippsFFTGetSize_C` are declared in the `ipps.h` file. These functions compute the size of FFT specification structure, the work buffer size for the FFT structure initialization functions `ippsFFTInit_R` and `ippsFFTInit_C` as well as the size of the FFT work buffer for the different flavors of `ippsFFTFwd` and `ippsFFTInv`. Their values in bytes are stored in *pSpecSize*, *pSpecBufferSize*, and *pBufferSize* respectively.

**ippsFFTGetSize\_R.** This function is used for real flavors of the FFT functions.

**ippsFFTGetSize\_C.** This function is used for complex flavors of the FFT functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsFftOrderErr</code>	Indicates an error if the <i>order</i> value is incorrect.
<code>ippStsFftFlagErr</code>	Indicates an error if the <i>flag</i> value is incorrect.

## FFTGetBufSize\_R, FFTGetBufSize\_C

*Computes the size of the FFT work buffer.*

---

### Syntax

#### Case 1: Operation on real signal

```
IppStatus ippsFFTGetBufSize_R_16s(const IppsFFTSpec_R_16s* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_R_32s(const IppsFFTSpec_R_32s* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_R_32f(const IppsFFTSpec_R_32f* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_R_64f(const IppsFFTSpec_R_64f* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_R_16s32s(const IppsFFTSpec_R_16s32s* pFFTSpec,
int* pBufferSize);
```

#### Case 2: Operation on complex signal

```
IppStatus ippsFFTGetBufSize_C_16s(const IppsFFTSpec_C_16s* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_C_32s(const IppsFFTSpec_C_32s* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_C_32f(const IppsFFTSpec_C_32f* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_C_64f(const IppsFFTSpec_C_64f* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_C_16sc(const IppsFFTSpec_C_16sc* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_C_32sc(const IppsFFTSpec_C_32sc* pFFTSpec, int*
pBufferSize);

IppStatus ippsFFTGetBufSize_C_32fc(const IppsFFTSpec_C_32fc* pFFTSpec, int*
pBufferSize);
```



```
IppStatus ippsFFTGetBufSize_C_64fc(const IppsFFTSpec_C_64fc* pFFTSpec, int*
pBufferSize);
```

### Parameters

<i>pFFTSpec</i>	Pointer to the FFT specification structure.
<i>pBufferSize</i>	Pointer to the size value of the FFT external work buffer.

### Description

The functions `ippsFFTGetBufSize_R` and `ippsFFTGetBufSize_C` are declared in the `ipps.h` file. These functions compute the size in bytes of the external work buffer that can be used for the functions that perform FFT, and stores the result in *pBufferSize*. To compute the size of the buffer the corresponding FFT specification structure *pFFTSpec* must be initialized.

**ippsFFTGetBufSize\_C.** This function is used to compute the size of the complex FFT work buffer.

**ippsFFTGetBufSize\_R.** This function is used to compute the size of the real FFT work buffer.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFFTSpec</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.

## FFTFwd\_CToC

*Computes the forward fast Fourier transform (FFT) of a complex signal.*

---

### Syntax

#### Case 1: Not-in-place operation on real data type

```
IppStatus ippsFFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsFFTSpec_C_16s* pFFTSpecx, int
scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_32s_Sfs(const Ipp32s* pSrcRe, const Ipp32s* pSrcIm,
Ipp32s* pDstRe, Ipp32s* pDstIm, const IppsFFTSpec_C_32s* pFFTSpec, int
scaleFactor, Ipp8u* pBuffer);
```

### **Case 2: Not-in-place operation on complex data type**

```
IppStatus ippsFFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, const
IppsFFTSpec_C_32sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

### **Case 3: In-place operation on real data type**

```
IppStatus ippsFFTFwd_CToC_32f_I(Ipp32f* pSrcDstRe, Ipp32f* pSrcDstIm, const
IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64f_I(Ipp64f* pSrcDstRe, Ipp64f* pSrcDstIm, const
IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_16s_ISfs(Ipp16s* pSrcDstRe, Ipp16s* pSrcDstIm,
const IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_32s_ISfs(Ipp32s* pSrcDstRe, Ipp32s* pSrcDstIm,
const IppsFFTSpec_C_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

### **Case 4: In-place operation on complex data type**

```
IppStatus ippsFFTFwd_CToC_32fc_I(Ipp32fc* pSrcDst, const IppsFFTSpec_C_32fc*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_CToC_64fc_I(Ipp64fc* pSrcDst, const IppsFFTSpec_C_64fc*
pFFTSpec, Ipp8u* pBuffer);
```

```

IppStatus ippsFFTFwd_CToC_16sc_ISfs(Ipp16sc* pSrcDst, const
IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_CToC_32sc_ISfs(Ipp32sc* pSrcDst, const
IppsFFTSpec_C_32sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

```

## Parameters

<i>pFFTSpec</i>	Pointer to the FFT specification structure.
<i>pSrc</i>	Pointer to the input array containing complex values.
<i>pDst</i>	Pointer to the output array containing complex values.
<i>pSrcRe</i>	Pointer to the input array containing real parts of the signal.
<i>pSrcIm</i>	Pointer to the input array containing imaginary parts of the signal.
<i>pDstRe</i>	Pointer to the output array containing real parts of the signal.
<i>pDstIm</i>	Pointer to the output array containing imaginary parts of the signal.
<i>pSrcDst</i>	Pointer to the input and output array containing complex values (for the in-place operation).
<i>pSrcDstRe</i>	Pointer to the input and output array containing real parts of the signal(for the in-place operation).
<i>pSrcDstIm</i>	Pointer to the input and output array containing imaginary parts of the signal(for the in-place operation).
<i>pBuffer</i>	Pointer to the external work buffer, can be NULL.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsFFTFwd_CToC` is declared in the `ipps.h` file. This function computes the forward FFT of a complex signal according to the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure must be initialized by the functions `ippsFFTInitAl-loc_C` or `ippsFFTInit_C`.

The functions using the complex data type, for example with the `32fc` suffixes, process the input complex array `pSrc` and store the result in `pDst`. Their in-place flavors use the complex array `pSrcDst`.

The functions using the real data type and processing complex signals represented by separate real `pSrcRe` and imaginary `pSrcIm` parts, for example with the `32f` suffixes, store the result separately in `pDstRe` and `pDstIm`, respectively. Their in-place flavors use separate real and imaginary arrays `pSrcDstRe` and `pSrcDstIm`, respectively.

For integer data types the output result is scaled according to the `scaleFactor` value, thus the output signal range and precision are retained.

The function may be used with the external work buffer `pBuffer` that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing FFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of the external buffer must be previously computed by the function `ippsFFTGetBufSize_C` or `ippsFFTGetSize_C`.

If the external buffer is not specified (`pBuffer` is set to `NULL`), then the function itself allocates the memory needed for operation.

The length of the FFT must be a power of 2.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <code>pBuffer</code> is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <code>pFFTSpec</code> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## FFTInv\_CToC

*Computes the inverse fast Fourier transform (FFT) of a complex signal.*

---

### Syntax

#### Case 1: Not-in-place operation on real data type

```
IppStatus ippsFFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsFFTSpec_C_32f* pFFTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsFFTSpec_C_64f* pFFTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsFFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsFFTSpec_C_16s* pFFTSpec, int
scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_32s_Sfs(const Ipp32s* pSrcRe, const Ipp32s* pSrcIm,
Ipp32s* pDstRe, Ipp32s* pDstIm, const IppsFFTSpec_C_32s* pFFTSpec, int
scaleFactor, Ipp8u* pBuffer);
```

#### Case 2: Not-in-place operation on complex data type

```
IppStatus ippsFFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
IppsFFTSpec_C_32fc* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
IppsFFTSpec_C_64fc* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_32sc_Sfs(const Ipp32sc* pSrc, Ipp32sc* pDst, const
IppsFFTSpec_C_32sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

#### Case 3: In-place operation on real data type

```
IppStatus ippsFFTInv_CToC_32f_I(Ipp32f* pSrcDstRe, Ipp32f* pSrcDstIm, const
IppsFFTSpec_C_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64f_I(Ipp64f* pSrcDstRe, Ipp64f* pSrcDstIm, const
IppsFFTSpec_C_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_16s_ISfs(Ipp16s* pSrcDstRe, Ipp16s* pSrcDstIm,
const IppsFFTSpec_C_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_32s_ISfs(Ipp32s* pSrcDstRe, Ipp32s* pSrcDstIm,
const IppsFFTSpec_C_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

#### Case 4: In-place operation on complex data type

```
IppStatus ippsFFTInv_CToC_32fc_I(Ipp32fc* pSrcDst, const IppsFFTSpec_C_32fc*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_64fc_I(Ipp64fc* pSrcDst, const IppsFFTSpec_C_64fc*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_16sc_ISfs(Ipp16sc* pSrcDst, const
IppsFFTSpec_C_16sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CToC_32sc_ISfs(Ipp32sc* pSrcDst, const
IppsFFTSpec_C_32sc* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

#### Parameters

<i>pFFTSpec</i>	Pointer to the FFT specification structure.
<i>pSrc</i>	Pointer to the input array containing complex values.
<i>pDst</i>	Pointer to the output array containing complex values.
<i>pSrcRe</i>	Pointer to the input array containing real parts of the signal.
<i>pSrcIm</i>	Pointer to the input array containing imaginary parts of the signal.
<i>pDstRe</i>	Pointer to the output array containing real parts of the signal.
<i>pDstIm</i>	Pointer to the output array containing imaginary parts of the signal.
<i>pSrcDst</i>	Pointer to the input and output array containing complex values (for the in-place operation).
<i>pSrcDstRe</i>	Pointer to the input and output array containing real parts of the signal(for the in-place operation).
<i>pSrcDstIm</i>	Pointer to the input and output array containing imaginary parts of the signal(for the in-place operation).
<i>pBuffer</i>	Pointer to the external work buffer, can be NULL.

*scaleFactor*                      Scale factor, refer to [Integer Scaling](#).

## Description

The function `ippsFFTInv_CToC` is declared in the `ipps.h` file. This function computes the inverse FFT of a complex signal according to the `pFFTSpec` specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. The FFT specification structure must be initialized by the functions `ippsFFTInitAlloc_C` or `ippsFFTInit_C` beforehand.

The function flavors using the complex data type, for example with the `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*. Their in-place flavors use the complex array *pSrcDst*.

The function flavors using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, for example with the `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively. Their in-place flavors uses separate real and imaginary arrays *pSrcDstRe* and *pSrcDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing FFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of the external buffer must be previously computed by the function `ippsFFTGetBufSize_C` or `ippsFFTGetSize_C`.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

The length of the FFT must be a power of 2.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <i>pBuffer</i> is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## FFTFwd\_RToPack, FFTFwd\_RToPerm, FFTFwd\_RToCCS

*Computes the forward or inverse fast Fourier transform (FFT) of a real signal.*

---

### Syntax

#### Case 1: Not-in-place operation, result in `pack` Format

```

IppStatus ippsFFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPack_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

```

#### Case 2: In-place operation, result in `pack` Format

```

IppStatus ippsFFTFwd_RToPack_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPack_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPack_16s_ISfs(Ipp16s* pSrcDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPack_32s_ISfs(Ipp32s* pSrcDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

```

#### Case 3: Not-in-place operation, result in `perm` Format

```

IppStatus ippsFFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsFFTFwd_RToPerm_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);

```



**Case 4: In-place operation, result in perm Format**

```
IppStatus ippsFFTFwd_RToPerm_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_64f_I(Ipp64f* pSrcDst const IppsFFTSpec_R_64f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_16s_ISfs(Ipp16s* pSrcDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToPerm_32s_ISfs(Ipp32s* pSrcDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

**Case 5: Not-in-place operation, result in ccs Format**

```
IppStatus ippsFFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst,
const IppsFFTSpec_R_16s32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

**Case 6: In-place operation, result in ccs Format**

```
IppStatus ippsFFTFwd_RToCCS_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTFwd_RToCCS_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

**Parameters**

*pFFTSpec*                      Pointer to the FFT specification structure.

<i>pSrc</i>	Pointer to the input array.
<i>pDs</i>	Pointer to the output array containing packed complex values.
<i>pSrcDst</i>	Pointer to the input and output arrays for the in-place operation.
<i>pBuffer</i>	Pointer to the external work buffer, can be <code>NULL</code> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The functions `ippsFFTFwd_RTtoPerm`, `ippsFFTFwd_RTtoPack`, and `ippsFFTFwd_RTtoCCS` are declared in the `ipps.h` file. They compute the forward FFT of a real signal and store the result in [Pack](#), [Perm](#), or [CCS](#) packed formats respectively. The transform is performed in accordance with the *pFFTSpec* specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure must be initialized by the corresponding flavors of either [ippsFFTInitAlloc\\_R](#) or [ippsFFTInitc\\_R](#). The length of the FFT must be a power of 2.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing FFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of the external buffer must be previously computed by the function [ippsFFTGetBufSize\\_R](#) or [ippsFFTGetSize\\_R](#).

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

**`ippsFFTFwd_RTtoPack`.** This function computes the forward FFT and stores the result in [Pack](#) format.

**`ippsFFTFwd_RTtoPerm`.** This function computes the forward FFT and stores the result in [Perm](#) format.

**`ippsFFTFwd_RTtoCCS`.** This function computes the forward FFT and stores the result in [CCS](#) format.

Tables 7-4 and 7-5 show how the output results are arranged in the packed formats. The code example 7-5 below shows how to initialize the specification structure and call the function `ippsFFTFwd_RToCCS_32f`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <i>pBuffer</i> is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

**Example 7-5. Using the ippsFFTFwd\_RToCC Function**

```

IppStatus fft( void ) {
    Ipp32f x[8], X[10];

    int n;

    IppStatus status;

    IppsFFTSpec_R_32f* spec;

    status = ippsFFTInitAlloc_R_32f(&spec, 3, IPP_FFT_DIV_INV_BY_N,
        ippAlgHintNone );

    for(n=0; n<8; ++n) x[n] = (float)cos(IPP_2PI *n *16/64);

    status = ippsFFTFwd_RToCCS_32f( x, X, spec, NULL );

    ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );

    ippsFFTFree_R_32f( spec );

    printf_32f("fft magn =", x, 4, status );

    return status;
}

```

Output:

```
fft magn = 0.000000 0.000000 4.000000 0.000000
```

Matlab\* Analog:

```
>> n=0:7; x=sin(2*pi*n*16/64); X=abs(fft(x)); X(1:4)
```

**FFTInv\_PackToR, FFTInv\_PermToR, FFTInv\_CCSToR**

*Computes the inverse fast Fourier transform (FFT)  
of a real signal.*

---

**Syntax****Case 1: Not-in-place operation on input data in pack format**

```

IppStatus ippsFFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);

IppStatus ippsFFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);

```

```
IppStatus ippsFFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PackToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

### Case 2: In-place operation on input data in `pack` format

```
IppStatus ippsFFTInv_PackToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PackToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PackToR_16s_ISfs(Ipp16s* pSrcDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PackToR_32s_ISfs(Ipp32s* pSrcDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

### Case 3: Not-in-place operation on input data in `perm` format

```
IppStatus ippsFFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

### Case 4: In-place operation on input data in `perm` format

```
IppStatus ippsFFTInv_PermToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_16s_ISfs(Ipp16s* pSrcDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_PermToR_32s_ISfs(Ipp16s* pSrcDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

**Case 5 Not-in-place operation on input data in ccs format**

```
IppStatus ippsFFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFFTSpec_R_32f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsFFTSpec_R_64f* pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsFFTSpec_R_16s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
IppsFFTSpec_R_32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst,
const IppsFFTSpec_R_16s32s* pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

**Case 6: In-place operation on input data in ccs format**

```
IppStatus ippsFFTInv_CCSToR_32f_I(Ipp32f* pSrcDst, const IppsFFTSpec_R_32f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_64f_I(Ipp64f* pSrcDst, const IppsFFTSpec_R_64f*
pFFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_16s_ISfs(Ipp16s* pSrcDst, const IppsFFTSpec_R_16s*
pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsFFTInv_CCSToR_32s_ISfs(Ipp32s* pSrcDst, const IppsFFTSpec_R_32s*
pFFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

**Parameters**

<i>pFFTSpec</i>	Pointer to the FFT specification structure.
<i>pSrc</i>	Pointer to the input array.
<i>pDst</i>	Pointer to the output array containing real values.
<i>pSrcDst</i>	Pointer to the input and output for the in-place operation.
<i>pBuffer</i>	Pointer to the external work buffer, can be NULL.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The functions `ippsFFTInv_PermToR`, `ippsFFTInv_PackToR`, and `ippsFFTInv_CCSToR` are declared in the `ipps.h` file. They compute the inverse FFT of data stored in **Pack**, **Perm**, or **CCS** packed formats respectively. The transform is performed in accordance with the `pFFTSpec` specification parameters: the transform *order*, the normalization *flag*, and the specific code *hint*. Before calling these functions the FFT specification structure must be initialized by the corresponding flavors of the functions `ippsFFTInitAlloc_R` or `ippsFFTInit_R`. The length of the FFT must be a power of 2.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing FFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of the external buffer must be previously computed by the function `ippsFFTGetBufSize_R` or `ippsFFTGetSize_R`.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

**ippsFFTInv\_PackToR.** This function computes the inverse FFT of input data in **Pack** format.

**ippsFFTInv\_PermToR.** This function computes the inverse FFT of input data in **Perm** format.

**ippsFFTInv\_CCSToR.** This function computes the inverse FFT of input data in **CCS** format.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <i>pBuffer</i> is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## Discrete Fourier Transform Functions

The functions described in this section compute the forward and inverse discrete Fourier transform of real and complex signals. The DFT is less efficient than the fast Fourier transform, however the length of the vector transformed by the DFT can be arbitrary.

The *hint* argument, passed to the initialization functions, suggests using special algorithm, faster or more accurate. The *flag* argument specifies the result normalization method. The complex signal can be represented as a single array containing complex elements, or two separate arrays containing real and imaginary parts. The output result of the FFT can be packed in [Pack](#), [Perm](#), or [CCS](#) formats.

To use the DFT functions, you should initialize the specification structure which contains such data as tables of twiddle factors. The initialization functions create the specifications for both forward and inverse transforms.

For more information about the fast computation of the discrete Fourier transform, see [\[Mit93\]](#), section 8-2, *Fast Computation of the DFT*.

A special set of Intel IPP functions provides the so called "out-of-order" DFT of the complex signal. In this case, the elements in frequency domain for both forward and inverse transforms can be re-ordered to speed-up the computation of the transforms. This re-ordering is hidden from the user and can be different in different implementations of the functions. However, reversibility of each pair of functions for forward/inverse transforms is ensured.

## DFTInitAlloc\_R, DFTInitAlloc\_C

*Initializes the DFT specification structure for real and complex signals.*

---

### Syntax

#### Case 1: Operation on real signal

```
IppStatus ippsDFTInitAlloc_R_16s(IppsDFTSpec _R_16s** ppDFTSpec, int len,
int flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_R_32f(IppsDFTSpec _R_32f** ppDFTSpec, int len,
int flag, IppHintAlgorithm hint);
```

```
IppStatus ippsDFTInitAlloc_R_64f(IppsDFTSpec _R_64f** ppDFTSpec, int len, int
flag, IppHintAlgorithm hint);
```



**Case 2: Operation on complex signal**

```

IppStatus ippsDFTInitAlloc_C_16s(IppsDFTSpec _C_16s** ppDFTSpec, int len,
int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_C_32f(IppsDFTSpec _C_32f** ppDFTSpec, int len,
int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_C_64f(IppsDFTSpec _C_64f** ppDFTSpec, int len,
int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_C_16sc(IppsDFTSpec _C_16sc** ppDFTSpec, int len,
int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_C_32fc(IppsDFTSpec _C_32fc** ppDFTSpec, int len,
int flag, IppHintAlgorithm hint);

IppStatus ippsDFTInitAlloc_C_64fc(IppsDFTSpec _C_64fc** ppDFTSpec, int len,
int flag, IppHintAlgorithm hint);

```

**Parameters**

<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>len</i>	Length of the DFT transform.
<i>ppDFTSpec</i>	Double pointer to the DFT specification structure to be created.

**Description**

The functions `ippsDFTInitAlloc_R` and `ippsDFTInitAlloc_C` are declared in the `ipps.h` file. These functions create and initialize the DFT specification structure `ppDFTSpec` with the following parameters: the transform `len`, the normalization `flag`, and the specific code `hint`. The `len` argument defines the transform length.

**ippsDFTInitAlloc\_R.** The function `ippsDFTInitAlloc_R` initializes the real DFT specification structure.

**ippsDFTInitAlloc\_C.** The function `ippsDFTInitAlloc_C` initializes the complex DFT specification structure.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppDFTSpec</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## DFTFree\_R, DFTFree\_C

*Closes the DFT specification structure for real and complex signals.*

---

### Syntax

#### Case 1: Operation on real signal

```
IppStatus ippDFTFree_R_16s( IppsDFTSpec_R_16s* pDFTSpec);
IppStatus ippDFTFree_R_32f( IppsDFTSpec_R_32f* pDFTSpec);
IppStatus ippDFTFree_R_64f( IppsDFTSpec_R_64f* pDFTSpec);
```

#### Case 2: Operation on complex signal

```
IppStatus ippDFTFree_C_16s( IppsDFTSpec_C_16s* pDFTSpec);
IppStatus ippDFTFree_C_32f( IppsDFTSpec_C_32f* pDFTSpec);
IppStatus ippDFTFree_C_64f( IppsDFTSpec_C_64f* pDFTSpec);
IppStatus ippDFTFree_C_16sc( IppsDFTSpec_C_16sc* pDFTSpec);
IppStatus ippDFTFree_C_32fc( IppsDFTSpec_C_32fc* pDFTSpec);
IppStatus ippDFTFree_C_64fc( IppsDFTSpec_C_64fc* pDFTSpec);
```

### Parameters

*pDFTSpec*                      Pointer to the DFT specification structure.

### Description

The function `ippDFTFree` is declared in the `ipp.h` file. This function closes the DFT specification structure *pDFTSpec* initialized by the function `ippDFTInitAlloc_C` or `ippDFTInitAlloc_R`. Call `ippDFTFree` after the transform is completed.

**ippsDFTFree\_R.** The function `ippsDFTFree_R` closes the real DFT specification structure.

**ippsDFTFree\_C.** The function `ippsDFTFree_C` closes the complex DFT specification structure.

## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when the `pDFTSpec` pointer is NULL.

`ippStsContextMatchErr` Indicates an error when the specification identifier `pDFTSpec` is incorrect.

## DFTGetBufSize\_R, DFTGetBufSize\_C

*Computes the size of the DFT work buffer.*

### Syntax

#### Case 1: Operation on real signal

```
IppStatus ippsDFTGetBufSize_R_16s(const IppsDFTSpec_R_16s* pDFTSpec, int* pSize);
```

```
IppStatus ippsDFTGetBufSize_R_32f(const IppsDFTSpec_R_32f* pDFTSpec, int* pSize);
```

```
IppStatus ippsDFTGetBufSize_R_64f(const IppsDFTSpec_R_64f* pDFTSpec, int* pSize);
```

#### Case 2: Operation on complex signal

```
IppStatus ippsDFTGetBufSize_C_16s(const IppsDFTSpec_C_16s* pDFTSpec, int* pSize);
```

```
IppStatus ippsDFTGetBufSize_C_32f(const IppsDFTSpec_C_32f* pDFTSpec, int* pSize);
```

```
IppStatus ippsDFTGetBufSize_C_64f(const IppsDFTSpec_C_64f* pDFTSpec, int* pSize);
```

```
IppStatus ippsDFTGetBufSize_C_16sc(const IppsDFTSpec_C_16sc* pDFTSpec, int* pSize);
```

```
IppStatus ippsDFTGetBufSize_C_32fc(const IppsDFTSpec_C_32fc* pDFTSpec, int* pSize);
```

```
IppStatus ippsDFTGetBufSize_C_64fc(const IppsDFTSpec_C_64fc* pDFTSpec, int* pSize);
```

## Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSize</i>	Pointer to the DFT work buffer size value.

## Description

The functions `ippsDFTGetBufSize_R` and `ippsDFTGetBufSize_C` are declared in the `ipps.h` file. These functions compute the size in bytes of an external memory buffer for the DFT described by the specification structure *pDFTSpec* and stores it in *pSize*.

To use external buffering, call this function after *pDFTSpec* initialization.

**ippsDFTGetBufSize\_R.** The function `ippsDFTGetBufSize_R` gets the size of the real DFT work buffer.

**ippsDFTGetBufSize\_C.** The function `ippsDFTGetBufSize_C` gets the size of the complex DFT work buffer.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDFTSpec</i> is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.

## DFTFwd\_CToC

*Computes the forward discrete Fourier transform of a complex signal.*

---

### Syntax

#### Case 1: Operation on real data type

```
IppStatus ippsDFTFwd_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f* pDFTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsDFTFwd_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s* pDFTSpec, int
scaleFactor, Ipp8u* pBuffer);
```

### Case 2: Operation on complex data type

```
IppStatus ippsDFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

### Case 3: Operation on complex data type with fixed length of DFT

```
IppStatus ippgDFTFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
int flag);
```

```
IppStatus ippgDFTFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
int flag);
```

```
IppStatus ippgDFTFwd_CToC_<len>_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
flag);
```

```
IppStatus ippgDFTFwd_CToC_<len>_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int
flag);
```

supported value for <len>: integer in the range [2, 64].

### Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSrc</i>	Pointer to the input array containing complex values.
<i>pDst</i>	Pointer to the output array containing complex values.
<i>pSrcRe</i>	Pointer to the input array containing real parts of the signal.
<i>pSrcIm</i>	Pointer to the input array containing imaginary parts of the signal.

<i>pDstRe</i>	Pointer to the output array containing real parts of the signal.
<i>pDstIm</i>	Pointer to the output array containing imaginary parts of the signal.
<i>pBuffer</i>	Pointer to the work buffer, can be <code>NULL</code> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>len</i>	Length of the DFT transform in range [2, 64].

## Description

The functions `ippsDFTFwd_CToC` and `ippgDFTFwd_CToC` are declared in the `ipps.h` file. These functions compute the forward DFT of a complex signal.

**Case 1 and Case 2.** The function flavors `ippsDFTFwd_CToC` computes the forward DFT according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

The functions operating on the complex data type process the input complex array *pSrc* and store the result in *pDst*.

The functions operating on the real data type (processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts) store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function can be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

Required buffer size must be computed by the corresponding function `ippsDFTGet_BufSize_C` prior to using DFT computation functions. If a null pointer is passed, memory will be allocated by the DFT computation functions internally.

**Case 3.** The function flavors `ippgDFTFwd_CToC` and `ippsDFTFwd_CToC_<len>` compute the forward DFT of the fixed length. They do not need the DFT specification structure. The length of transform can be specified by the parameter *len*, or by choosing the function flavor designed

for the specific length of transform and containing its value in the function name, for example, the function flavor `ippgDFTFwd_CToC_24_32fc` performs the forward DFT with length 24. Intel IPP support functions for the forward DFT with the fixed length in the range [2, 64].



**CAUTION.** Data vectors for these functions must be aligned to 16 bytes for the `w7` (and above) processor-specific library.

To avoid listing all the similar prototypes of function flavors of this type, only templates are given here. In the template the length of DFT is denoted by the modifier `<len>` and can be varied in the range of integers [2, 64].

The normalization method of the result is specified by the parameter `flag`.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi \frac{kn}{N}\right),$$

where  $k$  is the index of elements in the frequency domain,  $n$  is the index of elements in the time domain,  $N$  is the input signal `len`, and  $A$  is a multiplier defined by `flag`. Also,  $x(n)$  is `pSrc[n]` and  $X(k)$  is `pDst[k]`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <code>pBuffer</code> is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <code>pDFTSpec</code> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsFftFlagErr</code>	Indicates an error when the <code>flag</code> value is incorrect.

## DFTInv\_CToC

*Computes the inverse discrete Fourier transform of a complex signal.*

---

### Syntax

#### Case 1: Operation on real data type

```
IppStatus ippsDFTInv_CToC_32f(const Ipp32f* pSrcRe, const Ipp32f* pSrcIm,
Ipp32f* pDstRe, Ipp32f* pDstIm, const IppsDFTSpec_C_32f* pDFTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsDFTInv_CToC_64f(const Ipp64f* pSrcRe, const Ipp64f* pSrcIm,
Ipp64f* pDstRe, Ipp64f* pDstIm, const IppsDFTSpec_C_64f* pDFTSpec, Ipp8u*
pBuffer);
```

```
IppStatus ippsDFTInv_CToC_16s_Sfs(const Ipp16s* pSrcRe, const Ipp16s* pSrcIm,
Ipp16s* pDstRe, Ipp16s* pDstIm, const IppsDFTSpec_C_16s* pDFTSpec, int
scaleFactor, Ipp8u* pBuffer);
```

#### Case 2: Operation on complex data type

```
IppStatus ippsDFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, const
IppsDFTSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, const
IppsDFTSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CToC_16sc_Sfs(const Ipp16sc* pSrc, Ipp16sc* pDst, const
IppsDFTSpec_C_16sc* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

#### Case 3: Operation on complex data type with fixed length of DFT

```
IppStatus ippgDFTInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int len,
int flag);
```

```
IppStatus ippgDFTInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst, int len,
int flag);
```

```
IppStatus ippgDFTInv_CToC_<len>_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst, int
flag);
```

```
IppStatus ippgDFTInvd_CToC_<len>_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
int flag);
```

supported value for <len>: integer in the range [2, 64].



## Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSrc</i>	Pointer to the input array containing complex values.
<i>pDst</i>	Pointer to the output array containing complex values.
<i>pSrcRe</i>	Pointer to the input array containing real parts of the signal.
<i>pSrcIm</i>	Pointer to the input array containing imaginary parts of the signal.
<i>pDstRe</i>	Pointer to the output array containing real parts of the signal.
<i>pDstIm</i>	Pointer to the output array containing imaginary parts of the signal.
<i>pBuffer</i>	Pointer to the work buffer, can be <code>NULL</code> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>len</i>	Length of the DFT transform in range [2, 64].

## Description

The functions `ippsDFTInv_CToC` and `ippgDFTInv_CToC` are declared in the `ipps.h` file. These functions compute the inverse DFT of a complex signal.

**Case 1 and Case 2.** The function flavors `ippsDFTInv_CToC` compute the inverse DFT according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

The functions using the complex data type, for example with `32fc` suffixes, process the input complex array *pSrc* and store the result in *pDst*.

The functions using the real data type and processing complex signals represented by separate real *pSrcRe* and imaginary *pSrcIm* parts, for example with `32f` suffixes, store the result separately in *pDstRe* and *pDstIm*, respectively.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function can be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

Required buffer size must be computed by the corresponding function `ippsDFTGet_BufSize_C` prior to using DFT computation functions.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

**Case 3.** The function flavors `ippgDFTInv_CToC` and `ippgDFTInvd_CToC_<len>` compute the forward DFT of the fixed length. They do not need the DFT specification structure. The length of transform can be specified by the parameter *len*, or by choosing the function flavor designed for the specific length of transform and containing its value in the function name, for example, the function flavor `ippgDFTInv_CToC_24_32fc` performs the inverse DFT with the length 24. Intel IPP support functions for the inverse DFT with the fixed length in the range [2, 64].



**CAUTION.** Data vectors for these functions must be aligned to 16 bytes for the w7 (and above) processor-specific library.

To avoid listing all the similar prototypes of function flavors of this type, only templates are given here. In the template the length of DFT is denoted by the modifier *<len>* and can be varied in the range of integers [2, 64].

The normalization method of the result is specified by the parameter *flag*.

The inverse DFT functionality can be described as follows:

$$x(n) = B \sum_{k=0}^{N-1} X(k) \cdot \exp\left(j2\pi \frac{kn}{N}\right),$$

,

where *k* is the index of elements in the frequency domain, *n* is the index of elements in the time domain, *N* is the input signal *len*, and *B* is a multiplier defined by *flag*. Also, *x(n)* is *pDst[n]* and *X(k)* is *pSrc[k]*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <code>pBuffer</code> is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <code>pDFTSpec</code> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsFftFlagErr</code>	Indicates an error when the <code>flag</code> value is incorrect.

## DFTFwd\_RToPack, DFTFwd\_RToPerm, DFTFwd\_RToCCS

*Computes the forward discrete Fourier transform of a real signal.*

---

### Syntax

#### Case 1: Result in `pack` format

```
IppStatus ippSDFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippSDFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippSDFTFwd_RToPack_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

#### Case 2: Result in `pack` format with fixed length of DFT

```
IppStatus ippgDFTFwd_RToPack_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTFwd_RToPack_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTFwd_RToPack_<len>_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
flag);
```

```
IppStatus ippgDFTFwd_RToPack_<len>_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
flag);
```

supported value for `<len>`: integer in the range [2, 64].

**Case 3: Result in perm format**

```
IppStatus ippsDFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToPerm_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

**Case 4: Result in perm format with fixed length of DFT**

```
IppStatus ippgDFTFwd_RToPerm_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTFwd_RToPerm_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTFwd_RToPerm_<len>_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
flag);
```

```
IppStatus ippgDFTFwd_RToPerm_<len>_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
flag);
```

supported value for <len>: integer in the range [2, 64].

**Case 5: Result in ccs format**

```
IppStatus ippsDFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTFwd_RToCCS_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

**Case 6: Result in ccs format with fixed length of DFT**

```
IppStatus ippgDFTFwd_RToCCS_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTFwd_RToCCS_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTFwd_RToCCS_<len>_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
flag);
```

```
IppStatus ippgDFTFwd_RToCCS_<len>_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
flag);
```

supported value for *<len>*: integer in the range [2, 64].

## Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSrc</i>	Pointer to the input array containing real values .
<i>pDst</i>	Pointer to the output array containing packed complex values.
<i>pBuffer</i>	Pointer to the work buffer, can be <code>NULL</code> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>len</i>	Length of the DFT transform in range [2, 64].

## Description

The functions `ippsDFTFwd` and `ippgDFTFwd` are declared in the `ipps.h` file. These functions compute the forward DFT of a real signal. The result of the forward transform (that is in the frequency-domain) of real signals is represented in several possible packed formats: [Pack](#), [Perm](#), or [CCS](#). The data can be packed due to the symmetry property of the DFT transform of a real signal. [Tables 7-4 and 7-5](#) show how the output results are arranged in the packed formats.

**Case 1, Case3, Case 5.** These functions compute the forward DFT according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

These functions can be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

Required buffer size must be computed by the corresponding function `ippsDFTGetBufSize_R` beforehand.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

**Case 2, Case 4, Case 6.** These functions compute the forward DFT of the fixed length. They do not need the DFT specification structure. The length of transform can be specified by the parameter *len*, or by choosing the function flavor designed for the specific length of transform and containing its value in the function name, for example, the function flavor `ippgDFT-Fwd_RTtoCCS_33_64f` performs the forward DFT with length **33**. Intel IPP support functions for the forward DFT with the fixed length in the range [2, 64] .



**CAUTION.** Data vectors for these functions must be aligned to 16 bytes for the w7 (and above) processor-specific library.

To avoid listing all the similar prototypes of function flavors of this type, only templates are given here. In the template the length of DFT is denoted by the modifier *<len>* and can be varied in the range of integers [2, 64].

The normalization method of the result is specified by the parameter *flag*.

The forward DFT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi \frac{kn}{N}\right),$$

where *k* is the index of elements in the frequency domain, *n* is the index of elements in the time domain, *N* is the input signal *len*, and *A* is a multiplier defined by *flag*. Also, *x(n)* is *pSrc[n]* and *X(k)* is *pDst[k]*.

**ippsDFTFwd\_RTtoPack, ippgDFTFwd\_RTtoPack.** These functions compute the forward DFT and stores the result in `Pack` format.

**ippsDFTFwd\_RTtoPerm, ippgDFTFwd\_RTtoPerm.** These functions compute the forward DFT and stores the result in `Perm` format.

**ippsDFTFwd\_RTtoCCS, ippgDFTFwd\_RTtoCCS.** These functions compute the forward DFT and stores the result in `CCS` format.

Example 7-6 below shows how to initialize the specification and call `ippsDFTFwd_RTtoCCS_32f`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <i>pBuffer</i> is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.

## Example 7-6 Using the ippsDFTFwd\_RToCCS Function

```
IppStatus dft( void ) {
    Ipp32f x[7], X[8];
    int n;
    IppStatus status;
    IppsDFTSpec_R_32f* spec;
    status = ippsDFTInitAlloc_R_32f(&spec, 7, IPP_FFT_DIV_INV_BY_N,
        ippAlgHintNone);
    for( n=0; n<7; ++n ) x[n] = (float) cos(IPP_2PI * n * 14 / 49);
    status = ippsDFTFwd_RToCCS_32f( x, X, spec, NULL );
    ippsMagnitude_32fc( (Ipp32fc*)X, x, 4 );
    ippsDFTFree_R_32f( spec );
    printf_32f("dft magn =", x, 4, status );
    return status;
}

Output:
dft magn = 0.000000 0.000000 3.500000 0.000000

Matlab* analog:
>> N=7;F=14/49;n=0:N-1;x=cos(2*pi*n*F);y=abs(fft(x));y(1:4)
```

## DFTInv\_PackToR, DFTInv\_PermToR, DFTInv\_CCSToR

*Computes the inverse discrete Fourier transform of a real signal.*

---

### Syntax

#### Case 1: Input data in Pack format

```
IppStatus ippsDFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
    IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
    IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```



```
IppStatus ippsDFTInv_PackToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

### Case 2: Input data in `pack` format, fixed length of DFT

```
IppStatus ippgDFTInv_PackToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTInv_PackToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTInv_PackToR_<len>_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
flag);
```

```
IppStatus ippgDFTInv_PackToR_<len>_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
flag);
```

supported value for `<len>`: integer in the range [2, 64].

### Case 3: Input data in `perm` format

```
IppStatus ippsDFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_PermToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

### Case 4: Result in `perm` format, fixed length of DFT

```
IppStatus ippgDFTInv_PermToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTInv_PermToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTInv_PermToR_<len>_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
flag);
```

```
IppStatus ippgDFTInv_PermToR_<len>_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
flag);
```

supported value for `<len>`: integer in the range [2, 64].

### Case 5: Input data in `css` format

```
IppStatus ippsDFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDFTSpec_R_32f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDFTSpec_R_64f* pDFTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDFTInv_CCSToR_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDFTSpec_R_16s* pDFTSpec, int scaleFactor, Ipp8u* pBuffer);
```

## Case 6: Result in CCS format with fixed length of DFT

```
IppStatus ippgDFTInv_CCSToR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTInv_CCSToR_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len,
int flag);
```

```
IppStatus ippgDFTInv_CCSToR_<len>_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
flag);
```

```
IppStatus ippgDFTInv_CCSToR_<len>_64f(const Ipp64f* pSrc, Ipp64f* pDst, int
flag);
```

supported value for *<len>*: integer in the range [2, 64].

## Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSrc</i>	Pointer to the input array containing packed complex values.
<i>pDst</i>	Pointer to the output array containing real values.
<i>pBuffer</i>	Pointer to the work buffer, can be <code>NULL</code> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>len</i>	Length of the DFT transform in range [2, 64].

## Description

The `ippsDFTInv` and `ippgDFTInv` function is declared in the `ipps.h` file. These functions compute the inverse DFT of a real signal. The input data (that is in the frequency-domain) are represented in several possible packed formats: [Pack](#), [Perm](#), or [CCS](#). [Tables 7-4 and 7-5](#) show how the input data can be represented in the packed formats.

**Case 1, Case3, Case 5.** These functions compute the inverse DFT according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function can be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

Required buffer size must be computed by the corresponding function `ippsDFTGetBufSize_R` prior to using DFT computation functions.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

**Case 2, Case 4, Case 6.** These functions compute the inverse DFT of the fixed length. They do not need the DFT specification structure. The length of transform can be specified by the parameter *len*, or by choosing the function flavor designed for the specific length of transform and containing its value in the function name, for example, the function flavor `ippgDFTInv_CC-SToR_63_32f` performs the inverse DFT with length **63**. Intel IPP support functions for the inverse DFT with the fixed length in the range [2, 64].



**CAUTION.** Data vectors for these functions must be aligned to 16 bytes for the w7 (and above) processor-specific library.

To avoid listing all the similar prototypes of function flavors of this type, only templates are given here. In the template the length of DFT is denoted by the modifier *<len>* and can be varied in the range of integers [2, 64].

The normalization method of the result is specified by the parameter *flag*.

The inverse DFT functionality can be described as follows:

$$x(n) = B \sum_{k=0}^{N-1} X(k) \cdot \exp\left(j2\pi \frac{kn}{N}\right),$$

where  $k$  is the index of elements in the frequency domain,  $n$  is the index of elements in the time domain,  $N$  is the input signal *len*, and  $B$  is a multiplier defined by *flag*. Also,  $x(n)$  is *pDst[n]* and  $X(k)$  is *pSrc[k]*.

**ippsDFTInv\_PackToR.** This function computes the inverse DFT for input data in *Pack* format.

**ippsDFTInv\_PermToR.** This function computes the inverse DFT for input data in *Perm* format.

**ippsDFTInv\_CCSToR.** This function computes the inverse DFT for input data in *CCS* format.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <i>pBuffer</i> is <i>NULL</i> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.

## DFTOutOrdInitAlloc\_C

*Initializes the out-of-order discrete Fourier transform structure.*

---

### Syntax

```

IppStatus ippsDFTOutOrdInitAlloc_C_32fc(IppsDFTOutOrdSpec_C_32fc** ppDFTSpec,
int len, int flag, IppHintAlgorithm hint);

IppStatus ippsDFTOutOrdInitAlloc_C_64fc(IppsDFTOutOrdSpec_C_64fc** ppDFTSpec,
int len, int flag, IppHintAlgorithm hint);

```

### Parameters

<i>ppDFTSpec</i>	Double pointer to the DFT specification structure to be created.
<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .

<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>len</i>	Length of the DFT transform.

## Description

The function `ippsDFTOutOrdInitAlloc_C` is declared in the `ipps.h` file. This function creates and initializes the specification structure `pDFTSpec` for the [out-of-order DFT](#) with the following parameters: the transform *len*, the normalization *flag*, and the specific code *hint*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDFTSpec</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## DFTOutOrdFree\_C

*Closes the out-of-order discrete Fourier transform structure.*

### Syntax

```
IppStatus ippsDFTOutOrdFree_C_32fc(IppsDFTOutOrdSpec_C_32fc* pDFTSpec);
IppStatus ippsDFTOutOrdFree_C_64fc(IppsDFTOutOrdSpec_C_64fc* pDFTSpec);
```

### Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure to be closed.
-----------------	--

### Description

The function `ippsDFTOutOrdFree` is declared in the `ipps.h` file. This function closes the specification structure `pDFTSpec` for the [out-of-order DFT](#) initialized by the function `ippsDFTOutOrdInitAlloc_C`. Call `ippsDFTOutOrdFree` function after the transform is completed.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDFTSpec</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.

## DFTOutOrdGetBufSize\_C

*Computes the size of the work buffer for out-of-order discrete Fourier transform.*

---

### Syntax

```
IppStatus ippSDFTOutOrdGetBufSize_C_32fc(const IppsDFTOutOrdSpec_C_32fc*
pDFTSpec, int* pSize);
```

```
IppStatus ippSDFTOutOrdGetBufSize_C_64fc(const IppsDFTOutOrdSpec_C_64fc*
pDFTSpec, int* pSize);
```

### Parameters

<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pSize</i>	Pointer to the DFT work buffer size value.

### Description

The function `ippSDFTOutOrdGetBufSize` is declared in the `ipps.h` file. This function computes the size in bytes of an external memory buffer for the [out-of-order DFT](#) described by the specification structure *pDFTSpec* and stores it in *pSize*.

To use external buffering, call this function after *pDFTSpec* initialization.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDFTSpec</i> is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.

## DFTOutOrdFwd\_CToC

*Computes the forward out-of-order discrete Fourier transform.*

---

### Syntax

```
IppStatus ippsDFTOutOrdFwd_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
const IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippsDFTOutOrdFwd_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
const IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);
```

### Parameters

<i>pSrc</i>	Pointer to the source data.
<i>pDst</i>	Pointer to the output data.
<i>pDFTSpec</i>	Pointer to the DFT specification structure.
<i>pBuffer</i>	Pointer to the work buffer, can be NULL.

### Description

The function `ippsDFTOutOrdFwd_CToC` is declared in the `ipps.h` file. This function computes the forward [out-of-order DFT](#) of a complex signal *pSrc* according to the *pDFTSpec* specification parameters: the transform *len*, the normalization *flag*, and the specific code *hint*, and store the result in *pDst*.

This function is analogous to `ippsDFTFwd_CToC` and has the similar functionality. The difference is that the elements in frequency domain *pDst* can be rearranged if this helps to speed up computations. This procedure is hidden from the user and depends on the specific implementation of the functions. However, the reversibility of each pair of functions for forward/inverse transforms is ensured.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

Required buffer size must be computed by the function `ippsDFTOutOrdGetBufSize_C` beforehand.

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <code>pBuffer</code> is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <code>pDFTSpec</code> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## DFTOutOrdInv\_CToC

*Computes the inverse out-of-order discrete Fourier transform.*

---

### Syntax

```

IppStatus ippSDFTOutOrdInv_CToC_32fc(const Ipp32fc* pSrc, Ipp32fc* pDst,
const IppsDFTOutOrdSpec_C_32fc* pDFTSpec, Ipp8u* pBuffer);

IppStatus ippSDFTOutOrdInv_CToC_64fc(const Ipp64fc* pSrc, Ipp64fc* pDst,
const IppsDFTOutOrdSpec_C_64fc* pDFTSpec, Ipp8u* pBuffer);

```

### Parameters

<code>pSrc</code>	Pointer to the source data.
<code>pDst</code>	Pointer to the output data.
<code>pDFTSpec</code>	Pointer to the DFT specification structure.
<code>pBuffer</code>	Pointer to the work buffer, can be NULL.

### Description

The function `ippSDFTOutOrdInv_CToC` is declared in the `ipp.h` file. This function computes the inverse [out-of-order DFT](#) of a complex signal `pSrc` according to the `pDFTSpec` specification parameters: the transform `len`, the normalization `flag`, and the specific code `hint`, and store the result in `pDst`.

This function is analogous to `ippSDFTInv_CToC` and has the similar functionality. The difference is that the elements in frequency domain `pSrc` can be rearranged if this helps to speed up computations. This procedure is hidden from the user and depends on the specific implementation of the functions. However, the reversibility of each pair of functions for forward/inverse transforms is ensured.



The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DFT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

Required buffer size must be computed by the function `ippuDFTOutOrdGetBufSize_C` prior to using the DFT computation functions.

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <i>pBuffer</i> is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDFTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## DFT for a Given Frequency (Goertzel) Functions

The functions described in this section compute a single or a number of the discrete Fourier transforms for a given frequency. Note that the DFT exists only for the following normalized frequencies:  $0, 1/N, 2/N, \dots, (N-1)/N$ , where  $N$  is the number of time domain samples. Therefore you must select the frequency value from the above set.

These Intel IPP functions use a Goertzel algorithm [Mit98] and are more efficient when a small number of DFT values is needed.

Some of the functions compute two values, not one. The applications computing several values, for example the dual-tone multi frequency signal detection, work faster, especially on Intel® processors with SIMD instructions.

## Goertz

*Computes the discrete Fourier transform for a given frequency for a single complex signal.*

---

### Syntax

```
ippStatus ippGoertz_32f(const Ipp32f* pSrc, int len, Ipp32fc* pVal, Ipp32f
rFreq);
```

```

IppStatus ippsGoertz_64f(const Ipp64f* pSrc, int len, Ipp64fc* pVal, Ipp64f
rFreq);

IppStatus ippsGoertz_32fc(const Ipp32fc* pSrc, int len, Ipp32fc* pVal, Ipp32f
rFreq);

IppStatus ippsGoertz_64fc(const Ipp64fc* pSrc, int len, Ipp64fc* pVal, Ipp64f
rFreq);

IppStatus ippsGoertz_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16sc* pVal,
Ipp32f rFreq, int scaleFactor);

IppStatus ippsGoertz_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pVal,
Ipp32f rFreq, int scaleFactor);

IppStatus ippsGoertzQ15_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc* pVal,
Ipp16s rFreqQ15, int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the input complex data vector.
<i>len</i>	Number of elements in the vector.
<i>pVal</i>	Pointer to the output DFT value.
<i>rFreq</i>	Single relative frequency value [0, 1.0).
<i>rFreqQ15</i>	Single relative frequency value in Q15 format [0, 32767].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsGoertz` is declared in the `ipps.h` file. This function computes a DFT for a complex input *len*-length signal *pSrc* for a given frequency *rFreq*, and stores the result in *pVal*.

`ippsGoertzQ15`. This function operates with relative frequency in Q15 format. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 1.0).

The functionality of the Goertzel algorithm can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi\frac{kn}{N}\right),$$

where  $k/N$  is the normalized *rFreq* value for which the DFT is computed.

Example 7-7 below illustrates the use of Goertzel functions for selecting the magnitudes of a given frequency when computing DFTs.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRelFreqErr</code>	Indicates an error when <i>rFreq</i> is out of range.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

### Example 7-7. Using Goertzel Functions for Selecting Magnitudes of a Given Frequency

```

IppStatus
goertzel( void ) {

#undef LEN

#define LEN 100

    IppStatus status;

    Ipp32fc *x = ippsMalloc_32fc( LEN ), y;

    int n;

    ///generate a signal of 60 Hz freq that
    /// is sampled with 400 Hz freq
    for( n=0; n<LEN; ++n) {
        x[n].re =(Ipp32f)sin(IPP_2PI * n * 60 / 400);
        x[n].im = 0;
    }

    status = ippsGoertz_32fc( x, LEN, &y, 60.0f / 400 );
    printf_32fc("goertz =", &y, 1, status );
    ippsFree( x );
    return status;
}

Output:
goertz = {0.000090,-50.000008}

Matlab* Analog

>> N=100;F=60/400;n=0:N-1;x=sin(2*pi*n*F);y=fft(x);n=N*F;y(n+1)

```

## GoertzTwo

*Computes two discrete Fourier transforms for a given frequency for a single complex signal.*

---

### Syntax

```
IppStatus ippsGoertzTwo_32f(const Ipp32f* pSrc, int len, Ipp32fc val[2],
```

```
const Ipp32f rFreq[2]);

IppStatus ippsGoertztwo_64f(const Ipp64f* pSrc, int len, Ipp64fc val[2],
const Ipp64f rFreq[2]);

IppStatus ippsGoertztwo_32fc(const Ipp32fc* pSrc, int len, Ipp32fc val[2],
const Ipp32f rFreq[2]);

IppStatus ippsGoertztwo_64fc(const Ipp64fc* pSrc, int len, Ipp64fc val[2],
const Ipp64f rFreq[2]);

IppStatus ippsGoertztwo_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc val[2],
const Ipp32f rFreq[2], int scaleFactor);

IppStatus ippsGoertztwoQ15_16sc_Sfs(const Ipp16sc* pSrc, int len, Ipp16sc
val[2], const Ipp16s rFreqQ15[2], int scaleFactor);
```

## Parameters

<i>pSrc</i>	Pointer to the input complex data vector.
<i>len</i>	Number of elements in the vector.
<i>val</i>	Array of the output DFT values.
<i>rFreq</i>	Array of two relative frequency values [0, 1.0).
<i>rFreqQ15</i>	Array of two relative frequency values in Q15 format [0, 32767].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsGoertztwo` is declared in the `ipps.h` file. This function computes two DFTs for a complex input `len`-length signal `pSrc` for two given frequencies `rFreq`, and stores the result in the output array `val`. The computation of two DFTs on an Intel® Pentium® III processor is performed at the same speed as one. Therefore, the applications computing several DFTs are faster.

The functionality of the Goertzel algorithm can be described as follows:

$$y(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp\left(-j2\pi \frac{kn}{N}\right),$$

where  $k/N$  is one of the normalized *rFreq* values for which the DFTs are computed.

*ippGoertzTwoQ15*. This function operates with relative frequencies in Q15 format. Data in Q15 format are converted to the corresponding float data type that lay in the range [0, 1.0).

### Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when one of the pointers is <code>NULL</code> .
<i>ippStsRelFreqErr</i>	Indicates an error when <i>rFreq</i> is out of range.
<i>ippStsSizeErr</i>	Indicates an error when <i>len</i> is less or equal to 0.

## Hartley Transform Functions

This section describes the functions that perform a discrete Hartley transform (DHT) [Mit93].

DHT is a Fourier-related transform of discrete, periodic data similar to the discrete Fourier transform (DFT). DHT is a real-valued transform - it transforms real inputs to real outputs.

The DHT is its own inverse, or involutory transform.

### Hartley

*Performs Hartley transform of a real signal.*

---

#### Syntax

```
IppStatus ippgHartley_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, int flag);
```

```
IppStatus ippgHartley_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len, int flag);
```

```
IppStatus ippgHartley_<len>_32f(const Ipp32f* pSrc, Ipp32f* pDst, int flag);
```

```
IppStatus ippgHartley_<len>_32f(const Ipp64f* pSrc, Ipp64f* pDst, int flag);
```

supported value for *<len>*: integer in the range [2, 64].

#### Parameters

<i>pSrc</i>	Pointer to the input array.
<i>pDst</i>	Pointer to the output array.

<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>len</i>	Length of the DFT transform in range [2, 64].

## Description

The function `ippgHartley` is declared in the `ipps.h` file.

The function computes the discrete Hartley transform (DHT) of the fixed length of a real signal. The length of transform can be specified by the parameter *len*, or by choosing the function flavor designed for the specific length of transform and containing its value in the function name, for example, the function flavor `ippgHartley_11_32f` performs the DHT with length 11. Intel IPP support functions for DHT with fixed length in the range [2, 64].



**CAUTION.** Data vectors for these functions must be aligned to 16 bytes for the w7 (and above) processor-specific library.

To avoid listing all the similar prototypes of function flavors of this type, only templates are given here. In the template the length of DHT is denoted by the modifier `<len>` and can be varied in the range of integers [2, 64].

The normalization method of the result is specified by the parameter *flag*.

The DHT functionality can be described as follows:

$$X(k) = A \sum_{n=0}^{N-1} x(n) \cdot \left[ \cos\left(2\pi \frac{kn}{N}\right) + \sin\left(2\pi \frac{kn}{N}\right) \right],$$

where  $k$  is the index of elements in the frequency domain,  $n$  is the index of elements in the time domain,  $N$  is the input signal *len*, and  $A$  is a multiplier defined by *flag*. Also,  $x(n)$  is `pSrc[n]` and  $X(k)$  is `pDst[k]`.

The DHT is an involutory transform and the same function can be used both for forward and inverse transforms.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.

## Walsh-Hadamard Transform Functions

This section describes the functions that perform a discrete Walsh-Hadamard (or Hadamard) transform (WHT) [[Mit93](#)].

DHT is a Fourier-related transform. It performs an orthogonal, symmetric, linear operation on real numbers. The computation involves no multiplications.

The WHT is its own inverse, or involutory transform.

## WHT

*Performs Walsh-Hadamard transform of a real signal.*

---

### Syntax

```

IppStatus ippgWHT_32f(const Ipp32f* pSrc, Ipp32f* pDst, int order, int flag,
Ipp8u* pBuffer);

IppStatus ippgWHT_64f(const Ipp64f* pSrc, Ipp64f* pDst, int order, int flag,
Ipp8u* pBuffer);

IppStatus ippgWHT_<order>_32f(const Ipp32f* pSrc, Ipp32f* pDst, int flag,
Ipp8u* pBuffer);

IppStatus ippgWHT_<order>_64f(const Ipp64f* pSrc, Ipp64f* pDst, int flag,
Ipp8u* pBuffer);

```

supported value for *<order>*: integer in the range [1, 13].

### Parameters

<i>pSrc</i>	Pointer to the input array.
<i>pDst</i>	Pointer to the output array.



<i>flag</i>	Specifies the result normalization method. The values for the <i>flag</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>order</i>	Specifies the number of the samples $2^{order}$ in the source signal [1, 13].
<i>pBuffer</i>	Pointer to the working buffer.

## Description

The function `ippgWHT` is declared in the `ipps.h` file.

The function computes the discrete Walsh-Hadamard transform (WHT) of a real signal for the fixed number of samples  $2^{order}$ . The number of samples can be specified by the parameter *order*, or by choosing the function flavor designed for the specific length of the signal and containing *order* in the function name, for example, the function flavor `ippgWHT_11_32f` performs the WHT of the  $2^{11}$  samples of the source signal. Intel IPP support WHT functions for fixed number of input samples corresponding to the values of *order* in the range [1, 13].



**CAUTION.** Data vectors for these functions must be aligned to 16 bytes for the *w7* (and above) processor-specific library.

To avoid listing all the similar prototypes of function flavors of this type, only templates are given here. In the template the `<order>` modifier in a function name denotes a correspondent value of *order*.

The normalization method of the result is specified by the parameter *flag*.

This function requires the working buffer, its size must be computed with the function [ippgWHT-GetBufferSize](#) beforehand.

The WHT is an involutory transform and the same function can be used both for forward and inverse transforms.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFftFlagErr</code>	Indicates an error when the <i>flag</i> value is incorrect.

## WHTGetBufferSize

Computes the size of the working buffer for WHT.

### Syntax

```
IppStatus ippgWHTGetBufferSize_32f(int order, Ipp32u* pBufferSize);
IppStatus ippgWHTGetBufferSize_64f(int order, Ipp32u* pBufferSize);
```

### Parameters

<i>order</i>	Specifies the number of the samples $2^{order}$ in the source signal [1, 13].
<i>pBufferSize</i>	Pointer to the value of the size of the working buffer.

### Description

The function `ippgWHTGetBufferSize` is declared in the `ipps.h` file. This function computes in bytes the size *pBufferSize* of the work buffer for the WHT in accordance with the specified number of samples in the source signal.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pBufferSize</i> pointer is <code>NULL</code> .

## Discrete Cosine Transform Functions

This section describes the functions that compute the discrete cosine transform (DCT) of a signal. DCT functions used in the Intel IPP signal processing data-domain implement the modified computation algorithm proposed in [Rao90].

## DCTFwdInitAlloc

*Allocates memory and initializes the forward DCT structure.*

---

### Syntax

```
IppStatus ippsDCTFwdInitAlloc_16s(IppsDCTFwdSpec_16s** ppDCTSpec, int len,
IppHintAlgorithm hint);

IppStatus ippsDCTFwdInitAlloc_32f(IppsDCTFwdSpec_32f** ppDCTSpec, int len,
IppHintAlgorithm hint);

IppStatus ippsDCTFwdInitAlloc_64f(IppsDCTFwdSpec_64f** ppDCTSpec, int len,
IppHintAlgorithm hint);
```

### Parameters

<i>ppDCTSpec</i>	Double pointer to the forward DCT specification structure to be created.
<i>len</i>	Number of samples in the DCT.
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .

### Description

The function `ippsDCTFwdInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the forward DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

The forward DCT specification structure is used by the function `ippsDCTFwd`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>ppDCTSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.

## DCTInvInitAlloc

*Allocates memory and initializes the inverse DCT structure.*

---

### Syntax

```
IppStatus ippsDCTInvInitAlloc_16s(IppsDCTInvSpec_16s** ppDCTSpec, int len,
IppHintAlgorithm hint);

IppStatus ippsDCTInvInitAlloc_32f(IppsDCTInvSpec_32f** ppDCTSpec, int len,
IppHintAlgorithm hint);

IppStatus ippsDCTInvInitAlloc_64f(IppsDCTInvSpec_64f** ppDCTSpec, int len,
IppHintAlgorithm hint);
```

### Parameters

<i>ppDCTSpec</i>	Double pointer to the inverse DCT specification structure to be created.
<i>len</i>	Number of samples in the DCT.
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .

### Description

The function `ippsDCTInvInitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the inverse DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

The inverse DCT specification structure is used by the function [ippsDCTInv](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>ppDCTSpec</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.

## DCTFwdFree

*Closes the forward DCT structure.*

---

### Syntax

```
IppStatus ippSDCTFwdFree_16s(IppsDCTFwdSpec_16s* pDCTSpec);
IppStatus ippSDCTFwdFree_32f(IppsDCTFwdSpec_32f* pDCTSpec);
IppStatus ippSDCTFwdFree_64f(IppsDCTFwdSpec_64f* pDCTSpec);
```

### Parameters

*pDCTSpec*                      Pointer to the forward DCT specification structure.

### Description

The function `ippSDCTFwdFree` is declared in the `ipps.h` file. This function closes the forward DCT specification structure *pDCTSpec* initialized by the function `ippSDCTFwdInitAlloc`. Call the function `ippSDCTFwdFree` after the transform is completed.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error if the *pDCTSpec* pointer is NULL.  
`ippStsContextMatchErr`        Indicates an error if the specification identifier *pDCTSpec* is incorrect.

## DCTInvFree

*Closes the inverse discrete cosine transform structure.*

---

### Syntax

```
IppStatus ippSDCTInvFree_16s(IppsDCTInvSpec_16s* pDCTSpec);
IppStatus ippSDCTInvFree_32f(IppsDCTInvSpec_32f* pDCTSpec);
IppStatus ippSDCTInvFree_64f(IppsDCTInvSpec_64f* pDCTSpec);
```

## Parameters

*pDCTSpec* Pointer to the inverse DCT specification structure.

## Description

The function `ippsDCTInvFree` is declared in the `ipps.h` file. This function closes the inverse DCT specification structure *pDCTSpec* initialized by the function `ippsDCTInvInitAlloc`. Call the function `ippsDCTInvFree` after the transform is completed.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the *pDCTSpec* pointer is NULL.  
`ippStsContextMatchErr` Indicates an error when the specification identifier *pDCTSpec* is incorrect.

## DCTFwdGetBufSize

Computes the size of the forward DCT work buffer.

### Syntax

```
IppStatus ippsDCTFwdGetBufSize_32f(const IppsDCTFwdSpec_32f* pDCTSpec, int* pBufferSize);
```

```
IppStatus ippsDCTFwdGetBufSize_16s(const IppsDCTFwdSpec_16s* pDCTSpec, int* pBufferSize);
```

```
IppStatus ippsDCTFwdGetBufSize_64f(const IppsDCTFwdSpec_64f* pDCTSpec, int* pBufferSize);
```

## Parameters

*pDCTSpec* Pointer to the forward DCT specification structure.  
*pBufferSize* Pointer to the forward DCT work buffer size value.

## Description

The function `ippsDCTFwdGetBufSize` is declared in the `ipps.h` file. This function computes in bytes the size *pBufferSize* of the work buffer for the forward DCT specified by the structure *pDCTSpec*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDCTSpec</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDCTSpec</i> is incorrect.

## DCTInvGetBufSize

Computes the size of the inverse DCT work buffer.

### Syntax

```
IppStatus ippDCTInvGetBufSize_16s(const IppsDCTInvSpec_16s* pDCTSpec, int* pBufferSize);  
  
IppStatus ippDCTInvGetBufSize_32f(const IppsDCTInvSpec_32f* pDCTSpec, int* pBufferSize);  
  
IppStatus ippDCTInvGetBufSize_64f(const IppsDCTInvSpec_64f* pDCTSpec, int* pBufferSize);
```

### Parameters

<i>pDCTSpec</i>	Pointer to the DCT specification structure.
<i>pBufferSize</i>	Pointer to the DCT work buffer size value.

### Description

The function `ippDCTInvGetBufSize` is declared in the `ipps.h` file. This function computes in bytes the size *pBufferSize* of the work buffer for the inverse DCT specified by the structure *pDCTSpec*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDCTSpec</i> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDCTSpec</i> is incorrect.

## DCTFwdInit

*Initializes the forward discrete cosine transform structure.*

---

### Syntax

```

IppStatus ippsDCTFwdInit_16s(IppsDCTFwdSpec_16s** ppDCTSpec, int len,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsDCTFwdInit_32f(IppsDCTFwdSpec_32f** ppDCTSpec, int len,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippsDCTFwdInit_64f(IppsDCTFwdSpec_64f** ppDCTSpec, int len,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

```

### Parameters

<i>ppDCTSpec</i>	Double pointer to the forward DCT specification structure to be created.
<i>len</i>	Number of samples in the DCT.
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>pSpec</i>	Pointer to the area for the DCT specification structure.
<i>pSpecBuffer</i>	Pointer to the additional work buffer, can be NULL.

### Description

The function `ippsDCTFwdInit` is declared in the `ipps.h` file. This function initializes the forward DCT specification structure *ppDCTSpec* with the following parameters: the transform *len*, and the specific code *hint*.

Before calling this function the memory must be allocated for the DCT specification structure and the work buffer (if it is required). The size of the DCT specification structure and the work buffer must be computed by the function [ippsDCTFwdGetBufSize](#) beforehand.

If the work buffer is not used, the parameter *pSpecBuffer* can be NULL. If the working buffer is used, the parameter *pSpecBuffer* must not be NULL.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------



<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers with exception of <code>pSpecBuffer</code> is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## DCTInvInit

*Initializes the inverse discrete cosine transform structure.*

---

### Syntax

```

IppStatus ippSDCTInvInit_16s(IppsDCTInvSpec_16s** ppDCTSpec, int len,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippSDCTInvInit_32f(IppsDCTInvSpec_32f** ppDCTSpec, int len,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

IppStatus ippSDCTInvInit_64f(IppsDCTInvSpec_64f** ppDCTSpec, int len,
IppHintAlgorithm hint, Ipp8u* pSpec, Ipp8u* pSpecBuffer);

```

### Parameters

<code>ppDCTSpec</code>	Double pointer to the inverse DCT specification structure to be created.
<code>len</code>	Number of samples in the DCT.
<code>hint</code>	Suggests using specific code for calculation. The values for the <code>hint</code> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<code>pSpec</code>	Pointer to the area for the DCT specification structure.
<code>pSpecBuffer</code>	Pointer to the work buffer, can be <code>NULL</code> .

### Description

The function `ippSDCTInvInit` is declared in the `ipps.h` file. This function initializes in the buffer `pSpec` the inverse DCT specification structure `ppDCTSpec` with the following parameters: the transform `len`, and the specific code `hint`.

Before calling this function the memory must be allocated for the DCT specification structure and the work buffer (if it is required). The size of the DFT specification structure and the work buffer must be computed by the function [ippsDCTInvGetSize](#).

If the work buffer is not used, the parameter *pSpecBuffer* can be NULL. If the working buffer is used, the parameter *pSpecBuffer* must not be NULL.

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when one of the specified pointers with exception of <i>pSpecBuffer</i> is NULL.
<i>ippStsSizeErr</i>	Indicates an error when <i>len</i> is less than or equal to 0.

## DCTFwdGetSize

*Computes the size of all buffers required for the forward DCT.*

---

### Syntax

```

IppStatus ippDCTFwdGetSize_16s(int len, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);

IppStatus ippDCTFwdGetSize_32f(int len, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);

IppStatus ippDCTFwdGetSize_64f(int len, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);

```

### Parameters

<i>len</i>	Number of samples in the DCT.
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<i>pSpecSize</i>	Pointer to the size of the forward DCT specification structure.
<i>pSpecBufferSize</i>	Pointer to the size of the work buffer for the initialization function.
<i>pBufferSize</i>	Pointer to the size of the forward DCT work buffer.

## Description

The function `ippsDCTFwdGetSize` is declared in the `ipps.h` file. This function computes the size `pSpecSize` for the forward DCT structure with the following parameters: the transform `len`, and the specific code `hint`. Additionally the function computes the size `pSpecBufferSize` of the work buffer for the initialization function `ippsDCTFwdInit` , and the size `pBufferSize` of the work buffer for the function `ippsDCTFwd` .

The function `ippsDCTFwdGetSize` should be called prior to them.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## DCTInvGetSize

*Computes the size of all buffers required for the inverse DCT.*

---

### Syntax

```
IppStatus ippsDCTInvGetSize_16s(int len, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);

IppStatus ippsDCTInvGetSize_32f(int len, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);

IppStatus ippsDCTInvGetSize_64f(int len, IppHintAlgorithm hint, int*
pSpecSize, int* pSpecBufferSize, int* pBufferSize);
```

### Parameters

<code>len</code>	Number of samples in the DCT.
<code>hint</code>	Suggests using specific code for calculation. The values for the <code>hint</code> argument are described in the section <a href="#">Flag and Hint Arguments</a> .
<code>pSpecSize</code>	Pointer to the size of the forward DCT specification structure.
<code>pSpecBufferSize</code>	Pointer to the size of the work buffer for the initialization function.

*pBufferSize* Pointer to the size of the forward DCT work buffer.

## Description

The function `ippsDCTInvGetSize` is declared in the `ipps.h` file. This function computes in bytes the size *pSpecSize* of the external buffer for the inverse DCT structure with the following parameters: the transform *len*, and the specific code *hint*. Additionally the function computes the size *pSpecBufferSize* of the work buffer for the initialization function `ippsDCTInvInit` and the size *pBufferSize* of the work buffer for the function `ippsDCTInv`.

The function `ippsDCTInvGetSize` must be called prior to them.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## DCTFwd

*Computes the forward discrete cosine transform of a signal.*

---

### Syntax

#### Case 1: Not-in-place operation

```
ippStatus ippsDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDCTFwdSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
ippStatus ippsDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDCTFwdSpec_32f* pDCTSpec, Ipp8u* pBuffer);
```

```
ippStatus ippsDCTFwd_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDCTFwdSpec_64f* pDCTSpec, Ipp8u* pBuffer);
```

#### Case 2: In-place operation

```
ippStatus ippsDCTFwd_16s_ISfs(Ipp16s* pSrcDst, const IppsDCTFwdSpec_16s*
pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
ippStatus ippsDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsDCTFwdSpec_32f*
pDCTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDCTFwd_64f_I(Ipp64f* pSrcDst, const IppsDCTFwdSpec_64f*
pDCTSpec, Ipp8u* pBuffer);
```

## Parameters

<i>pDCTSpec</i>	Pointer to the forward DCT specification structure.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operations.
<i>pBuffer</i>	Pointer to the external work buffer, can be NULL
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsDCTFwd` is declared in the `ipps.h` file. This function computes the forward discrete cosine transform (DCT) of the source signal *pSrc* (*pSrcDst* for in-place operations) in accordance with the specification structure *pDCTSpec* that must be initialized by calling the functions `ippsDCTFwdInitAlloc` or `ippsDCTFwdInit` beforehand. The result is stored in the *pDst* (*pSrcDst* for in-place operations).

If *len* is a power of 2, the function uses an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *len*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT,  $N = len$ ,

$$c(k) = \frac{1}{\sqrt{N}} \text{ for } k = 0, \quad c(k) = \frac{\sqrt{2}}{\sqrt{N}} \text{ for } k > 0;$$

$x(n)$  is *pSrc*[*n*] and  $y(k)$  is *pDst*[*k*].

The forward DCT is defined by the formula:

$$y(k) = C(k) \sum_{n=0}^{N-1} x(n) \cdot \cos \frac{(2n+1)\pi k}{2N}$$

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DCT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of this buffer must be computed previously using the functions [ippsDCTFwdGetBufSize](#) or [ippsDCTFwdGetSize](#).

If the external buffer is not specified (*pBuffer* is set to NULL), then the function itself allocates the memory needed for operation.

[Example 7-8](#) shows how to use the DCT functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the <i>pDCTSpec</i> , <i>pSrc</i> , <i>pDst</i> , <i>pSrcDst</i> pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error if the specification identifier <i>pDCTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.

## DCTInv

*Computes the inverse discrete cosine transform of a signal.*

---

### Syntax

#### Case 1: Not-in-place operation

```

IppStatus ippsDCTInv_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDCTInvSpec_16s* pDCTSpec, int scaleFactor, Ipp8u* pBuffer);

IppStatus ippsDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDCTInvSpec_32f* pDCTSpec, Ipp8u* pBuffer);

IppStatus ippsDCTInv_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDCTInvSpec_64f* pDCTSpec, Ipp8u* pBuffer);

```

## Case 2: In-place operation

```
IppStatus ippsDCTInv_16s_ISfs(Ipp16s* pSrcDst, const IppsDCTInvSpec_16s*
pDCTSpec, int scaleFactor, Ipp8u* pBuffer);
```

```
IppStatus ippsDCTInv_32f_I(Ipp32f* pSrcDst, const IppsDCTInvSpec_32f*
pDCTSpec, Ipp8u* pBuffer);
```

```
IppStatus ippsDCTInv_64f_I(Ipp64f* pSrcDst, const IppsDCTInvSpec_64f*
pDCTSpec, Ipp8u* pBuffer);
```

## Parameters

<i>pDCTSpec</i>	Pointer to the inverse DCT specification structure.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for in-place operations.
<i>pBuffer</i>	Pointer to the external work buffer, may be <code>NULL</code> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsDCTInv` is declared in the `ipps.h` file. This function computes the inverse discrete cosine transform (DCT) of the source signal *pSrc* (*pSrcDst* for in-place operations) in accordance with the specification structure *pDCTSpec* that must be initialized by calling the functions [ippsDCTInvInitAlloc](#) or [ippsDCTInvInit](#) beforehand. The result is stored in the *pDst* (*pSrcDst* for in-place operations).

If *len* is a power of 2, the functions use an efficient algorithm that is significantly faster than the direct computation of DCT. For other values of *len*, these functions use the direct formulas given below; however, the symmetry of the cosine function is taken into account, which allows to perform about half of the multiplication operations in the formulas.

In the following definition of DCT,  $N = len$ ,

$$c(k) = \frac{1}{\sqrt{N}} \text{ for } k = 0, \quad c(k) = \frac{\sqrt{2}}{\sqrt{N}} \text{ for } k > 0;$$

$x(n)$  is *pDst*[*n*] and  $y(k)$  is *pSrc*[*k*].

The inverse DCT is defined by the formula:

$$x(n) = \sum_{k=0}^{N-1} C(k)Y(k) \cdot \cos\frac{(2n+1)\pi k}{2N}$$

For integer data types the output result is scaled according to the *scaleFactor* value, thus the output signal range and precision are retained.

The function may be used with the external work buffer *pBuffer* that allows to avoid memory allocation within the functions. Once the work buffer is allocated, it can be used for all following calls to the functions computing DCT. On modern Intel® architectures the use of an external buffer improves performance significantly, especially for the small size transforms.

The size of this buffer must be computed previously using the functions [ippsDCTInvGetBufSize](#) or [ippsDCTInvGetSize](#).

If the external buffer is not specified (*pBuffer* is set to `NULL`), then the function itself allocates the memory needed for operation.

Example 7-8 below shows how to use the functions `ippsDCTFwd_32f` and `ippsDCTInv_32f` to compress and reconstruct a signal.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the <i>pDCTSpec</i> , <i>pSrc</i> , <i>pDst</i> , <i>pSrcDst</i> pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pDCTSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.



---

**Example 7-8 Using ippsDCTFwd and ippsDCTInv to Compress and Reconstruct a Signal**

```
void dct( void ) {  
    #define LEN 256  
  
    Ipp32f x[LEN], y[LEN];  
    int n;  
    IppsDCTFwdSpec_32f* fspec;  
    IppsDCTInvSpec_32f* ispec;  
    IppStatus status;  
    /// data: Gaussian function, magn =1 and sigma=N/3  
  
    for(n=0; n<LEN; ++n)
```

```

        x[n] = (float)(exp(-0.5*(LEN/2-n)*(LEN/2-n)/(LEN/3.0)));

    /// get cosine transform coefficients
    status = ippsDCTFwdInitAlloc_32f( &fspec, LEN, ippAlgHintNone );
    status = ippsDCTFwd_32f( x, y, fspec, NULL );
    ippsDCTFwdFree_32f( fspec );

    /// Set 3/4 of these coefficients to zero
    for(n=LEN/4; n<LEN; ++n) y[n] = 0.0f;
    /// restore signal using len/4 values
    status = ippsDCTInvInitAlloc_32f( &ispec, LEN, ippAlgHintNone );
    status = ippsDCTInv_32f( y, x, ispec, NULL );
    ippsDCTInvFree_32f( ispec );
}

```

## DCT4InitAlloc

*Allocates memory and initializes the DCT-IV structure.*

---

### Syntax

```

IppStatus ippgDCT4InitAlloc_32f(IppgDCT4Spec_32f** ppSpec, int len);
IppStatus ippgDCT4InitAlloc_64f(IppgDCT4Spec_64f** ppSpec, int len);

```

### Parameters

<i>ppSpec</i>	Double pointer to the DCT-IV specification structure to be created.
<i>len</i>	Number of samples in the transform.

### Description

The function `ippgDCT4InitAlloc` is declared in the `ipps.h` file. This function allocates memory and initializes the DCT-IV specification structure *ppSpec* with the length of transform *len*.

This specification structure is used by the function `ippgDCT4`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>ppSpec</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>len</code> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.

## DCT4Free

*Closes the DCT-IV structure.*

---

### Syntax

```
IppStatus ippgDCT4Free_32f(IppgDCT4Spec_32f* pSpec);
IppStatus ippgDCT4Free_64f(IppgDCT4Spec_64f* pSpec);
```

### Parameters

`pSpec` Pointer to the DCT-IV specification structure.

### Description

The function `ippgDCT4Free` is declared in the `ipps.h` file. This function closes the DCT-IV specification structure `pSpec` initialized by the function `ippgDCT4InitAlloc`. Call the function `ippgDCT4Free` after the transform is completed.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pSpec</code> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error if the specification identifier <code>pSpec</code> is incorrect.

## DCT4Init

*Initializes the DCT-IV structure.*

---

### Syntax

```
IppStatus ippgDCT4Init_32f(IppgDCT4Spec_32f** ppSpec, int len, Ipp8u* pMem);
IppStatus ippgDCT4Init_64f(IppgDCT4Spec_64f** ppSpec, int len, Ipp8u* pMem);
```

## Parameters

<i>ppSpec</i>	Double pointer to the DCT-IV specification structure to be created.
<i>len</i>	Number of samples in the transform.
<i>pMem</i>	Pointer to the area for the DCT-IV specification structure.

## Description

The function `ippgDCT4Init` is declared in the `ipps.h` file. This function initializes the DCT-IV specification structure *ppSpec* with the length of transform *len*.

Before calling this function the memory must be allocated for the DCT-IV specification structure. The size of the DCT-IV specification structure must be computed by the function [ippgDCT4GetSize](#) beforehand.

This specification structure is used by the function [ippgDCT4](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>ppSpec</i> or <i>pMem</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

## DCT4GetSize

*Computes the size of the DCT-IV specification structure.*

---

### Syntax

```
ippStatus ippgDCT4GetSize_32f(int len, int* pSize);
ippStatus ippgDCT4GetSize_64f(int len, int* pSize);
```

### Parameters

<i>len</i>	Number of samples in the transform.
<i>pSize</i>	Pointer to the size of the DCT-IV specification structure.

## Description

The function `ippgDCT4GetSize` is declared in the `ipps.h` file. This function computes the size `pSize` of the DCT-IV structure with the length of transform `len`.

The function `ippgDCT4GetSize` must be called prior to the function `ippgDCT4Init`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pSize</code> is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## DCT4

*Computes the DCT-IV of a signal.*

---

## Syntax

```
IppStatus ippgDCT4_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppgDCT4Spec_32f* pSpec);
```

```
IppStatus ippgDCT4_64f(const Ipp64f* pSrc, Ipp64f* pDst, const
IppsDCT4Spec_64f* pSpec);
```

## Parameters

<code>pSpec</code>	Pointer to the DCT-IV specification structure.
<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.

## Description

The function `ippgDCT4` is declared in the `ipps.h` file. This function computes the type-IV discrete cosine transform (DCT-IV) of the source signal `pSrc` in accordance with the specification structure `pSpec` that must be initialized by calling the functions `ippgDCT4InitAlloc` or `ippgDCT4Init` beforehand. The result is stored in the `pDst`. The DCT-IV is its own inverse, or involutory transform.

In the following definition of DCT-IV,  $N = len$ ,

$$c(k) = \frac{1}{\sqrt{N}} \text{ for } k = 0, \quad c(k) = \frac{\sqrt{2}}{\sqrt{N}} \text{ for } k > 0;$$

$x(n)$  is  $pSrc[n]$  and  $y(k)$  is  $pDst[k]$ .

The DCT-IV is defined by the formula [Rao90]:

$$y(k) = C(k) \sum_{n=0}^{N-1} x(n) \cdot \cos \frac{(2n+1)(2k+1)\pi}{4N}$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the <i>pSpec</i> , <i>pSrc</i> , <i>pDst</i> pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error if the specification identifier <i>pSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.

## Hilbert Transform Functions

The functions described in this section compute a discrete-time analytic signal from a real data sequence using the Hilbert transform. The analytic signal is a complex signal whose real part is a replica of the original data, and imaginary part contains the Hilbert transform. That is, the imaginary part is a version of the original real data with a 90 degrees phase shift. The Hilbert transformed data have the same amplitude and frequency content as the original real data, plus the additional phase information.

### HilbertInitAlloc

*Initializes the Hilbert transform structure.*

---

#### Syntax

```
IppStatus ippHilbertInitAlloc_32f32fc(IppsHilbertSpec_32f32fc** ppSpec, int
len, IppHintAlgorithm hint);
```

```
IppStatus ippsHilbertInitAlloc_16s32fc(IppsHilbertSpec_16s32fc** ppSpec, int
len, IppHintAlgorithm hint);
```

```
IppStatus ippsHilbertInitAlloc_16s16sc(IppsHilbertSpec_16s16sc** ppSpec, int
len, IppHintAlgorithm hint);
```

## Parameters

<i>ppSpec</i>	Double pointer to the Hilbert context structure being initialized.
<i>len</i>	Number of samples in the Hilbert transform.
<i>hint</i>	Suggests using specific code for calculation. The values for the <i>hint</i> argument are described in the section <a href="#">Flag and Hint Arguments</a> .

## Description

The function `ippsHilbertInitAlloc` is declared in the `ipps.h` file. This function creates and initializes the Hilbert specification structure *pSpec* with the following parameters: the length of the transform *len*, and the specific code indicator *hint*. Call this function before using the Hilbert transform function `ippsHilbert`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSpec</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## HilbertFree

*Closes a Hilbert transform structure.*

---

### Syntax

```
IppStatus ippsHilbertFree_32f32fc(IppsHilbertSpec_32f32fc* pSpec);
```

```
IppStatus ippsHilbertFree_16s32fc(IppsHilbertSpec_16s32fc* pSpec);
```

```
IppStatus ippsHilbertFree_16s16sc(IppsHilbertSpec_16s16sc* pSpec);
```

## Parameters

*pSpec* Pointer to the Hilbert specification structure to be closed.

## Description

The function `ippsHilbertFree` is declared in the `ipps.h` file. This function closes the Hilbert specification structure *pSpec* by freeing all memory associated with the specification created by the function `ippsHilbertInitAlloc`. Call `ippsHilbertFree` after the transform is completed.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the *pSpec* pointer is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the specification identifier *pSpec* is incorrect.

## Hilbert

*Computes an analytic signal using the Hilbert transform.*

---

## Syntax

```
IppStatus ippsHilbert_32f32fc(const Ipp32f* pSrc, Ipp32fc* pDst,
IppsHilbertSpec_32f32fc* pSpec);

IppStatus ippsHilbert_16s32fc(const Ipp16s* pSrc, Ipp32fc* pDst,
IppsHilbertSpec_16s32fc* pSpec);

IppStatus ippsHilbert_16s16sc_Sfs(const Ipp16s* pSrc, Ipp16sc* pDst,
IppsHilbertSpec_16s16sc* pSpec, int scaleFactor);
```

## Parameters

*pSpec* Pointer to the Hilbert specification structure.  
*pSrc* Pointer to the vector containing original real data.  
*pDst* Pointer to the output array containing complex data.  
*scaleFactor* Scale factor, refer to [Integer Scaling](#).



## Description

The function `ippsHilbert` is declared in the `ipps.h` file. This function computes a complex analytic signal *pDst*, which contains the original real signal *pSrc* as its real part and computed Hilbert transform as its imaginary part. The Hilbert transform is performed according to the *pSpec* specification parameters: the number of samples *len*, and the specific code *hint*. The input data is zero-padded or truncated to the size of *len* as appropriate. For integer data types, the output result is scaled according to the *scaleFactor* value, which ensures that the output signal range and precision are retained.

Example 7-9 below shows how to initialize the specification structure and use the function `ippsHilbert_32f32fc`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pSpec</i> is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## Example 7-9. Using Hilbert Functions

```
IppStatus hilbert(void) {
    Ipp32f x[10];
    Ipp32fc y[10];
    int n;
    IppStatus status;
    IppsHilbertSpec_32f32fc* spec;
    status = ippsHilbertInitAlloc_32f32fc(&spec, 10, ippAlgHintNone);
    for(n = 0; n < 10; n++) {
        x[n] = (Ipp32f)cos(IPP_2PI * n * 2 / 9);
    }
    status = ippsHilbert_32f32fc(x, y, spec);
    ippsMagnitude_32fc((Ipp32fc*)y, x, 5);
    ippsHilbertFree_32f32fc(spec);
    printf_32f("hilbert magn =", x, 5, status);
    return status;
}
```

Output:

```
hilbert magn = 1.0944 1.1214 1.0413 0.9707 0.9839
```

Matlab\* Analog:

```
>> n=0:9; x=cos(2*pi*n*2/9); y=abs(hilbert(x)); y(1:5)
```

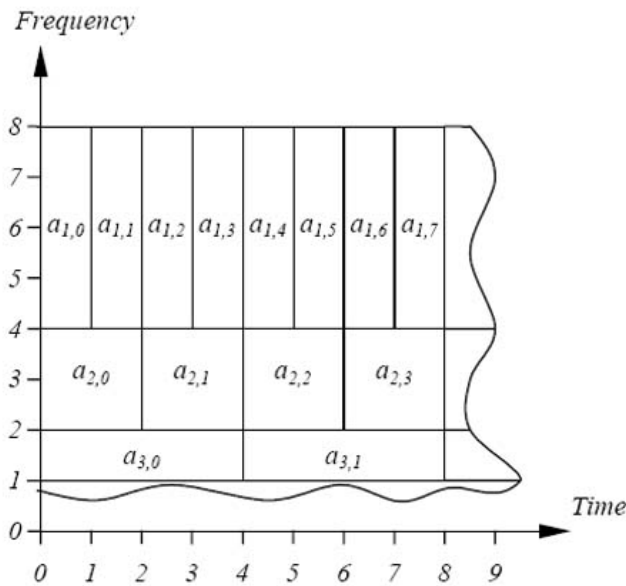
## Wavelet Transform Functions

This section describes the wavelet transform functions implemented in Intel IPP.

In signal processing, signals can be represented in both frequency and time-frequency domains. In many cases the wavelet transforms become an alternative to short time Fourier transforms.

The discrete wavelet signal can be considered as a set of the coefficients  $a_{i,k}$  with two indices, one of which is a “frequency” characteristic and the other is a time localization. The coefficient value corresponds to the localized wave amplitude or to one of basis transform functions. The “frequency” index shows the time scale of the localized wave. Function bases originated from one local wave by decreasing the wave by  $2^n$  in time are the most widely used. Such transforms can be used for building very efficient implementations called fast wavelet transforms by analogy with fast Fourier transforms. Figure 7-1 shows how the time and frequency plane is divided into areas that correspond to the local wave amplitudes. This kind of transforms is implemented in Intel IPP and referred to as the discrete wavelet transform (DWT).

**Figure 149: Figure 7-1 Wavelet Decomposition Coefficients in Time-Frequency Domain**



The DWT is one of the wavelet analysis methods that stem from the basis functions related to the scale factor 2. Thus, there is a basic common element shared by the DWT and the other packet analysis methods.

Likewise another basic element for signal reconstruction or synthesis can be defined, called the one-level inverse DWT. Figure 7-2 shows the diagram of the forward DWT which allows to switch to time-frequency representation shown in Figure above. The diagram includes three levels of decomposition. Figure 7-3 shows the corresponding procedure of signal reconstruction based on the elementary one-level inverse transform.

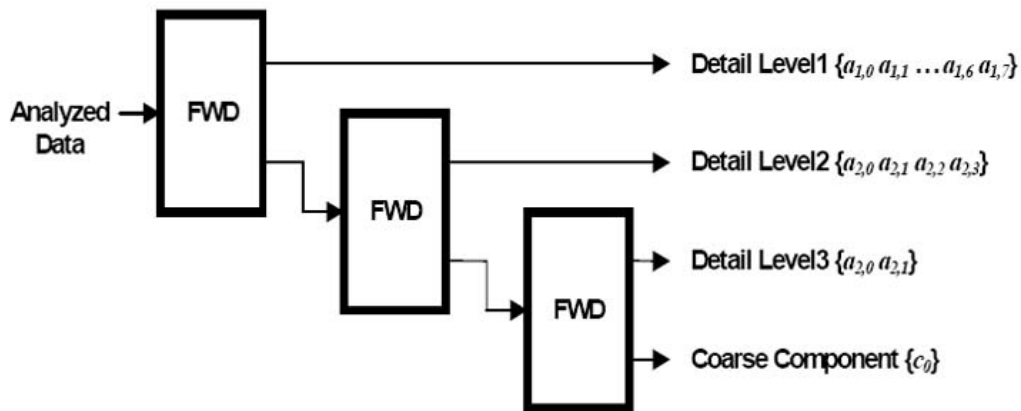
The implementation of discrete multi-scale transforms is based on the use of interpolation and decimation filters with the resampling factor 2. The basis of the multi-scale signal decomposition and reconstruction functions uniquely defines the filter parameters. The Intel IPP multi-scale transform functions use filters with finite impulse response.

The Primitives contains two sets of functions.

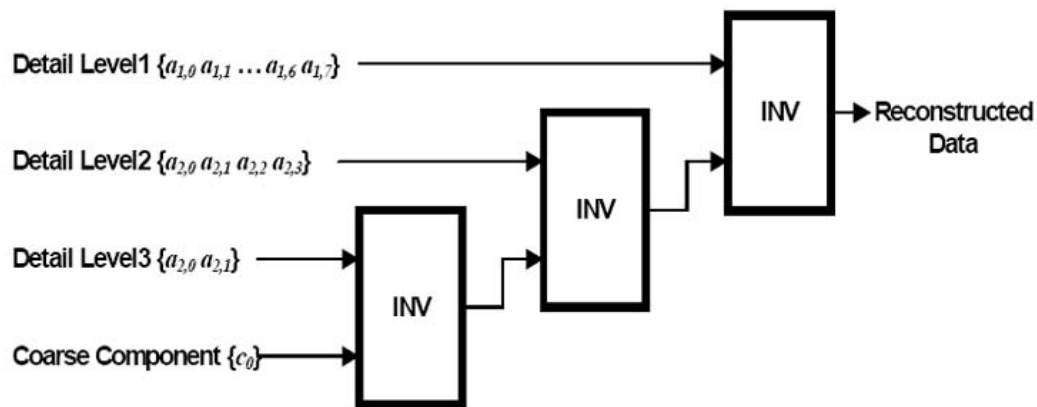
- Transforms designed for fixed filter banks. These transforms yield the highest performance.

- Transforms that enable the user to work with arbitrary filters. These functions use effective polyphase filtration algorithms. The transform interface gives the option of processing the data in blocks, including in real-time applications.

**Figure 7-2 Three-Level Discrete Wavelet Decomposition**



**Figure 7-3 Three-Level Discrete Wavelet Reconstruction**



## Transforms for Fixed Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for fixed filter banks.

### WTHaarFwd, WTHaarInv

*Performs forward or inverse single-level discrete wavelet Haar transforms.*

---

#### Syntax

##### Case 1: Forward transform

```

IppStatus ippsWTHaarFwd_8s(const Ipp8s* pSrc, int lenSrc, Ipp8s* pDstLow,
Ipp8s* pDstHigh);

IppStatus ippsWTHaarFwd_16s(const Ipp16s* pSrc, int lenSrc, Ipp16s* pDstLow,
Ipp16s* pDstHigh);

IppStatus ippsWTHaarFwd_32s(const Ipp32s* pSrc, int lenSrc, Ipp32s* pDstLow,
Ipp32s* pDstHigh);

IppStatus ippsWTHaarFwd_64s(const Ipp64s* pSrc, int lenSrc, Ipp64s* pDstLow,
Ipp64s* pDstHigh);

IppStatus ippsWTHaarFwd_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh);

IppStatus ippsWTHaarFwd_64f(const Ipp64f* pSrc, int lenSrc, Ipp64f* pDstLow,
Ipp64f* pDstHigh);

IppStatus ippsWTHaarFwd_8s_Sfs(const Ipp8s* pSrc, int lenSrc, Ipp8s* pDstLow,
Ipp8s* pDstHigh, int scaleFactor);

IppStatus ippsWTHaarFwd_16s_Sfs(const Ipp16s* pSrc, int lenSrc, Ipp16s*
pDstLow, Ipp16s* pDstHigh, int scaleFactor);

IppStatus ippsWTHaarFwd_32s_Sfs(const Ipp32s* pSrc, int lenSrc, Ipp32s*
pDstLow, Ipp32s* pDstHigh, int scaleFactor);

IppStatus ippsWTHaarFwd_64s_Sfs(const Ipp64s* pSrc, int lenSrc, Ipp64s*
pDstLow, Ipp64s* pDstHigh, int scaleFactor);

```

## Case 2: Inverse transform

```

IppStatus ippsWTHaarInv_8s(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh,
Ipp8s* pDst, int lenDst);

IppStatus ippsWTHaarInv_16s(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh,
Ipp16s* pDst, int lenDst);

IppStatus ippsWTHaarInv_32s(const Ipp32s* pSrcLow, const Ipp32s* pSrcHigh,
Ipp32s* pDst, int lenDst);

IppStatus ippsWTHaarInv_64s(const Ipp64s* pSrcLow, const Ipp64s* pSrcHigh,
Ipp64s* pDst, int lenDst);

IppStatus ippsWTHaarInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
Ipp32f* pDst, int lenDst);

IppStatus ippsWTHaarInv_64f(const Ipp64f* pSrcLow, const Ipp64f* pSrcHigh,
Ipp64f* pDst, int lenDst);

IppStatus ippsWTHaarInv_8s_Sfs(const Ipp8s* pSrcLow, const Ipp8s* pSrcHigh,
Ipp8s* pDst, int lenDst, int scaleFactor);

IppStatus ippsWTHaarInv_16s_Sfs(const Ipp16s* pSrcLow, const Ipp16s* pSrcHigh,
Ipp16s* pDst, int lenDst, int scaleFactor);

IppStatus ippsWTHaarInv_32s_Sfs(const Ipp32s* pSrcLow, const Ipp32s* pSrcHigh,
Ipp32s* pDst, int lenDst, int scaleFactor);

IppStatus ippsWTHaarInv_64s_Sfs(const Ipp64s* pSrcLow, const Ipp64s* pSrcHigh,
Ipp64s* pDst, int lenDst, int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector for forward transform.
<i>lenSrc</i>	Number of elements in the source vector <i>pSrc</i> .
<i>pDstLow</i>	Pointer to the array with the coarse "low frequency" components of the output for forward transform.
<i>pDstHigh</i>	Pointer to the array with the detail "high frequency" components of the output for forward transform.
<i>pSrcLow</i>	Pointer to the array with the coarse "low frequency" components of the input for inverse transform.
<i>pSrcHigh</i>	Pointer to the array with the detail "high frequency" components of the input for inverse transform.

<i>pDst</i>	Pointer to the array with the output signal for inverse transform.
<i>lenDst</i>	Number of elements in the destination vector <i>pDst</i>
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The functions `ippsWTHaarFwd` and `ippsWTHaarInv` are declared in the `ipps.h` file. These functions perform forward and inverse single-level discrete Haar transforms. These transforms are orthogonal and reconstruct the original signal perfectly.

The forward transform can be considered as wavelet signal decomposition with lowpass decimation filter coefficients  $\{1/2, 1/2\}$  and highpass decimation filter coefficients  $\{1/2, -1/2\}$ .

The inverse transform is represented as wavelet signal reconstruction with lowpass interpolation filter coefficients  $\{1, 1\}$  and highpass interpolation filter coefficients  $\{-1, 1\}$ .

The decomposition filter coefficients are frequency response normalized to provide the same value range for both input and output signals. Thus, the amplitude of the low pass filter frequency response is 1 for zero-valued frequency, and the amplitude of the high pass filter frequency response is also 1 for the frequency value near to 0.5.

As the absolute values of the interpolation filter coefficients are equal to 1, the reconstruction of the signal requires few operations. It is well suited for usage in data compression applications. As the decomposition filter coefficients are powers of 2, the integer functions perform lossless decomposition with the *scaleFactor* value equal to -1. To avoid saturation, use higher-precision data types.

Note that the filter coefficients can be power spectral response normalized, see [[Strang96](#)] for more information. Thus, the decomposition filter coefficients are  $\{2^{-1/2}, 2^{-1/2}\}$  and  $\{2^{-1/2}, -2^{-1/2}\}$ ; accordingly; the reconstruction filter coefficients are  $\{2^{-1/2}, 2^{-1/2}\}$  and  $\{-2^{-1/2}, 2^{-1/2}\}$ .

In the following definition of the forward single-level discrete Haar transform,  $N = \text{lenSrc}$ . The coarse "low-frequency" component  $c(k)$  is `pDstLow[k]` and the detail "high-frequency" component  $d(k)$  is `pDstHigh[k]`; also  $x(2k)$  and  $x(2k+1)$  are even and odd values of the input signal *pSrc*, respectively.

$$c(k) = (x(2k) + x(2k+1))/2$$

$$d(k) = (x(2k+1) - x(2k))/2$$

In the inverse direction,  $N = \text{lenDst}$ . The coarse "low-frequency" component  $c(k)$  is `pSrcLow[k]` and the detail "high-frequency" component  $d(k)$  is `pSrcHigh[k]`; also  $y(2i)$  and  $y(2i+1)$  are even and odd values of the output signal *pDst*, respectively.



$$y(2i) = c(i) - d(i)$$

$$y(2i+1) = c(i) + d(i)$$

For even length  $N$ ,  $0 \leq k < N/2$  and  $0 \leq i < N/2$ . Also, “low-frequency” and “high-frequency” components are of size  $N/2$  for both original and reconstructed signals. The total length of components is equal to the signal length  $N$ .

In case of odd length  $N$ , the vector is considered as a vector of the extended length  $N+1$  whose two last elements are equal to each other  $x[N] = x[N - 1]$ . The last elements of the coarse and detail components of the decomposed signal are defined as follows:

$$c((N+1)/2 - 1) = x(N - 1)$$

$$d((N + 1)/2 - 1) = 0$$

Correspondingly, the last element of the reconstructed signal is defined as:

$$y(N) = y(N - 1) = c((N + 1)/2 - 1)$$

For odd length  $N$ ,  $0 \leq k < (N - 1)/2$  and  $0 \leq i < (N - 1)/2$ , assuming that  $c((N + 1)/2 - 1) = x(N - 1)$  and  $y(N - 1) = c((N + 1)/2 - 1)$ . The “low-frequency” component is of size  $(N + 1)/2$ . The “high-frequency” component is of size  $(N - 1)/2$ , because the last element  $d((N + 1)/2 - 1)$  is always equal to 0. The total length of components is also  $N$ .

Such an approach applies continuation of boundaries for filters having the symmetry properties, see [Bris94].

When performing block mode transforms, take into consideration that for decomposition and reconstruction of even-length signals no extrapolations at the boundaries is used. In case of odd-length signals, a symmetric continuation of the signal boundary with the last point replica is applied.

When it is necessary to have a continuous set of output blocks, all the input blocks are to be of even length, besides the last one (which can be either of odd or even length). Thus, if the whole amount of elements is odd, only the last block can be of odd length.

**ippsWTHaarFwd.** This function performs the forward single-level discrete Haar transform of a *lenSrc*-length signal *pSrc* and stores the decomposed coarse “low-frequency” components in *pDstLow*, and the detail “high-frequency” components in *pDstHigh*.

**ippsWTHaarInv.** This function performs the inverse single-level discrete Haar transform of the coarse “low-frequency” components *pSrcLow* and detail “high-frequency” components *pSrcHigh*, and stores the reconstructed signal in the *lenDst*-length vector *pDst*.

For more information on wavelet transforms see [Strang96] and [Bris94].

Example 7-10 below illustrates the use of the function `ippsWTHaarFwd_32f`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than 4 for the function <code>ippsWinBlackmanOpt</code> and less than 3 for all other functions of the family.

## Example 7-11. Using the `ippsWTHaarFwd` Function

```
IppStatus wthaar(void) {
    Ipp32f x[8], lo[4], hi[4];
    IppStatus status;
    ippsSet_32f(7, x, 8); --x[4];
    status = ippsWTHaarFwd_32f(x, 8, lo, hi);
    printf_32f("WT Haar low  =", lo, 4, status);
    printf_32f("WT Haar high =", hi, 4, status);
    return status;
}
```

Output:

```
WT Haar low  =  7.000000 7.000000 6.500000 7.000000
WT Haar high =  0.000000 0.000000 0.500000 0.000000
```

## Transforms for User Filter Banks

This section describes the functions that perform forward or inverse wavelet transforms for user filter banks.

## WTFwdInitAlloc, WTInvInitAlloc

*Initializes the wavelet transform structure.*

---

### Syntax

#### Case 1: Forward transform

```
IppStatus ippsWTFwdInitAlloc_32f(IppsWTFwdState_32f** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTFwdInitAlloc_8s32f(IppsWTFwdState_8s32f** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTFwdInitAlloc_8u32f(IppsWTFwdState_8u32f** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTFwdInitAlloc_16s32f(IppsWTFwdState_16s32f** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTFwdInitAlloc_16u32f(IppsWTFwdState_16u32f** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

#### Case 2: Inverse transform

```
IppStatus ippsWTInvInitAlloc_32f(IppsWTInvState_32f** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInitAlloc_32f8s(IppsWTInvState_32f8s** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInitAlloc_32f8u(IppsWTInvState_32f8u** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInitAlloc_32f16s(IppsWTInvState_32f16s** ppState, const Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int lenHigh, int offsHigh);
```

```
IppStatus ippsWTInvInitAlloc_32f16u(IppsWTInvState_32f16u** ppState, const
Ipp32f* pTapsLow, int lenLow, int offsLow, const Ipp32f* pTapsHigh, int
lenHigh, int offsHigh);
```

## Parameters

<i>ppState</i>	Pointer to the pointer to the allocated and initialized state structure.
<i>pTapsLow</i>	Pointer to the vector of lowpass filter taps.
<i>lenLow</i>	Number of taps in the lowpass filter.
<i>offsLow</i>	Additional delay (offset) of the lowpass filter.
<i>pTapsHigh</i>	Pointer to the vector of highpass filter taps.
<i>lenHigh</i>	Number of taps in the highpass filter.
<i>offsHigh</i>	Additional delay (offset) of the highpass filter.

## Description

The functions `ippsWTFwdInitAlloc` and `ippsWTInvInitAlloc` are declared in the `ipps.h` file. These functions create and initialize the WT state structure *ppState* with the following parameters: the lowpass and highpass filter taps *pTapsLow* and *pTapsHigh*, lengths *lenLow* and *lenHigh*, input additional delays *offsLow* and *offsHigh*, respectively.

`ippsWTFwdInitAlloc`. This function initializes the forward WT state structure.

`ippsWTInvInitAlloc`. This function initializes the inverse WT state structure.

## Application Notes

These functions allocate memory for the wavelet state structure, initialize it, and returns the *ppState* pointer to the state structure. The initialization procedures are implemented separately for forward and inverse transforms. To perform both forward and inverse wavelet transforms, create two separate state structures. In general, the meanings of initialization parameters of forward and inverse transforms are similar. Each function has parameters describing of a pair of filters. The forward transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of analysis filters. The inverse transform uses the taps *pTapsHigh* and *pTapsLow*, and the lengths *lenHigh* and *lenLow* of a pair of synthesis filters. Besides lengths and sets of taps the functions allow to specify an additional delay *offsLow* and *offsHigh* for each filter. The adjustable values of delays allow to synchronize:

- group delays for highpass and lowpass filters;

- delays between data of different levels in multilevel decomposition and reconstruction algorithms.

For more information about using these parameters, see descriptions of the functions [ippsWTFwd](#) and [ippsWTInv](#). The minimum allowed value of the additional delay for the forward transform is -1. For the inverse transform the delay values must be greater or equal to 0. See descriptions of the functions [ippsWTFwd](#) and [ippsWTInv](#) for an example showing how to choose additional delay values. The initialization functions copy filter taps into the state structure *pState*. So all the memory referred to with the pointers can be freed or modified after the functions finished operating. In case of the memory shortage, the function sets a zero pointer to the structure.

**Boundaries extrapolation.** Typically, reversible wavelet transforms of a bounded signal require data extrapolation towards one or both sides. All internal delay lines are set to zero at the stage of initialization. To set a non-zero signal prehistory, call the function [ippsWTFwdSetDlyLine](#). When processed an entire limited data set, data extrapolation may be performed both towards the start and the end of the data vector. For that, the source data and their initial extrapolation are used to form the delay line, the rest of the signal is subdivided into the main block and the signal end. The signal end data and their extrapolation are used to form the last block.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppState</i> , <i>pTapsHigh</i> , or <i>pTapsLow</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>lenLow</i> or <i>lenHigh</i> is less than or equal to 0.
<code>ippStsWtOffsetErr</code>	Indicates an error when the filter delay <i>offsLow</i> or <i>offsHigh</i> is less than -1 for the forward transform; and is less than 0 for the inverse transform.

## WTFwdFree, WTInvFree

*Closes a wavelet transform structure.*

---

### Syntax

#### Case 1: Forward transform

```
IppStatus ippsWTFwdFree_32f(IppsWTFwdState_32f* pState);
IppStatus ippsWTFwdFree_8s32f(IppsWTFwdState_8s32f* pState);
```

```
IppStatus ippsWTFwdFree_8u32f(IppsWTFwdState_8u32f* pState);
IppStatus ippsWTFwdFree_16s32f(IppsWTFwdState_16s32f* pState);
IppStatus ippsWTFwdFree_16u32f(IppsWTFwdState_16u32f* pState);
```

## Case 2: Inverse transform

```
IppStatus ippsWTInvFree_32f(IppsWTInvState_32f* pState);
IppStatus ippsWTInvFree_32f8s(IppsWTInvState_32f8s* pState);
IppStatus ippsWTInvFree_32f8u(IppsWTInvState_32f8u* pState);
IppStatus ippsWTInvFree_32f16s(IppsWTInvState_32f16s* pState);
IppStatus ippsWTInvFree_32f16u(IppsWTInvState_32f16u* pState);
```

## Parameters

*pState*                                      Pointer to the state structure to be closed.

## Description

The functions `ippsWTFwdFree` and `ippsWTInvFree` are declared in the `ipps.h` file. These functions close the WT state structure *pState* by freeing all the internal memory associated with the state created by the functions `ippsWTFwdInitAlloc` or `ippsWTInvInitAlloc`. Call `ippsWTFwdFree` or `ippsWTInvFree` after the transform is completed. If the *pState* pointer is NULL, the function performs no operation and returns the `ippStsNullPtrErr` status.

**ippsWTFwdFree.** This function closes the forward WT state structure.

**ippsWTInvFree.** This function closes the inverse WT state structure.

## Return Values

`ippStsNoErr`                                      Indicates no error.

`ippStsNullPtrErr`                                      Indicates an error when *pState* is NULL.

`ippStsStateMatchErr`                                      Indicates an error when the state identifier *pState* is incorrect.

## WTFwd

*Computes the forward wavelet transform.*

---

### Syntax

```
IppStatus ippsWTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDstLow, Ipp32f* pDstHigh,
int dstLen, IppsWTFwdState_32f* pState);

IppStatus ippsWTFwd_8s32f(const Ipp8s* pSrc, Ipp32f* pDstLow, Ipp32f*
pDstHigh, int dstLen, IppsWTFwdState_8s32f* pState);

IppStatus ippsWTFwd_8u32f(const Ipp8u* pSrc, Ipp32f* pDstLow, Ipp32f*
pDstHigh, int dstLen, IppsWTFwdState_8u32f* pState);

IppStatus ippsWTFwd_16s32f(const Ipp16s* pSrc, Ipp32f* pDstLow, Ipp32f*
pDstHigh, int dstLen, IppsWTFwdState_16s32f* pState);

IppStatus ippsWTFwd_16u32f(const Ipp16u* pSrc, Ipp32f* pDstLow, Ipp32f*
pDstHigh, int dstLen, IppsWTFwdState_16u32f* pState);
```

### Parameters

<i>pSrc</i>	Pointer to the vector which holds the input signal for decomposition.
<i>pDstLow</i>	Pointer to the vector which holds output coarse “low frequency” components.
<i>pDstHigh</i>	Pointer to the vector which holds output detail “high frequency” components.
<i>dstLen</i>	Number of elements in the vectors <i>pDstHigh</i> and <i>pDstLow</i> .
<i>pState</i>	Pointer to the state structure.

### Description

The function `ippsWTFwd` is declared in the `ipps.h` file. This function computes the forward wavelet transform. The function transforms the  $(2*dstLen)$ -length source data block *pSrc* into “low frequency” components *pDstLow* and “high frequency” components *pDstHigh*. The transform parameters are specified in the state structure *pState*.

## Application Notes

These functions perform the one-level forward discrete multi-scale transform. An equivalent transform diagram is shown in Figure 7-4 below. The input signal is divided into the “low frequency” and “high frequency” components. The transfer characteristics of filters are defined by the coefficients set at the initialization stage. The functions are designed for the block processing of data; the transform state structure *pState* contains all needed filter delay lines. Besides these main delay lines each function has an additional delay line for each filter. Adjustable extra delay lines help synchronize group delay times of both highpass and lowpass filters. Moreover, in multilevel systems of signal decomposition delays between different decomposition levels may also be synchronized.

**Input and output data block lengths.** The functions are designed to decompose signal blocks of even length, therefore, these functions have one parameter only, that is the length of input components. The length of the input block must be double the size of each component.

**Filter group delays synchronization.** Some applications may require synchronization of highpass and lowpass filter time responses. A typical example of this synchronization is synchronizing symmetrical filters of different length.

Below follows an example of bi orthogonal set of spline filters of respective length of 6 and 2:

```
static const float decLow[6] =
{
    -6.25000000e-002f,
     6.25000000e-002f,
     5.00000000e-001f,
     5.00000000e-001f,
     6.25000000e-002f,
    -6.25000000e-002f
};

static const float decHigh[2] =
{
    -5.00000000e-001f,
     5.00000000e-001f
};
```

In this case the lowpass filter gives a delay two samples longer than the highpass filter, which is exactly what the difference between additional initialization function delays should be. The following values must be selected to ensure minimum common signal delay, *offsLow* = -1, *offsHigh* = -1 + 2 = 1. In this case the group times of filter delays are balanced by additional delays. The total delay time is equal to the lowpass filter group delay which has the value of two samples in the decomposition stage in the original signal time frame.





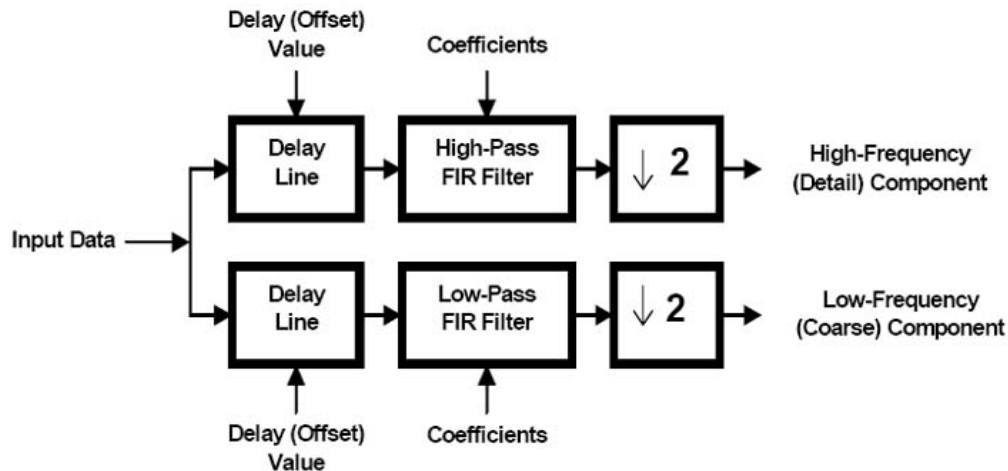
**NOTE.** Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, forward transform additional delays must be uniformly even for faultless signal reconstruction.

**Multilevel decomposition algorithm.** The implementation of multilevel decomposition algorithms may require synchronization of signal delays across components of different levels.

This is illustrated in the example of the three-level decomposition shown in Figure 7-2. Assume that for transformation the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. Since group delay definitely needs to be synchronized, for the last level select additional filter delays  $offsLow3 = -1$ ,  $offsHigh3 = 1$ . Total delay at the last stage of decomposition for this set of filters is two samples. This value corresponds to the time scale of the input of the last stage of decomposition. In order to ensure an equivalent delay of the “detail” part on the second level, the delay must be increased by  $2 \times 2$  samples. Respective values of additional delays for the second level is equal to  $offsLow2 = -1$ ,  $offsHigh2 = offsHigh3 + 4 = 5$ . A greater value of the “high frequency” component delay needs to be selected for the first level of decomposition,  $offsLow1 = -1$ ,  $offsHigh1 = offsHigh2 + 2 \times 4 = 13$ .

Total delay for three levels of decomposition is equal to 12 samples.

**Figure 158: Figure 7-4 One Level Forward Transform**



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state identifier <code>pState</code> is incorrect.
<code>ippStsSizeErr</code>	Indicates an error when <code>dstLen</code> or <code>srcLen</code> is less than or equal to 0.

## WTFwdSetDlyLine, WTFwdGetDlyLine

*Sets and gets the delay lines of the forward wavelet transform.*

---

### Syntax

```

IppStatus ippSWTFwdSetDlyLine_32f(IppsWTFwdState_32f* pState, const Ipp32f*
pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippSWTFwdSetDlyLine_8s32f(IppsWTFwdState_8s32f* pState, const
Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippSWTFwdSetDlyLine_8u32f(IppsWTFwdState_8u32f* pState, const
Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippSWTFwdSetDlyLine_16s32f(IppsWTFwdState_16s32f* pState, const
Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippSWTFwdSetDlyLine_16u32f(IppsWTFwdState_16u32f* pState, const
Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippSWTFwdGetDlyLine_32f(IppsWTFwdState_32f* pState, Ipp32f* pDlyLow,
Ipp32f* pDlyHigh);

IppStatus ippSWTFwdGetDlyLine_8s32f(IppsWTFwdState_8s32f* pState, Ipp32f*
pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippSWTFwdGetDlyLine_8u32f(IppsWTFwdState_8u32f* pState, Ipp32f*
pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippSWTFwdGetDlyLine_16s32f(IppsWTFwdState_16s32f* pState, Ipp32f*
pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippSWTFwdGetDlyLine_16u32f(IppsWTFwdState_16u32f* pState, Ipp32f*
pDlyLow, Ipp32f* pDlyHigh);

```

## Parameters

<i>pState</i>	Pointer to the state structure.
<i>pDlyLow</i>	Pointer to the vector which holds the delay lines for “low frequency” components.
<i>pDlyHigh</i>	Pointer to the vector which holds the delay lines for “high frequency” components.

## Description

The functions `ippsWTFwdSetDlyLine` and `ippsWTFwdGetDlyLine` are declared in the `ipps.h` file. These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and stores them into the state structure *pState*.

`ippsWTFwdSetDlyLine`. This function sets the delay line values of the forward WT state.

`ippsWTFwdGetDlyLine`. This function gets the delay line values of the forward WT state.

## Application Notes

These functions are designed to shape the signal prehistory, save and reconstruct delay lines. Delay lines are implemented separately for highpass and lowpass filters, which gives the option of getting independent signal prehistories for each filter.

**Delay line data format.** Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the initial signal fed into the forward transform functions, i.e., delay line vectors must be made up of a succession of the signal prehistory counts in the same time frame as the initial signal.

**Delay line lengths.** The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter:

$$dlyLowLen = lenLow + offsLow - 1,$$

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter:  $dlyHighLen = lenHigh + offsHigh - 1,$

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the “high frequency” component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are specified by the function `ippsWTFwdInitAlloc`.

## Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <i>pDlyLow</i> or <i>pDlyHigh</i> is NULL.
<code>ippsStsStateMatchErr</code>	Indicates an error when the state identifier <i>pState</i> is incorrect.

## WTInv

*Computes the inverse wavelet transform.*

---

### Syntax

```
IppStatus ippsWTInv_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
srcLen, Ipp32f* pDst, IppsWTInvState_32f* pState);
```

```
IppStatus ippsWTInv_32f8s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
srcLen, Ipp8s* pDst, IppsWTInvState_32f8s* pState);
```

```
IppStatus ippsWTInv_32f8u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh, int
srcLen, Ipp8u* pDst, IppsWTInvState_32f8u* pState);
```

```
IppStatus ippsWTInv_32f16s(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
int srcLen, Ipp16s* pDst, IppsWTInvState_32f16s* pState);
```

```
IppStatus ippsWTInv_32f16u(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,
int srcLen, Ipp16u* pDst, IppsWTInvState_32f16u* pState);
```

### Parameters

<i>pSrcLow</i>	Pointer to the vector which holds input coarse “low frequency” components.
<i>pSrcHigh</i>	Pointer to the vector which holds detail “high frequency” components.
<i>srcLen</i>	Number of elements in the vectors <i>pSrcHigh</i> and <i>pSrcLow</i> .

<i>pDst</i>	Pointer to the vector which holds the output reconstructed signal.
<i>pState</i>	Pointer to the state structure.

## Description

The function `ippsWTInv` is declared in the `ipps.h` file. This function computes the inverse wavelet transform. The function transforms the “low frequency” components *pSrcLow* and “high frequency” components *pSrcHigh* into the  $(2*srcLen)$ -length destination data block *pDst*. The transform parameters are specified in the state structure *pState*.

## Application Notes

These functions are used for one level of inverse multiscale transformation which results in reconstructing the original signal from the two “low frequency” and “high frequency” components. Figure 7-5 below shows an equivalent transform algorithm. Two interpolation filters are used for signal reconstruction; their coefficients are set at the initialization stage. The inverse transform implementation, similar to forward transform implementation, contains additional delay lines needed to synchronize the group time of filter delays and delays across different levels of data reconstruction.

**Input and output data block lengths.** These functions are designed to reconstruct the blocks of the even length signal. The signal component length must be the input data. The length of the output block of the reconstructed signal must be double the length of each of the components.

**Filter group delay synchronization.** In this example consider a biorthogonal set of spline filters of length 2 and 6:

```
static const float recLow[2] =
{
    1.00000000e+000f,
    1.00000000e+000f
};
static const float recHigh[6] =
{
    -1.25000000e-001f,
    -1.25000000e-001f,
    1.00000000e+000f,
    -1.00000000e+000f,
    1.25000000e-001f,
    1.25000000e-001f
};
```

This set of filters corresponds to the set of filters considered in a similar section of the description of the forward transform function `ippsWTFwd`.

Unlike the case described above, this time the highpass filter generates a delay greater by two samples compared against the low frequency filter. The two sample difference should also exist between initialization function additional delays. The following parameters of additional delays need to be selected in order to ensure the minimum total delay,  $offsLow = 2$ ,  $offsHigh = 0$ . In this case the total delay is equal to the highpass filter group delay, which at the decomposition stage is equal to two samples in the original signal time frame.

Total delay of one level of decomposition and reconstruction is equal to 4 samples, considering the decomposition stage delay.




---

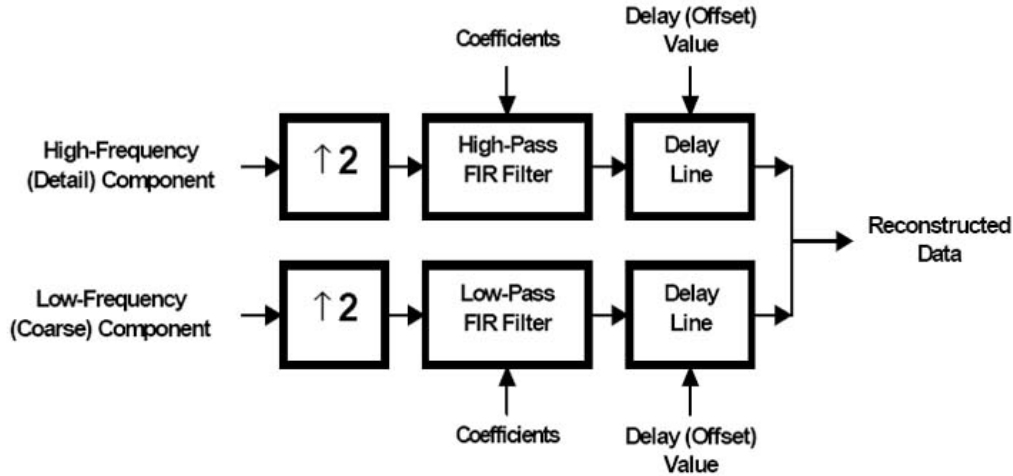
**NOTE.** Biorthogonal and orthogonal filter banks are distinguished by one specific peculiarity, that is, inverse transform additional delays must be uniformly even and opposite to the evenness of the decomposition delays for faultless signal reconstruction.

---

**Multilevel reconstruction algorithms.** An example of a three-level signal reconstruction algorithm is shown in [Figure 7-3](#). The scheme corresponds to the decomposition scheme described in the section of the description of the forward transform function `ippsWTFwd`. Therefore, for the inverse transform the biorthogonal set of spline filters with respective filter length of 6 and 2 is used. The lowest level filter delays are set to  $offsLow3 = 2$ ,  $offsHigh3 = 0$ . The total delay at this stage of reconstruction is equal to two samples. In order to ensure an equivalent delay of the “detail” part in the middle level, the delay must be increased. Respective values of additional delays for the second level are equal to  $offsLow2 = 2$ ,  $offsHigh2 = offsHigh3 + 2*2 = 4$ . A greater value of high frequency component delay needs to be selected for the last level of reconstruction,  $offsLow1 = -1$ ,  $offsHigh1 = offsHigh2 + 2*4 = 12$ .

The total delay for three levels of reconstruction is equal to 12 samples. The total delay of the three-level decomposition and reconstruction cycle is equal to 24 samples.

**Figure 159: Figure 7-5. One Level Inverse Wavelet Transform**



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsStateMatchErr</code>	Indicates an error when the state identifier <code>pState</code> is incorrect.
<code>ippStsSizeErr</code>	Indicates an error when <code>dstLen</code> or <code>srcLen</code> is less than or equal to 0.

## WTInvSetDlyLine, WTInvGetDlyLine

Sets and gets the delay lines of the inverse wavelet transform.

### Syntax

```

IppStatus ippSWTInvSetDlyLine_32f(IppsWTInvState_32f* pState, const Ipp32f*
pDlyLow, const Ipp32f* pDlyHigh);
  
```

```

IppStatus ippsWTInvSetDlyLine_32f8s(IppsWTInvState_32f8s* pState, const
Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTInvSetDlyLine_32f8u(IppsWTInvState_32f8u* pState, const
Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTInvSetDlyLine_32f16s(IppsWTInvState_32f16s* pState, const
Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTInvSetDlyLine_32f16u(IppsWTInvState_32f16u* pState, const
Ipp32f* pDlyLow, const Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f(IppsWTInvState_32f* pState, Ipp32f* pDlyLow,
Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f8s(IppsWTInvState_32f8s* pState, Ipp32f*
pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f8u(IppsWTInvState_32f8u* pState, Ipp32f*
pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f16s(IppsWTInvState_32f16s* pState, Ipp32f*
pDlyLow, Ipp32f* pDlyHigh);

IppStatus ippsWTInvGetDlyLine_32f16u(IppsWTInvState_32f16u* pState, Ipp32f*
pDlyLow, Ipp32f* pDlyHigh);

```

## Parameters

<i>pState</i>	Pointer to the state structure.
<i>pDlyLow</i>	Pointer to the vector which holds delay lines for “low frequency” components.
<i>pDlyHigh</i>	Pointer to the vector which holds delay lines for “high frequency” components.

## Description

The functions `ippsWTInvSetDlyLine` and `ippsWTInvGetDlyLine` are declared in the `ipps.h` file. These functions copy the delay line values from *pDlyHigh* and *pDlyLow*, and store them into the state structure *pState*.

**ippsWTInvSetDlyLine.** This function sets the delay line values of the inverse WT state.

**ippsWTInvGetDlyLine.** This function gets the delay line values of the inverse WT state.



## Application Notes

These functions set and read delay lines of inverse multiscale transformation. The functions receive or return filter low and high frequency component delay line vectors. The functions may be used to shape previous history of each of the components. Installation functions and read functions together ensure that delay lines from each filter are saved and reconstructed.

**Delay line data format.** Despite that any delay line formats could be used inside transformations, the functions provide the simplest format of received and returned vectors. Data either transferred to or returned from the delay lines have the same format as the low and high frequency components at the input of the inverse transform functions. Thus, delay line vectors must be made up of a succession of signal prehistory counts in the same time frame as the input components.

**Delay line lengths.** The length of the vectors that are transferred to or received by the delay line installation or reading functions is uniquely defined by the filter length and the value of additional filter delay.

The following expression defines the length of the delay line vector of the “low frequency” component filter in terms of the C language (integer division by two is used here for simplicity):

```
dlyLowLen = (lenLow + offsLow - 1) / 2,
```

where *lenLow* and *offsLow* are respectively the length and additional delay of the “low frequency” component filter.

The following expression defines the length of the delay line vector of the “high frequency” component filter in terms of the C language:

```
dlyHighLen = (lenHigh + offsHigh - 1) / 2,
```

where *lenHigh* and *offsHigh* are respectively the length and additional delay of the “high frequency” component filter.

The *lenLow*, *offsLow*, *lenHigh*, and *offsHigh* parameters are specified by the function [ippsWTInvInitAlloc](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDlyLow</i> or <i>pDlyHigh</i> is NULL.
<code>ippStsStateMatchErr</code>	Indicates an error when the state identifier <i>pState</i> is incorrect.

## Wavelet Transforms Example

The delay line paradigm is well-known interface solution for functions that require some pre-history in the streaming processing. In such application the use of the Intel IPP wavelet transform functions is similar to the use of the FIR, IIR, or multi-rate filters. (See also the discussion on the [synchronization](#) of low-pass and high-pass filter delays in this chapter.) But very often the wavelet transforms are used to process entire non-streaming data by extending with borders that are suitable for filter bank type that are used in transforms.

The following code example demonstrates how to implement this approach using the Intel IPP functions. It performs forward and inverse wavelet transforms of a short vector containing 12 elements. It uses Daubechies filter bank of the order 2 (that allows the perfect reconstruction) and periodical data extension by wrapping.

It is also may be useful as an illustration of how to fill delay line, if you need non-zero pre-history of signal in streaming applications.

**Example 7-11 Using Wavelet Transforms Functions**

```
#include <stdio.h>

#include "ipp.h"

// Filter bank for Daubechies, order 2
static const int fwdFltLenL = 4;
static const int fwdFltLenH = 4;
static const float fwdFltL[4] =
{
    -1.294095225509215e-001f,
    2.241438680418574e-001f,
    8.365163037374690e-001f,
    4.829629131446903e-001f
};

static const float fwdFltH[4] =
{
    -4.829629131446903e-001f,
    8.365163037374690e-001f,
    -2.241438680418574e-001f,
    -1.294095225509215e-001f
};

static const int invFltLenL = 4;
static const int invFltLenH = 4;
static const float invFltL[4] =
{
    4.829629131446903e-001f,
    8.365163037374690e-001f,
    2.241438680418574e-001f,
    -1.294095225509215e-001f
```

```
};

static const float invFltH[4] =
{
    -1.294095225509215e-001f,
    -2.241438680418574e-001f,
    8.365163037374690e-001f,
    -4.8296291314446903e-001f
};

// minimal values
static const int fwdFltOffsL = -1;
static const int fwdFltOffsH = -1;
// minimal values, that corresponds to perfect reconstruction
static const int invFltOffsL = 0;
static const int invFltOffsH = 0;

void main(void)
{
    Ipp32f src[] = {1, -10, 324, 48, -483, 4, 7, -5532, 34, 8889, -57, 54};
    Ipp32f low[7];
    Ipp32f high[7];
    int i;
    printf("original:\n");
    for(i = 0; i < 12; i++) printf("%.0f; ", src[i]);
    printf("\n");
    {
        IppsWTFwdState_32f* state;
        ippsWTFwdInitAlloc_32f (&state, fwdFltL, fwdFltLenL, fwdFltOffsL,
                                fwdFltH, fwdFltLenH, fwdFltOffsH);

        // We substitute wrapping extension in "the beginning of stream"
```

---

```
// Here should be the same pointers for this offsets,  
// but in the general case it may be different  
ippsWTFwdSetDlyLine_32f( state, &src[10], &src[10]);  
// Forward transform  
ippsWTFwd_32f  
    (src, low, high, 6, state);  
ippsWTFwdFree_32f      (state);  
// print decomposition result
```

```

printf("approx:\n");
for(i = 0; i < 6; i++) printf("%.4f; ", low[i]);
printf("\n");
printf("details:\n");
for(i = 0; i < 6; i++) printf("%.4f; ", high[i]);
printf("\n");
}
{
Ipp32f dst[12];
IppsWTInvState_32f* state;
ippsWTInvInitAlloc_32f (&state,
    invFltL, invFltLenL, invFltOffsL,
    invFltH, invFltLenH, invFltOffsH);
// For this particular case (non-shifted reconstruction)
// here is first data itself,
// that we need to place to delay line
// [(invFltLenL + invFltOffsL - 1) / 2] elements for l. filtering
// [(invFltLenH + invFltOffsH - 1) / 2] elements for h. filtering
ippsWTInvSetDlyLine_32f( state, low, high);
// Inverse transform
ippsWTInv_32f          (&low[1], &high[1], 5, dst, state);
// Here are the substitution of the wrapping extension
// at the "end of stream" and calculation of last samples of reconstruction
// We do not use additional buffer and do not copy any data externally,
// just substitute beginning of input data itself to simulate wrapping
ippsWTInv_32f          (low, high, 1, &dst[10], state);
ippsWTInvFree_32f      (state);
// print reconstruction result
printf("reconstruction:\n");

```

```
    for(i = 0; i < 12; i++) printf("%.0f; ", dst[i]);  
    printf("\n");  
}  
}
```

After compiling and running it gives the following console output:

original:

1; -10; 324; 48; -483; 4; 7; -5532; 34; 8889; -57; 54;

approx:

19.1612; 58.5288; 87.8536; 487.5375; -5766.9277; 7432.4502;

details:

0.9387; 249.9611; -458.6568; 2739.2146; -3025.5576; -2070.5762;

reconstruction:

1; -10; 324; 48; -483; 4; 7; -5532; 34; 8889; -57; 54;

The program prints on console the original data, approximation, and details components after forward transform and perfect reconstruction of original data after inverse transform.

---

---



# Speech Recognition Functions

This chapter describes Intel® IPP functions that are designed for use in speech recognition applications. Table 8-1 gives the full list of functions in this group.

**Table 8-1 Intel IPP Speech Recognition Functions**

Function Base Name	Operation
Basic Arithmetics Functions	
AddAllRowSum	Calculates the sums of column vectors in a matrix and adds the sums to a vector.
SumColumn	Calculates sums of column vectors in a matrix.
SumRow	Calculates sums of row vectors in a matrix.
SubRow	Subtracts a vector from all matrix rows.
CopyColumn_Indirect	Copies the input matrix with columns redirection.
BlockDMatrixInitAlloc	Initializes the structure that represents a symmetric block diagonal matrix.
BlockDMatrixFree	Deallocates the block diagonal matrix structure.
NthMaxElement	Searches for the N-th maximal element of a vector.
VecMatMul	Multiplies a vector by a matrix.
MatVecMul	Multiplies a matrix by a vector
Feature Processing Functions	
ZeroMean	Subtracts the mean value from the input vector.
CompensateOffset	Removes the DC offset of the input signals.
SignChangeRate	Counts the zero-cross rate for the input signal.
LinearPrediction	Performs linear prediction analysis on the input vector.
Durbin	Performs Durbin's recursion on an input vector of autocorrelations.
Schur	Calculates reflection coefficients using Schur algorithm.
LPToSpectrum	Calculates smoothed magnitude spectrum.
LPToCepstrum	Calculates cepstrum coefficients from linear prediction coefficients.
CepstrumToLP	Calculates linear prediction coefficients from cepstrum coefficients.
LPToReflection	Calculates the linear prediction reflection coefficients from the linear prediction coefficients.
ReflectionToLP	Calculates the linear prediction coefficients from the linear prediction reflection coefficients.

Function Base Name	Operation
<a href="#">ReflectionToAR, ReflectionToLAR, ReflectionToTrueAR</a>	Converts reflection coefficients to area ratios.
<a href="#">ReflectionToTilt</a>	Calculates tilt for rise/fall/connection parameters.
<a href="#">PitchmarkToF0Cand</a>	Calculates rise and fall amplitude and duration for tilt.
<a href="#">UnitCurve</a>	Calculates tilt for rise and fall coefficients.
<a href="#">LPToLSP</a>	Calculates line spectrum pairs vector from linear prediction coefficients.
<a href="#">LSPToLP</a>	Converts line spectrum pairs vector to linear prediction coefficients.
<a href="#">MelToLinear</a>	Converts Mel-scaled values to linear scale values.
<a href="#">LinearToMel</a>	Converts linear-scale values to Mel-scale values.
<a href="#">CopyWithPadding</a>	Copies the input signal to the output with zero-padding.
<a href="#">MelFBankGetSize</a>	Gets the size of the Mel-frequency filter bank structure.
<a href="#">MelFBankInit</a>	Initializes the structure for performing the Mel-frequency filter bank analysis.
<a href="#">MelFBankInitAlloc</a>	Initializes the structure and allocates memory for performing the Mel-frequency filter bank analysis.
<a href="#">MelLinFBankInitAlloc</a>	Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.
<a href="#">EmptyFBankInitAlloc</a>	Initializes an empty filter bank structure.
<a href="#">FBankFree</a>	Destroys the structure for the filter bank analysis.
<a href="#">FBankGetCenters</a>	Retrieves the center frequencies of the triangular filter banks.
<a href="#">FBankSetCenters</a>	Sets the center frequencies of the triangular filter banks.
<a href="#">FBankGetCoeffs</a>	Retrieves the filter bank weight coefficients.
<a href="#">FBankSetCoeffs</a>	Sets the filter bank weight coefficients.
<a href="#">EvalFBank</a>	Performs the filter bank analysis.
<a href="#">DCTLifterGetSize_MulC0</a>	Gets the size of the DCT structure.
<a href="#">DCTLifterInit_MulC0</a>	Initializes the structure to perform DCT and lift the DCT coefficients.
<a href="#">DCTLifterInitAlloc</a>	Initializes the structure and allocates memory to perform DCT and lift the DCT coefficients.
<a href="#">DCTLifterFree</a>	Destroys the structure used for the DCT and lifting.
<a href="#">DCTLifter</a>	Performs the DCT and lifts the DCT coefficients.
<a href="#">NormEnergy</a>	Normalizes a vector of energy values.
<a href="#">SumMeanVar</a>	Calculates both the sum of a the vector and its square sum.

Function Base Name	Operation
<a href="#">NewVar</a>	Calculates the variances given the sum and square sum accumulators.
<a href="#">RecSqrt</a>	Calculates square roots of a vector and their reciprocals.
<a href="#">AccCovarianceMatrix</a>	Accumulates covariance matrix.
Derivative Functions	
<a href="#">CopyColumn</a>	Copies the input sequence into the output sequence.
<a href="#">EvalDelta</a>	Calculates the derivatives of feature vectors.
<a href="#">Delta</a>	Copies the base features and calculates the derivatives of feature vectors.
<a href="#">DeltaDelta</a>	Copies the base features and calculates their first and second derivatives.
Pitch Super Resolution Functions	
<a href="#">CrossCorrCoeffDecim</a>	Calculates vector of cross correlation coefficients with decimation.
<a href="#">CrossCorrCoeff</a>	Calculates the cross correlation coefficient.
<a href="#">CrossCorrCoeffInterpolation</a>	Calculates interpolated cross correlation coefficient.
Model Evaluation Functions	
<a href="#">AddNRows</a>	Adds N vectors from a vector array.
<a href="#">ScaleLM</a>	Scales vector elements with thresholding.
<a href="#">LogAdd</a>	Adds two vectors in the logarithmic representation.
<a href="#">LogSub</a>	Subtracts a vector from another vector, in the logarithmic representation.
<a href="#">LogSum</a>	Sums vector elements in the logarithmic representation.
<a href="#">MahDistSingle</a>	Calculates the Mahalanobis distance for a single observation vector.
<a href="#">MahDist</a>	Calculates the Mahalanobis distances for multiple observation vectors.
<a href="#">MahDistMultiMix</a>	Calculates the Mahalanobis distances for multiple means and variances.
<a href="#">LogGaussSingle</a>	Calculates the observation probability for a single Gaussian with an observation vector.
<a href="#">LogGauss</a>	Calculates the observation probability for a single Gaussian with multiple observation vectors.
<a href="#">LogGaussMultiMix</a>	Calculates the observation probability for multiple Gaussian mixture components.
<a href="#">LogGaussMax</a>	Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.

Function Base Name	Operation
<a href="#">LogGaussMaxMultiMix</a>	Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.
<a href="#">LogGaussAdd</a>	Calculates the likelihood probability for multiple observation vectors.
<a href="#">LogGaussAddMultiMix</a>	Calculates the likelihood probability for multiple Gaussian mixture components.
<a href="#">LogGaussMixture</a>	Calculates the likelihood probability for the Gaussian mixture.
<a href="#">LogGaussMixture_Select</a>	Calculates the likelihood probability for the Gaussian mixture using Gaussian selection.
<a href="#">BuildSignTable</a>	Fills sign table for Gaussian mixture calculation.
<a href="#">FillShortlist_Row</a>	Fills row-wise shortlist table for Gaussian selection.
<a href="#">FillShortlist_Column</a>	Fills column-wise shortlist table for Gaussian selection.
<a href="#">DTW</a>	Computes the distance between observation and reference vector sequences using Dynamic Time Warping algorithm.
<b>Model Estimation Functions</b>	
<a href="#">MeanColumn</a>	Computes the mean values for the column elements.
<a href="#">VarColumn</a>	Calculates the variances for the column elements.
<a href="#">MeanVarColumn</a>	Calculates the means and variances for the column elements of a matrix.
<a href="#">WeightedMeanColumn</a>	Computes the weighted mean values for the column elements.
<a href="#">WeightedVarColumn</a>	Computes the weighted variance values for the column elements.
<a href="#">WeightedMeanVarColumn</a>	Computes weighted mean and variance values for the column elements.
<a href="#">NormalizeColumn</a>	Normalizes the matrix columns given the column means and variances.
<a href="#">GaussianSplit</a>	Normalizes and scales input vector elements.
<a href="#">MeanVarAcc</a>	Accumulates the estimates for the mean and variance re-estimation.
<a href="#">GaussianDist</a>	Calculates the distance between two Gaussians.
<a href="#">GaussianSplit</a>	Splits a single Gaussian component into two with the same variance.
<a href="#">GaussianMerge</a>	Merges two Gaussian probability distribution functions.
<a href="#">Entropy</a>	Calculates entropy of the input vector.
<a href="#">SinC</a>	Calculates sine divided by its argument.
<a href="#">ExpNegSqr</a>	Calculates exponential of the squared argument taken with the inverted sign.

Function Base Name	Operation
<a href="#">BhatDist</a>	Calculates the Bhattacharia distance between two Gaussians.
<a href="#">UpdateMean</a>	Updates the mean vector in the EM training algorithm.
<a href="#">UpdateVar</a>	Updates the variance vector in the EM training algorithm.
<a href="#">UpdateWeight</a>	Updates the weight values of Gaussian mixtures in the EM training algorithm.
<a href="#">UpdateGConst</a>	Updates the fixed constant in the Gaussian output probability density function.
<a href="#">OutProbPreCalc</a>	Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.
<a href="#">DcsClustLAccumulate</a>	Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.
<a href="#">DcsClustLCompute</a>	Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.
<b>Model Adaptation Functions</b>	
<a href="#">AddMulColumn</a>	Adds a weighted matrix column to the other column.
<a href="#">AddMulRow</a>	Adds a weighted vector to the other vector.
<a href="#">QRTransColumn</a>	Performs the QR transformation.
<a href="#">DotProdColumn</a>	Calculates the dot product of two matrix columns.
<a href="#">MulColumn</a>	Multiplies a matrix column by a value.
<a href="#">SumColumnAbs</a>	Calculates the absolute sum of matrix column elements.
<a href="#">SumColumnSqr</a>	Calculates the square sums of weighted matrix column elements.
<a href="#">SumRowAbs</a>	Calculates the absolute sum of the vector elements.
<a href="#">SumRowSqr</a>	Calculates the square sum of weighted vector elements.
<a href="#">SVD, SVDSort</a>	Performs Single Value Decomposition on a matrix.
<a href="#">WeightedSum</a>	Calculates the weighted sums of two input vector elements.
<b>Vector Quantization Functions</b>	
<a href="#">FormVector</a>	Constructs an output vector of multiple streams from codebook entries.
<a href="#">CdbkGetSize</a>	Calculates the size in bytes of the codebook.
<a href="#">CdbkInit</a>	Initializes the structure that contains the codebook.
<a href="#">CdbkInitAlloc</a>	Initializes the codebook structure.
<a href="#">CdbkFree</a>	Destroys the codebook structure.
<a href="#">GetCdbkSize</a>	Retrieves the number of codevectors in the codebook.

Function Base Name	Operation
<a href="#">GetCodebook</a>	Retrieves the codevectors from the codebook.
<a href="#">VQ</a>	Quantizes the input vectors given a codebook.
<a href="#">SplitVQ</a>	Quantizes a multiple-stream vector given the codebooks.
<a href="#">VQSingle_Sort</a> , <a href="#">VQSingle_Thresh</a>	Quantizes the input vector given a codebook and gets several closest clusters.
<a href="#">FormVectorVQ</a>	Constructs multiple-stream vectors from codebooks, given indexes.
<b>Polyphase Resampling Functions</b>	
<a href="#">ResamplePolyphaseInit</a>	Initializes the structure for polyphase resampling without calculating the filter coefficients.
<a href="#">ResamplePolyphaseGetSize</a>	Gets the size of the polyphase resampling structure.
<a href="#">ResamplePolyphaseSetFilter</a>	Sets polyphase resampling filter coefficients.
<a href="#">ResamplePolyphaseGetFilter</a>	Gets polyphase resampling filter coefficients.
<a href="#">ResamplePolyphaseInitAlloc</a>	Initializes the structure for polyphase data resampling.
<a href="#">ResamplePolyphaseFree</a>	Free structure for polyphase data resampling.
<a href="#">ResamplePolyphase</a>	Resamples input data using polyphase filters.
<b>Advanced Aurora Functions</b>	
<a href="#">SmoothedPowerSpectrumAurora</a>	Calculates smoothed magnitude of the FFT output.
<a href="#">NoiseSpectrumUpdate_Aurora</a>	Updates the noise spectrum.
<a href="#">WienerFilterDesign_Aurora</a>	Calculates an improved transfer function of the adaptive Wiener filter.
<a href="#">MelFBankInitAlloc_Aurora</a>	Initializes the structure for performing the Mel-frequency filter bank analysis.
<a href="#">TabsCalculation_Aurora</a>	Calculates filter coefficients for residual filter.
<a href="#">ResidualFilter_Aurora</a>	Calculates a denoised waveform signal.
<a href="#">WaveProcessing_Aurora</a>	Processes waveform data after noise reduction.
<a href="#">LowHighFilter_Aurora</a>	Calculates low band and high band filters.
<a href="#">HighBandCoding_Aurora</a>	Codes and decodes the high frequency band energy values.
<a href="#">BlindEqualization_Aurora</a>	Equalizes the cepstral coefficients.
<a href="#">DeltaDelta_Aurora</a>	Calculates the first and second derivatives according to ETSI ES 202 050 standard.
<a href="#">VADGetBufSize_Aurora</a>	Queries the memory size for VAD decision.
<a href="#">VADInit_Aurora</a>	Gets the VAD structure size.
<a href="#">VADDecision_Aurora</a>	Takes the VAD decision.
<a href="#">VADFlush_Aurora</a>	Takes VAD decision for zero input frame.
<b>Ephraim-Malah Noise Suppressor Functions</b>	
<a href="#">FilterUpdateEMNS</a>	Calculates the noise suppression filter coefficients.

Function Base Name	Operation
<a href="#">FilterUpdateWiener</a>	Calculates the Wiener filter coefficients.
<a href="#">GetSizeMCRA</a>	Calculates the size in bytes required for the state structure.
<a href="#">InitMCRA</a>	Initializes the MCRA state structure.
<a href="#">AltInitMCRA</a>	Allocates memory and initializes the MCRA state structure.
<a href="#">UpdateNoisePSDMCRA</a>	Re-estimates the noise power spectrum.
Voice Activity Detector Functions	
<a href="#">FindPeaks</a>	Identifies peaks in the input vector.
<a href="#">PeriodicityLSPE</a>	Computes the periodicity of the input speech frame.
<a href="#">Periodicity</a>	Computes the periodicity of the input block.



**NOTE.** In the below descriptions of function arguments, when an argument refers to a vector or an array, the expression in square brackets given after the explanation of the argument specifies the dimension of that vector or array. In this chapter, unlike other chapters of this manual, arguments that represent step values are measured in elements of the corresponding array, unless explicitly stated to the contrary.

The use of the Intel IPP speech recognition functions is demonstrated in the Intel<sup>®</sup> IPP Samples. See *Intel IPP Speech Recognition Samples* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## Basic Arithmetics

The functions described in this section perform generic arithmetic operations on vectors and matrices.

### AddAllRowSum

*Calculates the sums of column vectors in a matrix and adds the sums to a vector.*

#### Syntax

```
IppStatus ippsAddAllRowSum_32f_D2(const Ipp32f* pSrc, int step, int height,
Ipp32f* pSrcDst, int width);
```

```
IppStatus ippsAddAllRowSum_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
pSrcDst, int width);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height*step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pSrcDst</i>	Pointer to the output vector [ <i>width</i> ].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the output vector <i>pSrcDst</i> .

### Description

The function `ippsAddAllRowSum` is declared in the `ippsr.h` file. This function calculates sums of column vectors in the input matrix, and adds these sums to the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} pSrc[i \cdot step + j],$$

$$0 \leq j < width.$$

For functions with the D2L suffix,

$$pSrcDst[j] = pSrcDst[j] + \sum_{i=0}^{height-1} mSrc[i][j],$$

$$0 \leq j < width.$$



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## SumColumn

*Calculates sums of column vectors in a matrix.*

### Syntax

```

IppStatus ippsSumColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int step, int height,
Ipp32s* pDst, int width, int scaleFactor);

IppStatus ippsSumColumn_16s32f_D2(const Ipp16s* pSrc, int step, int height,
Ipp32f* pDst, int width);

IppStatus ippsSumColumn_32f_D2(const Ipp32f* pSrc, int step, int height,
Ipp32f* pDst, int width);

IppStatus ippsSumColumn_64f_D2(const Ipp64f* pSrc, int step, int height,
Ipp64f* pDst, int width);

IppStatus ippsSumColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height, Ipp32s*
pDst, int width, int scaleFactor);

IppStatus ippsSumColumn_16s32f_D2L(const Ipp16s** mSrc, int height, Ipp32f*
pDst, int width);

IppStatus ippsSumColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
pDst, int width);

IppStatus ippsSumColumn_64f_D2L(const Ipp64f** mSrc, int height, Ipp64f*
pDst, int width);

```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> * <i>step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in the input vector <i>pSrc</i> .

<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>pDst</i>	Pointer to the output vector [ <i>width</i> ].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> and the length of the output vector <i>pDst</i> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippSumColumn` is declared in the `ippsr.h` file. This function calculates sums of column vectors in the input matrix, and stores these sums in the output vector. The operations are as follows:

For functions with the D2 suffix,

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i \cdot step + j], \quad \dots$$

$$0 \leq j < width.$$

For functions with the D2L suffix,

$$pDst[j] = \sum_{i=0}^{height-1} mSrc[i][j], \quad \dots$$

$$0 \leq j < width.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## SumRow

*Calculates sums of row vectors in a matrix.*

---

### Syntax

```

IppStatus ippsSumRow_16s32s_D2Sfs(const Ipp16s* pSrc, int width, int step,
Ipp32s* pDst, int height, int scaleFactor);

IppStatus ippsSumRow_16s32f_D2(const Ipp16s* pSrc, int width, int step,
Ipp32f* pDst, int height);

IppStatus ippsSumRow_32f_D2(const Ipp32f* pSrc, int width, int step, Ipp32f*
pDst, int height);

IppStatus ippsSumRow_64f_D2(const Ipp64f* pSrc, int width, int step, Ipp64f*
pDst, int height);

IppStatus ippsSumRow_16s32s_D2LSfs(const Ipp16s** mSrc, int width, Ipp32s*
pDst, int height, int scaleFactor);

IppStatus ippsSumRow_16s32f_D2L(const Ipp16s** mSrc, int width, Ipp32f* pDst,
int height);

IppStatus ippsSumRow_32f_D2L(const Ipp32f** mSrc, int width, Ipp32f* pDst,
int height);

IppStatus ippsSumRow_64f_D2L(const Ipp64f** mSrc, int width, Ipp64f* pDst,
int height);

```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> * <i>step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> , and also the length of the output vector <i>pDst</i> .
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the output vector [ <i>height</i> ].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsSumRow` is declared in the `ippsr.h` file. This function calculates sums of row vectors in the input matrix *mSrc* and stores these sums in the output vector *pDst*. The operations are as follows:

For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[i \cdot step + j],$$

$$0 \leq i < height.$$

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} mSrc[i][j],$$

$$0 \leq i < height.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## SubRow

*Subtracts a vector from all matrix rows.*

---

### Syntax

```
IppStatus ippsSubRow_16s_D2(const Ipp16s* pSrc, int width, Ipp16s* pSrcDst,
int dstStep, int height);
```

```
IppStatus ippsSubRow_32f_D2(const Ipp32f* pSrc, int width, Ipp32f* pSrcDst,
int dstStep, int height);
```

```
IppStatus ippsSubRow_16s_D2L(const Ipp16s* pSrc, Ipp16s** mSrcDst, int width,
int height);
```

```
IppStatus ippsSubRow_32f_D2L(const Ipp32f* pSrc, Ipp32f** mSrcDst, int width,
int height);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>width</i> ].
<i>pSrcDst</i>	Pointer to the source and destination vector [ <i>height*dstStep</i> ].
<i>mSrcDst</i>	Pointer to the source and destination matrix [ <i>height</i> ][ <i>width</i> ].
<i>width</i>	Number of columns in the matrix <i>mSrcDst</i> .
<i>dstStep</i>	Row step in the vector <i>pSrcDst</i> .
<i>height</i>	Number of rows in the matrix <i>mSrcDst</i> .

### Description

The function `ippsSubRow` is declared in the `ippsr.h` file. This function subtracts the input vector *pSrc* from all matrix rows. The operations are as follows:

For functions with the D2 suffix,

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] - pSrc[j],$$

$$0 \leq i < height, 0 \leq j < width.$$

For functions with the D2L suffix,

$$mSrcDst[i][j] = mSrcDst[i][j] - pSrc[j],$$

$0 \leq i < height, 0 \leq j < width.$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## CopyColumn\_Indirect

Copies the input matrix with columns redirection.

### Syntax

```
ippStatus ippsCopyColumn_Indirect_16s_D2(const Ipp16s* pSrc, int srcLen, int
srcStep, Ipp16s* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
height);
```

```
ippStatus ippsCopyColumn_Indirect_32f_D2(const Ipp32f* pSrc, int srcLen, int
srcStep, Ipp32f* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
height);
```

```
ippStatus ippsCopyColumn_Indirect_64f_D2 (const Ipp64f* pSrc, int srcLen,
int srcStep, Ipp64f* pDst, const Ipp32s* pIndx, int dstLen, int dstStep, int
height);
```

```
ippStatus ippsCopyColumn_Indirect_16s_D2L(const Ipp16s** mSrc, int srcLen,
Ipp16s** mDst, const Ipp32s* pIndx, int dstLen, int height);
```

```
ippStatus ippsCopyColumn_Indirect_32f_D2L(const Ipp32f** mSrc, int srcLen,
Ipp32f** mDst, const Ipp32s* pIndx, int dstLen, int height);
```

```
ippStatus ippsCopyColumn_Indirect_64f_D2L (const Ipp64f** mSrc, int srcLen,
Ipp64f** mDst, const Ipp32s* pIndx, int dstLen, int height);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> * <i>srcStep</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>srcLen</i> ].
<i>srcLen</i>	Number of columns in the input matrix <i>mSrc</i> .
<i>srcStep</i>	Row step in the vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the output vector [ <i>height</i> * <i>dstStep</i> ].
<i>mDst</i>	Pointer to the output matrix [ <i>height</i> ][ <i>dstLen</i> ].
<i>pIndx</i>	Pointer to the redirection vector [ <i>dstLen</i> ].
<i>dstLen</i>	Number of columns in the output matrix <i>mDst</i> .
<i>dstStep</i>	Row step in the vector <i>pDst</i> .
<i>height</i>	Number of rows in both the input and output matrices.

## Description

The function `ippsCopyColumn_Indirect` is declared in the `ippsr.h` file. This function copies the input matrix *mSrc* to the output matrix *mDst* with the columns redirected by *pIndx*. The operations are as follows:

For functions with the D2 suffix,

$$pDst[i*dstStep + j] = pSrc[i*srcStep + pIndx[j]], 0 \leq i < height, 0 \leq j < dstLen.$$

For functions with the D2L suffix,

$$mDst[i][j] = mSrc[i][pIndx[j]], 0 \leq i < height, 0 \leq j < dstLen.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>srcLen</i> , or <i>dstLen</i> is less than or equal to 0, or <i>pIndx</i> [ <i>j</i> ] $\geq$ <i>srcLen</i> or <i>pIndx</i> [ <i>j</i> ] $< 0$ for $0 \leq j < dstLen$ .
<code>ippStsStrideErr</code>	Indicates an error when <i>srcStep</i> is less than <i>srcLen</i> or <i>dstStep</i> is less than <i>dstLen</i> .

## BlockDMatrixInitAlloc

*Initializes the structure that represents a symmetric block diagonal matrix.*

---

### Syntax

```

IppStatus ippsBlockDMatrixInitAlloc_16s(IppsBlockDMatrix_16s** pMatrix, const
Ipp16s** mSrc, const int* bSize, int nBlocks);

IppStatus ippsBlockDMatrixInitAlloc_32f(IppsBlockDMatrix_32f** ppMatrix,
const Ipp32f** mSrc, const int* bSize, int nBlocks);

IppStatus ippsBlockDMatrixInitAlloc_64f(IppsBlockDMatrix_64f** pMatrix, const
Ipp64f** mSrc, const int* bSize, int nBlocks);

```

### Parameters

<i>pMatrix</i>	Pointer to the block diagonal matrix to be created.
<i>mSrc</i>	Pointer to the vector of pointers to matrix rows.
<i>bSize</i>	Pointer to vector of block sizes [ <i>nBlocks</i> ].
<i>nBlocks</i>	Number of blocks in the matrix.

### Description

The function `ippsBlockDMatrixInitAlloc` is declared in the `ippsr.h` file. This function creates the structure that contains a symmetric block diagonal matrix. Matrix elements outside the blocks are assumed to be zero. The block diagonal matrix *A* is defined as follows:

$$A[K_l + i][K_l + j] = A[K_l + j][K_l + i] = mSrc[K_l + i][j],$$

for  $i, j = 0 \dots bSize[l] - 1$  and  $l = 0 \dots nBlocks - 1$ ,

where

$$K_l = \sum_{k < l} bSize[k]$$

and `mSrc[Kl + i]` points to the non-zero part of the matrix *A* row. The size of the matrix is equal to  $K_{nBlocks}$  by  $K_{nBlocks}$ .



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>bSize</code> or <code>nBlocks</code> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## BlockDMatrixFree

Deallocates the block diagonal matrix structure.

### Syntax

```
IppStatus ippBlockDMatrixFree_16s(IppsBlockDMatrix_16s* pMatrix);
IppStatus ippBlockDMatrixFree_32f(IppsBlockDMatrix_32f* pMatrix);
IppStatus ippBlockDMatrixFree_64f(IppsBlockDMatrix_64f* pMatrix);
```

### Parameters

*pMatrix*                      Pointer to the block diagonal matrix.

### Description

The function `ippBlockDMatrixFree` is declared in the `ippsr.h` file. This function destroys the block diagonal matrix structure, and frees all memory associated with it.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pMatrix</i> pointer is <code>NULL</code> .

## NthMaxElement

Searches for the N-th maximal element of a vector.

### Syntax

```
IppStatus ippNthMaxElement_32s(const Ipp32s* pSrc, int len, int N, Ipp32s* pRes);
```

```
IppStatus ippsNthMaxElement_32f(const Ipp32f* pSrc, int len, int N, Ipp32f* pRes);
```

```
IppStatus ippsNthMaxElement_64f(const Ipp64f* pSrc, int len, int N, Ipp64f* pRes);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>len</i> ].
<i>len</i>	Number of elements in the input vector <i>pSrc</i> .
<i>N</i>	Rank of element to find.
<i>pRes</i>	Pointer to the value of <i>N</i> -th maximal element.

### Description

The function `ippsNthMaxElement` is declared in the `ippsr.h` file. This function finds the *N*-th maximal element of the input vector *pSrc*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates an error when <i>N</i> is less than 0, or greater than or equal to <i>len</i> .

## VecMatMul

*Multiplies a vector by a matrix.*

---

### Syntax

```
IppStatus ippsVecMatMul_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pMatr,
int step, int height, Ipp16s* pDst, int width, int scaleFactor);
```

```
IppStatus ippsVecMatMul_16s_D2LSfs(const Ipp16s* pSrc, const Ipp16s** mMatr,
int height, Ipp16s* pDst, int width, int scaleFactor);
```

```
IppStatus ippsVecMatMul_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMatr, int
step, int height, Ipp32f* pDst, int width);
```

```

IppStatus ippsVecMatMul_32f_D2L(const Ipp32f* pSrc, const Ipp32f** mMatr,
int height, Ipp32f* pDst, int width);

IppStatus ippsVecMatMul_16s32s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pMatr,
int step, int height, Ipp32s* pDst, int width, int scaleFactor);

IppStatus ippsVecMatMul_16s32s_D2LSfs(const Ipp16s* pSrc, const Ipp16s*
pMatr, int step, int height, Ipp32s* pDst, int width, int scaleFactor);

IppStatus ippsVecMatMul_32s_D2Sfs(const Ipp32s* pSrc, const Ipp32s* pMatr,
int step, int height, Ipp32s* pDst, int width, int scaleFactor);

IppStatus ippsVecMatMul_32s_D2LSfs(const Ipp32s* pSrc, const Ipp32s* pMatr,
int step, int height, Ipp32s* pDst, int width, int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> ].
<i>pMatr</i>	Pointer to the input matrix vector [ <i>height</i> * <i>step</i> ].
<i>mMatr</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in <i>pMatr</i> vector (in <i>pMatr</i> elements).
<i>width</i>	Length of the result vector and input matrix rows.
<i>height</i>	Number of rows in the input matrix.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsVecMatMul` is declared in the `ippsr.h` file. This function multiplies the given matrix by the input column vector as:

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot pMatr[i \cdot step + j]$$

for the function flavors with D2 suffix,  
and

$$pDst[j] = \sum_{i=0}^{height-1} pSrc[i] \cdot mMatr[i][j]$$

for the function flavors with D2L suffix,

where  $0 \leq j < width$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## MatVecMul

*Multiplies a matrix by a vector.*

---

### Syntax

```
IppStatus ippMatVecMul_16s_D2Sfs(const Ipp16s* pMatr, int step, const Ipp16s* pSrc, int width, Ipp16s* pDst, int height, int scaleFactor);
```

```
IppStatus ippMatVecMul_16s_D2LSfs(const Ipp16s** mMatr, const Ipp16s* pSrc, int width, Ipp16s* pDst, int height, int scaleFactor);
```

```
IppStatus ippMatVecMul_32f_D2(const Ipp32f* pMatr, int step, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
```

```
IppStatus ippMatVecMul_32f_D2L(const Ipp32f** mMatr, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
```

```
IppStatus ippMatVecMul_16s32s_D2Sfs(const Ipp16s* pMatr, int step, const Ipp16s* pSrc, int width, Ipp32s* pDst, int height, int scaleFactor);
```

```
IppStatus ippMatVecMul_16s32s_D2LSfs(const Ipp16s** mMatr, const Ipp16s* pSrc, int width, Ipp32s* pDst, int height, int scaleFactor);
```

```
IppStatus ippMatVecMul_32s_D2Sfs(const Ipp32s* pMatr, int step, const Ipp32s* pSrc, int width, Ipp32s* pDst, int height, int scaleFactor);
```

---

```
IppStatus ippsMatVecMul_32s_D2LSfs(const Ipp32s** mMatr, const Ipp32s* pSrc,
int width, Ipp32s* pDst, int height, int scaleFactor);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>width</i> ].
<i>pMatr</i>	Pointer to the input matrix vector [ <i>height</i> * <i>step</i> ].
<i>mMatr</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in <i>pMatr</i> vector (in <i>pMatr</i> elements).
<i>width</i>	Length of the input vector and input matrix rows.
<i>height</i>	Number of rows in the input matrix and the length of the result vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsMatVecMul` is declared in the `ippsr.h` file. This function multiplies the given matrix by the input column vector as:

for the function flavors with D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot pMatr[i \cdot step + j]$$

and

$$pDst[i] = \sum_{j=0}^{width-1} pSrc[j] \cdot mMatr[i][j]$$

for the function flavors with D2L suffix,

where  $0 \leq i < height$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> or <i>width</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## Feature Processing

This section describes functions that pre-process raw speech signals. Feature extraction is the first step in the recognition process. Speech features are a compressed form of the original speech signals. By extracting features, the problem dimension is reduced. To some extent, feature extraction normalizes speaker variations and environmental distortions. This section also includes functions needed to support both silence/speech detection and also some well-known analysis techniques on speech signals.

Many of these functions are used in the Intel® IPP Speech Recognition Samples. See *Aurora Encoder-Decoder Sample* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## ZeroMean

*Subtracts the mean value from the input vector.*

### Syntax

```
IppStatus ippsZeroMean_16s(Ipp16s* pSrcDst, int len);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector [ <i>len</i> ].
<i>len</i>	The number of elements in the vector.

### Description

The function `ippsZeroMean` is declared in the `ippsr.h` file. This function calculates the mean value of the *pSrcDst* vector and subtracts it from the vector *pSrcDst*. The resulting values are saturated if they exceed the range `[-32768..32767]`. The operations are as follows:

$$pSrcDst[i] = \max(-32768, \min(32767, pSrcDst[i] - \frac{1}{len} \sum_{j=0}^{len-1} pSrcDst[j]))$$

for  $0 \leq i < len$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## CompensateOffset

Removes the DC offset of the input signals.

### Syntax

```
IppStatus ippsCompensateOffset_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp32f val);
```

```
IppStatus ippsCompensateOffset_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f* pSrcDst0, Ipp32f dst0, Ipp32f val);
```

```
IppStatus ippsCompensateOffset_16s_I(Ipp16s* pSrcDst, int len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp32f val);
```

```
IppStatus ippsCompensateOffset_32f_I(Ipp32f* pSrcDst, int len, Ipp32f* pSrcDst0, Ipp32f dst0, Ipp32f val);
```

```
IppStatus ippsCompensateOffsetQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
```

```
IppStatus ippsCompensateOffsetQ15_16s_I(Ipp16s* pSrcDst, int len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector [ <code>len</code> ].
<code>pDst</code>	Pointer to the destination vector [ <code>len</code> ].

<i>pSrcDst</i>	Pointer to the source and destination vector [ <i>len</i> ] for in-place operations.
<i>pSrcDst0</i>	Pointer to the previous source element.
<i>len</i>	Number of elements in the vector.
<i>dst0</i>	Previous destination element.
<i>val</i>	Constant for offset compensation.
<i>valQ15</i>	Real-valued Q15 format constant for offset compensation ( $0.0_{Q31} \leq valQ15 < 1.0_{Q31}$ ).
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsCompensateOffset` is declared in the `ippsr.h` file. This function removes the offset of the input signals. The destination vector is calculated as follows.

For the not-inplace flavors of the `ippsCompensateOffset` function:

$$pDst[0] = pSrc[0] - pSrc[0] + val * dst0$$

$$pDst[i] = pSrc[i] - pSrc[i-1] + val * pDst[i-1], 1 \leq i \leq len - 1$$

$$pSrcDst0[0] = pSrc[len - 1],$$

and for in-place flavors of the `ippsCompensateOffset` function:

$$y = pSrcDst[0] - pSrcDst0[0] + val * dst0, x = pSrcDst[0], pSrcDst[0] = y,$$

$$y = pSrcDst[i] - pSrcDst[i-1] + val * x, x = pSrcDst[i], pSrcDst[i] = y,$$

$$1 \leq i \leq len - 1,$$

$$pSrcDst0[0] = x.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or when <i>valQ15</i> is out of range.



## SignChangeRate

Counts the zero-cross rate for the input signal.

### Syntax

```
IppStatus ippsSignChangeRate_16s(const Ipp16s* pSrc, int len, Ipp32s* pRes);
IppStatus ippsSignChangeRate_32f(const Ipp32f* pSrc, int len, Ipp32f* pRes);
IppStatus ippsSignChangeRate_Count0_16s (const Ipp16s* pSrc, int len, Ipp32s*
pRes);
IppStatus ippsSignChangeRate_Count0_32f (const Ipp32f* pSrc, int len, Ipp32f*
pRes);
IppStatus ippsSignChangeRateXor_32f(const Ipp32f* pSrc, int len, Ipp32f*
pRes);
```

### Parameters

<i>pSrc</i>	Pointer to the input signal [ <i>len</i> ].
<i>len</i>	Number of elements in the input signal <i>pSrc</i> .
<i>pRes</i>	Pointer to the result variable.

### Description

The function `ippsSignChangeRate` is declared in the `ippsr.h` file. This function counts the number of sign changes in the input signal and can be used to detect speech in continuous speech input. The operations are as follows:

For the function `ippsSignChangeRate`,

$$pRes[0] = \sum_{i=1}^{len-1} \begin{cases} 1, & \text{if } pSrc[i] \cdot pSrc[i-1] < 0 \\ 0, & \text{otherwise} \end{cases}$$

For the function `ippsSignChangeRateCount0`,

$$pRes[0] = \frac{1}{2} \sum_{i=1}^{len-1} |sign(pSrc[i]) - sign(pSrc[i-1])|, \quad sign(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

For the function `ippsSignChangeRateXor`,

$$pRes[0] = \sum_{i=0}^{len-2} sign(pSrc[i]) \oplus sign(pSrc[i+1]), \text{ where } sign(x) = \begin{cases} 0, & x > 0, x = +0 \\ 1, & x < 0, x = -0 \end{cases}$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## LinearPrediction

*Performs linear prediction analysis on the input vector.*

---

### Syntax

```
ippStatus ippsLinearPrediction_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
Ipp16s* pDst, int lenDst, int scaleFactor);
```

```
ippStatus ippsLinearPrediction_Auto_32f(const Ipp32f* pSrc, int lenSrc,
Ipp32f* pDst, int lenDst);
```

```
ippStatus ippsLinearPredictionNeg_Auto_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
Ipp16s* pDst, int lenDst, int scaleFactor);
```

```
ippStatus ippsLinearPredictionNeg_Auto_32f(const Ipp32f* pSrc, int lenSrc,
Ipp32f* pDst, int lenDst);
```

```
ippStatus ippsLinearPrediction_Cov_16s_Sfs(const Ipp16s* pSrc, int lenSrc,
Ipp16s* pDst, int lenDst, int scaleFactor);
```

```
IppStatus ippsLinearPrediction_Cov_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
pDst, int lenDst);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>lenSrc</i> ].
<i>lenSrc</i>	Length of the input vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the output LPC coefficients vector [ <i>lenDst</i> ].
<i>lenDst</i>	Length of the output vector <i>pDst</i> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsLinearPrediction` is declared in the `ippsr.h` file. This function performs linear prediction analysis on the input signal. For functions with the `Auto` suffix, the LPC coefficients are calculated by solving the following equations that represent the autocorrelation approach:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = r[k] ,$$

$k = 1, \dots, lenDst,$

where

$$r[k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k] .$$

.

For functions with the `Neg_Auto` suffix, the LPC coefficients are calculated by solving the similar equations with inverted sign of the right-hand side:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot r[|i-k|] = -r[k] ,$$

$k = 1 \dots lenDst,$

For functions with the `Cov` suffix, the LPC coefficients are calculated by solving the following equations that represent the covariance approach:

$$\sum_{i=1}^{lenDst} pDst[i-1] \cdot c[i][k] = c[0][k] ,$$

$k = 1, \dots, lenDst,$

where

$$c[i][k] = \sum_{j=0}^{lenSrc-k-1} pSrc[j] \cdot pSrc[j+k-i] .$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>lenSrc</code> or <code>lenDst</code> is less than or equal to 0, or <code>lenDst</code> is greater or equal than <code>lenSrc</code> .
<code>ippStsNoOperation</code>	Indicates no solution to the LPC problem.

## Durbin

*Performs Durbin's recursion on an input vector of autocorrelations.*

---

### Syntax

```
IppStatus ippsDurbin_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp32f* pErr, int scaleFactor);

IppStatus ippsDurbin_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f*
pErr);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>len</i> +1].
<i>pDst</i>	Pointer to the output LPC coefficients vector [ <i>len</i> ] .
<i>len</i>	Length of the output vector.
<i>pErr</i>	Pointer to the result prediction error.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsDurbin` is declared in the `ippsr.h` file. This function performs the Durbin's recursion on the input autocorrelation vector and calculates the linear prediction coefficients and the prediction error as follows:

1) Initialization:

$$m = \text{len}, R_j = pSrc[j], j = 0, \dots, m, E^0 = R_0$$

2) Iterations for  $i = 1, \dots, m$  :

$$k_i = \left( R_i - \sum_{j=1}^{i-1} y_j^{i-1} \cdot R_{i-j} \right) / E^{i-1}, \quad E^i = (1 - k_i^2) \cdot E^{i-1},$$

$$y_i^i = k_i, \quad y_j^i = y_j^{i-1} - k_i \cdot y_{i-j}^{i-1},$$

$j = 1, \dots, i - 1$

3) Result:

$pDst[i - 1] = y_i^m, i = 1, \dots, m, pErr[i0] = E^m$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsNoOperation</code>	Indicates no solution to the LPC problem ( $E^i \leq 0$ ).

## Schur

*Calculates reflection coefficients using Schur algorithm.*

---

### Syntax

```
IppStatus ippsSchur_16s_sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, Ipp32f*
pErr, int scaleFactor);

IppStatus ippsSchur_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f*
pErr);
```

### Parameters

<code>pSrc</code>	Pointer to the input autocorrelations vector.
<code>pDst</code>	Pointer to the output reflection coefficients vector.
<code>len</code>	Length of the output vector.
<code>pErr</code>	Pointer to the result prediction error.
<code>scaleFactor</code>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsSchur` is declared in the `ippsr.h` file. This function uses Schur algorithm to compute reflection coefficients in the following steps:

1) Initialization:

$$b_1^0 = pSrc[0], \quad a_j^0 = b_{j+1}^0 = pSrc[j], \quad \text{for } j = 1, \dots, len-1, \quad a_{len}^0 = pSrc[len]$$

2) Iterations for  $i = 1, \dots, len$ :

$$k_i = -a_i^{i-1} / b_i^{i-1}$$

$$a_j^i = a_j^{i-1} + k_i b_j^{i-1}, \quad b_j^i = b_{j-1}^{i-1} + k_i a_{j-1}^{i-1}, \quad j = i+1, \dots, len$$

3) Result:

$$pDst[i-1] = k_i, \quad i = 1, \dots, len$$

$$pErr[0] = b_{len}^{len-1} + k_{len} \cdot a_{len}^{len-1}.$$

## Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippsStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippsStsNoOperation</code>	Indicates no solution to the LPC problem ( $b_i^i = 0$ ).

## LPToSpectrum

*Calculates smoothed magnitude spectrum.*

---

### Syntax

```
IppStatus ippsLPToSpectrum_16s_Sfs(const Ipp16s* pSrc, int len, Ipp16s* pDst,
int order, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLPToSpectrum_32f(const Ipp32f* pSrc, int len, Ipp32f* pDst,
int order, Ipp32f val);
```

### Parameters

<i>pSrc</i>	Pointer to the input LPC coefficients vector .
<i>pDst</i>	Pointer to the output LP spectrum coefficients vector.
<i>len</i>	Number of LPC coefficients.
<i>order</i>	FFT order for spectrum calculation.
<i>val</i>	The value to add to spectrum.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsLPToSpectrum` is declared in the `ippsr.h` file. This function computes the first half of a linear prediction magnitude spectrum as follows:

$$pDst[k] = \frac{1}{\left| val - \sum_{i=1}^{len} pSrc[i-1] \cdot e^{-\frac{jik\pi}{N}} \right|},$$

$k = 0, \dots, N-1, N = 2^{order}$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcDst</i> pointer is NULL.



<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0, or when it is greater or equal to $2^{order+1}$ .
<code>ippStsFftOrderErr</code>	Indicates an error when the <code>order</code> value is incorrect.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>+Inf</code> , and for integer operations is set to <code>IPP_MAX_16S</code> .

## LPToCepstrum

*Calculates cepstrum coefficients from linear prediction coefficients.*

### Syntax

```
IppStatus ippsLPToCepstrum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsLPToCepstrum_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

### Parameters

<code>pSrc</code>	Pointer to the linear prediction coefficients.
<code>pDst</code>	Pointer to the cepstrum coefficients .
<code>len</code>	Number of elements in the source and destination vectors.
<code>scaleFactor</code>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsLPToCepstrum` is declared in the `ippsr.h` file. This function calculates the cepstrum coefficients from the linear prediction coefficients according to the following formula:

$$pDst[k] = - \left( pSrc[k] + \frac{1}{k+1} \sum_{i=1}^k (k-i+1) \cdot pSrc[i-1] \cdot pDst[k-i] \right),$$

$k = 0 \dots len - 1$  .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## CepstrumToLP

*Calculates linear prediction coefficients from cepstrum coefficients.*

---

### Syntax

```

IppStatus ippsCepstrumToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
len, int scaleFactor);

IppStatus ippsCepstrumToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);

```

### Parameters

<code>pSrc</code>	Pointer to the cepstrum coefficients.
<code>pDst</code>	Pointer to the linear prediction coefficients.
<code>len</code>	Number of elements in the source and destination vectors.
<code>scaleFactor</code>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsCepstrumToLP` is declared in the `ippsr.h` file. This function calculates the linear prediction coefficients from the cepstrum coefficients according to the following formula:

$$pDst[k] = \left( pSrc[k] + \frac{1}{k+1} \sum_{i=1}^k (k-i+1) \cdot pSrc[k-i] \cdot pDst[i-1] \right),$$

$k = 0 \dots len - 1$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## LPToReflection

*Calculates the linear prediction reflection coefficients from the linear prediction coefficients.*

### Syntax

```

IppStatus ippsLPToReflection_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int
len, int scaleFactor);

IppStatus ippsLPToReflection_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);

```

### Parameters

<i>pSrc</i>	Pointer to the linear prediction coefficients.
<i>pDst</i>	Pointer to the linear prediction reflection coefficients.
<i>len</i>	Number of elements in the source and destination vectors.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsLPToReflection` is declared in the `ippsr.h` file. This function calculates the linear prediction reflection coefficients according to the following formulae:

1) Initialization:

$$n = len, a_i^n = pSrc[i - 1], i = 1, \dots, n$$

2) Iterations for  $i = m, \dots, 1$  :

$$k_i = a_i^i, a_j^{j-1} = (a_j^i - a_i^i * a_{i-j}^i) / (1 - k_i^2),$$

$$j = 1, \dots, i-1$$

3) Result:

$$pDst[i-1] = k_i, i = 1, \dots, n.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsNoOperation</code>	Indicates that reflection coefficients could not be calculated (that is, $ k_i  = 1$ for one of iterations).

## ReflectionToLP

*Calculates the linear prediction coefficients from the linear prediction reflection coefficients.*

---

### Syntax

```
IppStatus ippReflectionToLP_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippReflectionToLP_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the linear prediction reflection coefficients.
<i>pDst</i>	Pointer to the linear prediction coefficients.
<i>len</i>	Number of elements in the source and destination vectors.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippReflectionToLP` is declared in the `ippsr.h` file. This function calculates the linear prediction reflection coefficients from the linear prediction coefficients according to the following formulas:

1) Initialization:

$$n = len, k_i = pSrc[i-1], i = 1, \dots, n$$

2) Iterations for  $i = 1, \dots, m$ :

$$a_i^i = k_i, a_j^i = a_j^{j-1} - k_i * a_{i-j}^{i-1},$$

$$j = 1, \dots, i-1$$

3) Result:

$$pDst[i-1] = a_i^n, i = 1, \dots, n.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## ReflectionToAR, ReflectionToLAR, ReflectionToTrueAR

Converts reflection coefficients to area ratios.

### Syntax

```

IppStatus ippsReflectionToAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsReflectionToAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);

IppStatus ippsReflectionToLAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
Ipp16s* pDst, int len, Ipp32f val, int scaleFactor);

IppStatus ippsReflectionToLAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f val);

IppStatus ippsReflectionToTrueAR_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal,
Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsReflectionToTrueAR_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
len);

```

### Parameters

<code>pSrc</code>	Pointer to the input vector.
<code>pDst</code>	Pointer to the destination vector.
<code>len</code>	Length of the input and output vectors.
<code>val</code>	Threshold value in the range (0.. 1).
<code>srcShiftVal</code>	Scale factor used in the function flavors <code>ippsReflectionToLAR</code> only. Refer to <a href="#">Integer Scaling</a> .
<code>scaleFactor</code>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

These functions are declared in the `ippsr.h` file. The functions `ippsReflectionToAR`, and calculates approximate area ratios using the following formula:

$$pDst[k] = \frac{1 - pSrc[k]}{1 + pSrc[k]} ,$$

$k = 0, \dots, len - 1$

The function `ippsReflectionToLAR` calculates logarithm of area ratios as given by:

$$pDst[k] = \begin{cases} \ln \frac{1 - val}{1 + val} , & \text{if } pSrc[k] \geq val \\ \ln \frac{1 - pSrc[k]}{1 + pSrc[k]} , & \text{if } val > pSrc[k] > -val , \\ \ln \frac{1 + val}{1 - val} , & \text{if } -val \geq pSrc[k] \end{cases}$$

$k = 0, \dots, len - 1$  .

The function `ippsReflectionToTrueAR` calculates area ratios using the formulas:

$$pDst[0] = \frac{1 - pSrc[0]}{1 + pSrc[0]} , \quad pDst[k] = pDst[k-1] \cdot \frac{1 - pSrc[k]}{1 + pSrc[k]} ,$$

$k = 1, \dots, len - 1$  .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsRangeErr</code>	Indicates an error when <code>val</code> is out of the range (0..1).

`ippStsDivByZero` Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to: *NaN*, if the dividend vector element has zero value; *+Inf*, if the dividend vector element has positive value; *-Inf*, if the dividend vector element is negative.

The value of the destination vector element for integer operations is set to:

`IPP_MAX_16S`, if the dividend vector element is positive;  
`IPP_MIN_16S`, if the dividend vector element is negative.

## ReflectionToTilt

Calculates tilt for rise/fall/connection parameters.

### Syntax

```
IppStatus ippsReflectionToAbsTilt_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsReflectionToAbsTilt_32f(const Ipp32f* pSrc1, const Ipp32f*
pSrc2, Ipp32f* pDst, int len);

IppStatus ippsReflectionToTilt_16s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsReflectionToTilt_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
Ipp32f* pDst, int len);
```

### Parameters

<code>pSrc1</code>	Pointer to the first input vector [ <i>len</i> ].
<code>pSrc2</code>	Pointer to the second input vector [ <i>len</i> ].
<code>pDst</code>	Pointer to the destination vector [ <i>len</i> ].
<code>len</code>	Length of the input and output vectors.
<code>scaleFactor</code>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

These functions are declared in the `ippsr.h` file. The function `ippsReflectionToAbsTilt` converts rise and fall coefficients to absolute tilt as given by:

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{|pSrc1[k]| + |pSrc2[k]|} ,$$

$k = 0, \dots, len - 1$  .

The function `ippsReflectionToTilt` converts rise and fall coefficients to tilt:

$$pDst[k] = \frac{|pSrc1[k]| - |pSrc2[k]|}{pSrc1[k] + pSrc2[k]} ,$$

$k = 0, \dots, len - 1$  .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to: <code>NaN</code> , if the dividend vector element has zero value; <code>+Inf</code> , if the dividend vector element has positive value; <code>-Inf</code> , if the dividend vector element is negative. The value of the destination vector element for integer operations is set to: <code>IPP_MAX_16S</code> , if the dividend vector element is positive; <code>IPP_MIN_16S</code> , if the dividend vector element is negative.

## PitchmarkToF0Cand

*Calculates rise and fall amplitude and duration for tilt.*

---

### Syntax

```
ippStatus ippsPitchmarkToF0Cand_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
int len, int scaleFactor);
```



```
IppStatus ippsPitchmarkToF0Cand_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Length of the input and output vectors.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsPitchmarkToF0Cand` is declared in the `ippsr.h` file. This function calculates F0 candidate values from pitchmarks using the following formula:

$$pDst[0] = 1/pSrc[0],$$

$$pDst[k] = 1/(pSrc[k] - pSrc[k - 1]), k = 1, \dots, len - 1.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>+Inf</code> . The value of the destination vector element for integer operations is set to <code>IPP_MAX_16S</code> .

## UnitCurve

*Calculates tilt for rise and fall coefficients.*

---

### Syntax

```
IppStatus ippsUnitCurve_16s_Sfs(const Ipp16s* pSrc, int srcShiftVal, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsUnitCurve_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
```

```
IppStatus ippsUnitCurve_16s_ISfs(Ipp16s* pSrcDst, int srcShiftVal, int len,
int scaleFactor);
```

```
IppStatus ippsUnitCurve_32f_I(Ipp32f* pSrcDst, int len);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector.
<i>pSrcDst</i>	Pointer to the input and destination vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Length of the input and output vectors.
<i>srcShiftVal</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsUnitCurve` is declared in the `ippsr.h` file. This function calculates the destination vector according to the following formula:

$$pDst[k] = \begin{cases} 0, & \text{if } pSrc[k] < 0 \\ pSrc[k]^2, & \text{if } 0 \leq pSrc[k] < 1 \\ 2 - (2 - pSrc[k])^2, & \text{if } 1 \leq pSrc[k] \leq 2 \\ 2, & \text{if } 2 < pSrc[k] \end{cases},$$

for  $k = 0, \dots, len - 1$ .

In-place function flavors overwrite source vector with destination data.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## LPToLSP

*Calculates line spectrum pairs vector from linear prediction coefficients.*

---

### Syntax

```
IppStatus ippsLPToLSP_16s_Sfs(const Ipp16s* pSrcLP, int srcShiftVal, Ipp16s*
pDstLSP, int len, int* nRoots, int nInt, int nDiv, int scaleFactor);

IppStatus ippsLPToLSP_32f(const Ipp32f* pSrcLP, Ipp32f* pDstLSP, int len,
int* nRoots, int nInt, int nDiv);
```

### Parameters

<i>pSrcLP</i>	Pointer to the input LP coefficients vector.
<i>pDstLSP</i>	Pointer to the output LSP coefficients vector .
<i>len</i>	Number of LP coefficients.
<i>nRoots</i>	Pointer to the number of LSP values found.
<i>nInt</i>	Number of intervals in zero crossing search.
<i>nDiv</i>	Number of interval bisections during search for roots.
<i>srcShiftVal</i>	Scale factor for <i>pSrcLP</i> values.
<i>scaleFactor</i>	Scale factor for <i>pDstLSP</i> values, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsLPToLSP` is declared in the `ippsr.h` file. This function converts linear prediction (LP) coefficients to the line spectrum pair (LSP) representation and implements the algorithm generally used in speech coding.

The LST coefficients are written to *pDstLSP* vector in ascending order.

The number of LST coefficients found is written to *nRoots*[0]. If all *len* coefficients are not found, the error status is returned.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0, or `nInt` is less than `len`, or `nDiv` is less than 0.

`ippStsNoRootFoundErr` Indicates that less than `len` roots were found.

## LSPToLP

*Converts line spectrum pairs vector to linear prediction coefficients.*

---

### Syntax

```
IppStatus ippsLSPToLP_16s_Sfs(const Ipp16s* pSrcLSP, int srcShiftVal, Ipp16s* pDstLP, int len, int scaleFactor);
```

```
IppStatus ippsLSPToLP_32f(const Ipp32f* pSrcLSP, Ipp32f* pDstLP, int len);
```

### Parameters

`pSrcLSP` Pointer to the input LSP coefficients vector.

`pDstLP` Pointer to the output LP coefficients vector .

`len` Number of LSP and LP coefficients.

`srcShiftVal` Scale factor for `pSrcLSP` values.

`scaleFactor` Scale factor for `pDstLP` values, refer to [Integer Scaling](#).

### Description

The function `ippsLSPToLP` is declared in the `ippsr.h` file. This function converts line spectrum pair (LSP) values to linear prediction (LP) coefficients using the Kabal's method (see [[Kab86](#)]).

The polynomials

$$G_1(e^{-jw}) = e^{-jwm_1} \cdot G_1'(w) \quad \text{and} \quad G_2(e^{-jw}) = e^{-jwm_2} \cdot G_2'(w)$$

are reconstructed from their roots `pSrcLSP[k]`,  $k = 0, \dots, len-1$ , using Chebyshev representation.

Next the polynomials  $F_1(z)$  and  $F_2(z)$  and are reconstructed.

Finally, the linear prediction coefficients are determined from

$$A(z) = \frac{F_1(z) + F_2(z)}{2} = 1 - \sum_{k=0}^{len-1} a_k z^{-k}$$

and

$pDstLSP[k] = a_k, k = 0, \dots, len-1$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsIncorrectLSP</code>	Indicates a warning that <code>pSrcLSP</code> elements are not valid LSP values (so that condition $-1 < \dots < pSrcLSP[k+1] < pSrcLSP[k] < \dots < 1$ is not met).

## MelToLinear

Converts Mel-scaled values to linear scale values.

### Syntax

```
IppStatus ippMelToLinear_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f melMul, Ipp32f melDiv);
```

### Parameters

<code>pSrc</code>	Pointer to the input vector.
<code>pDst</code>	Pointer to the output vector .
<code>len</code>	Length of input and output vectors.
<code>melMul</code>	Multiply factor in the Mel-scale equation.
<code>melDiv</code>	Divide factor in the Mel-scale equation.

### Description

The function `ippMelToLinear` is declared in the `ippsr.h` file. This function converts the Mel-frequency scale to the linear frequency scale:

$$pDst[i] = melDiv \cdot \left[ \exp\left(\frac{pSrc[i]}{melMul}\right) - 1 \right] ,$$

$i = 0, \dots, len-1$  .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0, or <code>melMul</code> or <code>melDiv</code> is equal to 0.

## LinearToMel

Converts linear-scale values to Mel-scale values.

### Syntax

```
IppStatus ippLinearToMel_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len,
Ipp32f melMul, Ipp32f melDiv);
```

### Parameters

<code>pSrc</code>	Pointer to the input vector.
<code>pDst</code>	Pointer to the output vector.
<code>len</code>	Length of the input and output vectors.
<code>melMul</code>	Multiply factor in the Mel-scale equation.
<code>melDiv</code>	Divide factor in the Mel-scale equation.

### Description

The function `ippLinearToMel` is declared in the `ippsr.h` file. This function converts the linear frequency scale to the Mel-frequency scale:

$$pDst[i] = melMul \cdot \ln\left(1 + \frac{pSrc[i]}{melDiv}\right) ,$$

$i = 0, \dots, len-1$  .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <i>NULL</i> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or <i>melMul</i> or <i>melDiv</i> is equal to 0.

## CopyWithPadding

*Copies the input signal to the output with zero-padding.*

---

### Syntax

```
IppStatus ippCopyWithPadding_16s(const Ipp16s* pSrc, int lenSrc, Ipp16s*
pDst, int lenDst);
```

```
IppStatus ippCopyWithPadding_32f(const Ipp32f* pSrc, int lenSrc, Ipp32f*
pDst, int lenDst);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector.
<i>pDst</i>	Pointer to the output vector .
<i>lenSrc</i>	Length of the input vector.
<i>lenDst</i>	Length of the output vector.

### Description

The function `ippCopyWithPadding` is declared in the `ippsr.h` file. This function copies the input vector to the output vector. If the length of the output vector is bigger, the trailing elements are padded with zeroes.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <i>NULL</i> .
<code>ippStsSizeErr</code>	Indicates an error when <i>lenSrc</i> or <i>lenDst</i> is less than or equal to 0, or when <i>lenDst</i> is less than <i>lenSrc</i> .

## MelFBankGetSize

*Gets the size of the Mel-frequency filter bank structure.*

---

### Syntax

```
IppStatus ippsMelFBankGetSize_32s(int winSize, int nFilter, IppMelMode mode,
int* pSize);
```

### Parameters

<i>winSize</i>	Frame length (samples) in the range [32 ..8192].
<i>nFilter</i>	Number of Mel-scale filter banks K ( $0 < nFilter \leq winSize$ ).
<i>mode</i>	Flag that determines the execution mode. Currently only IPP_FBANK_FREQWGT is supported.
<i>pSize</i>	Pointer to the variable to contain the size of the filter bank structure.

### Description

The function `ippsMelFBankGetSize` is declared in the `ippsr.h` file. This function determines the size required for the Mel-frequency filter bank structure and associated storage. It should be called before memory allocation and before the call to `ippsMelFBankInit`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is NULL.
<code>ippStsFBankFlagErr</code>	Indicates an error when the <i>mode</i> value is incorrect.
<code>ippStsSizeErr</code>	Indicates an error when <i>nFilter</i> is out of range or when <i>winSize</i> is less than or equal to 0.



## MelBankInit

*Initializes the structure for performing the Mel-frequency filter bank analysis.*

---

### Syntax

```
IppStatus ippsMelFBankInit_32s(IppsFBankState_32s* pFBank, int* pFFTLen, int
winSize, Ipp32s sampFreq, Ipp32s lowFreq, Ipp32s highFreq, int nFilter,
Ipp32s melMulQ15, Ipp32s melDivQ15, IppMelMode mode);
```

### Parameters

<i>pFBank</i>	Pointer to the Mel-scale filter bank structure to be initialized.
<i>pFFTLen</i>	Pointer to the length of FFT ( $N = pFFTLen[0]$ ) used for the filter bank evaluation.
<i>winSize</i>	Frame length (in samples).
<i>sampFreq</i>	Input signal sampling frequency $f_i$ (in Hz).
<i>lowFreq</i>	Start frequency $f_{low}$ of the first band-pass filter (in Hz).
<i>highFreq</i>	End frequency $f_{high}$ of the last band-pass filter (in Hz).
<i>nFilter</i>	Number of Mel-scale filter banks K.
<i>melMulQ15</i>	Real-valued mel-scale formula multiply factor in Q15.
<i>melDivQ15</i>	Real-valued mel-scale formula divisor in Q15.
<i>mode</i>	Flags that determine execution mode; can have the following values: <code>IPP_FBANK_MELWGT</code> – the function calculates filter bank weights in Mel-scale; <code>IPP_FBANK_FREQWGT</code> – the function calculates filter bank weights in the frequency space. One of the above two flags should necessarily be set. <code>IPP_POWER_SPECTRUM</code> – indicates that the FFT power spectrum is used during the filter bank analysis.

## Description

The function `ippsMelFBankInit` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis function. Mel filter coefficients are calculated in the same way as for the `ippsMelFBankInitAlloc` functions. Memory for the structure should be previously allocated, the size of this memory can be determined by the `ippsMelFBankGetSize` function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>winSize</code> , <code>nFilter</code> , <code>sampFreq</code> , or <code>lowFreq</code> is less than or equal to 0.
<code>ippStsFBankFreqErr</code>	Indicates an error when <code>highFreq</code> is less than <code>lowFreq</code> or <code>highFreq</code> is greater than <code>sampFreq/2</code> .
<code>ippStsFBankFlagErr</code>	Indicates an error when the <code>mode</code> value is incorrect.

## MelFBankInitAlloc

*Allocates memory and initializes the structure for performing the Mel-frequency filter bank analysis.*

### Syntax

```
IppStatus ippsMelFBankInitAlloc_16s(IppsFBankState_16s** pFBank, int* pFFTLen,
int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f highFreq, int nFilter,
Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

```
IppStatus ippsMelFBankInitAlloc_32f(IppsFBankState_32f** pFBank, int* pFFTLen,
int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f highFreq, int nFilter,
Ipp32f melMul, Ipp32f melDiv, IppMelMode mode);
```

### Parameters

<code>pFBank</code>	Pointer to the Mel-scale filter bank structure to be initialized.
<code>pFFTLen</code>	Pointer to the length of FFT ( <code>N = pFFTLen [0]</code> ) used for the filter bank evaluation.
<code>winSize</code>	Frame length (in samples).

<i>sampFreq</i>	Input signal sampling frequency $f_i$ (in Hz).
<i>lowFreq</i>	Start frequency $f_{low}$ of the first band-pass filter (in Hz).
<i>highFreq</i>	End frequency $f_{high}$ of the last band-pass filter (in Hz).
<i>nFilter</i>	Number of Mel-scale filter banks $K$ .
<i>melMul</i>	Mel-scale formula multiply factor.
<i>melDiv</i>	Mel-scale formula divisor.
<i>mode</i>	Flags that determine execution mode; can have the following values: <code>IPP_FBANK_MELWGT</code> – the function calculates filter bank weights in Mel-scale; <code>IPP_FBANK_FREQWGT</code> – the function calculates filter bank weights in the frequency space. One of the above two flags should necessarily be set. <code>IPP_POWER_SPECTRUM</code> – indicates that the FFT power spectrum is used during the filter bank analysis.

## Description

The function `ippsMelFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis. The filter bank analysis is one of the major steps in the Mel-Frequency Cepstrum Coefficients (MFCC) feature calculation.

If *mode* is set to `IPP_FBANK_FREQWGT`, then the filter outputs is calculated (by the function `ippsEvalFBank`) according to the following formula:

$$fBank_{k-1} = \sum_{i=y_{k-1}}^{y_k} \frac{i - y_{k-1} + 1}{y_k - y_{k-1} + 1} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{y_{k+1} - i + 1}{y_{k+1} - y_k + 1} \cdot x_i ,$$

for  $k = 1 \dots K$  , (8.1)

where  $x_i$  is the magnitude of the corresponding FFT spectrums.

If *mode* is set to `IPP_FBANK_MELWGT`, the filter outputs is calculated (by the function `ippsEvalFBank`) according to the formula:

$$fFank_{k-1} = \sum_{i=y_{k-1}+1}^{y_k} \frac{\text{mel}\left(i \frac{f_s}{N}\right) - c_{k-1}}{c_k - c_{k-1}} \cdot x_i + \sum_{i=y_k+1}^{y_{k+1}} \frac{c_{k+1} - \text{mel}\left(i \frac{f_s}{N}\right)}{c_{k+1} - c_k} \cdot x_i,$$

$k = 1 \dots K$  (8.2)

For the function flavor `ippsMelFBankInitAlloc_32s` currently only `IPP_FBANK_MELWGT` mode is supported.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>winSize</code> , <code>nFilter</code> , <code>sampFreq</code> , or <code>lowFreq</code> is less than or equal to 0.
<code>ippStsFBankFreqErr</code>	Indicates an error when <code>highFreq</code> is less than <code>lowFreq</code> or <code>highFreq</code> is greater than <code>sampFreq/2</code> .
<code>ippStsFBankFlagErr</code>	Indicates an error when the <code>mode</code> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.

## MelLinFBankInitAlloc

*Initializes the structure for performing a combined linear and Mel-frequency filter bank analysis.*

---

### Syntax

```
IppStatus ippsMelLinFBankInitAlloc_16s(IppsFBankState_16s** pFBank, int*
pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f highFreq, int
nFilter, Ipp32f highLinFreq, int nLinFilter, Ipp32f melMul, Ipp32f melDiv,
IppMelMode mode);
```

```
IppStatus ippsMelLinFBankInitAlloc_32f(IppsFBankState_32f** pFBank, int*
pFFTLen, int winSize, Ipp32f sampFreq, Ipp32f lowFreq, Ipp32f highFreq, int
nFilter, Ipp32f highLinFreq, int nLinFilter, Ipp32f melMul, Ipp32f melDiv,
IppMelMode mode);
```

## Parameters

<i>pFBank</i>	Pointer to the Mel-scale filter bank structure to be initialised.
<i>pFFTLen</i>	Pointer to the length of FFT ( $N = pFFTLen[0]$ ) used for the filter bank evaluation.
<i>winSize</i>	Frame length (in samples).
<i>sampFreq</i>	Input signal sampling frequency $f_i$ (in Hz).
<i>lowFreq</i>	Start frequency $f_{low}$ of the first band-pass filter (in Hz).
<i>highLinFreq</i>	End frequency $f_{linw}$ of the last band-pass filter in linear scale (in Hz).
<i>highFreq</i>	End frequency $f_{high}$ of the last band-pass filter (in Hz).
<i>nFilter</i>	Number of Mel-scale filter banks $K$ .
<i>nFilter</i>	Number of linear-scale filter banks $L$ .
<i>melMul</i>	Mel-scale formula multiply factor.
<i>melDiv</i>	Mel-scale formula divisor.
<i>mode</i>	Flags that determine execution mode; can have the following values: <code>IPP_FBANK_MELWGT</code> – the function calculates filter bank weights in Mel-scale; <code>IPP_FBANK_FREQWGT</code> – the function calculates filter bank weights in the frequency space. One of the above two flags should necessarily be set. <code>IPP_POWER_SPECTRUM</code> – indicates that the FFT power spectrum is used during the filter bank analysis.

## Description

The function `ippsMelLinFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the linear and Mel-frequency triangular filter banks. The first  $L$  filters are placed linearly on the frequency band from  $f_{low}$  to  $f_{lin}$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

<code>ippStsSizeErr</code>	Indicates an error when <code>winSize</code> , <code>nFilter</code> , <code>sampFreq</code> , <code>lowFreq</code> is less than or equal to 0, or when <code>nLinFilter</code> is less than or equal to 1, or <code>nLinFilter</code> is greater than <code>nFilter</code> .
<code>ippStsFBankFreqErr</code>	Indicates an error when <code>highLinFreq</code> is less than <code>lowFreq</code> , or <code>highFreq</code> is less than <code>highLinFreq</code> , or <code>highFreq</code> is greater than <code>sampFreq/2</code> , or <code>(highLinFreq &gt; lowFreq)&amp;&amp;(nLinFilter==0)</code> , or <code>(highLinFreq &lt; highFreq)&amp;&amp;(nLinFilter==nFilter)</code> .
<code>ippStsFBankFlagErr</code>	Indicates an error when the <code>mode</code> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.

## EmptyFBankInitAlloc

*Initializes an empty filter bank structure.*

---

### Syntax

```
IppStatus ippsEmptyFBankInitAlloc_16s(IppsFBankState_16s** pFBank, int*
pFFFTLen, int winSize, int nFilter, IppMelMode mode);

IppStatus ippsEmptyFBankInitAlloc_32f(IppsFBankState_32f** pFBank, int*
pFFFTLen, int winSize, int nFilter, IppMelMode mode);
```

### Parameters

<code>pFBank</code>	Pointer to the filter bank structure to be created.
<code>pFFFTLen</code>	Pointer to the length of FFT ( <code>N = pFFFTLen [0]</code> ) used for the filter bank evaluation.
<code>winSize</code>	Frame length (in samples).
<code>nFilter</code>	Number of filters banks $K$ .
<code>mode</code>	Flag determining the function's execution mode; can have the following value: <code>IPP_POWER_SPECTRUM</code> – indicates that the FFT power spectrum is used during the filter bank analysis.

## Description

The function `ippsEmptyFBankInitAlloc` is declared in the `ippsr.h` file. This function initializes the filter bank structure for the *nFilter* filters. The filter bank center frequencies can be set by the function `ippsFBankSetCenters` and the filter weights can be set by the function `ippsFBankSetCoeffs`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>winSize</i> or <i>nFilter</i> or <i>pFBank</i> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.

## FBankFree

Destroys the structure for the filter bank analysis.

## Syntax

```
IppStatus ippsFBankFree_16s(IppsFBankState_16s* pFBank);
IppStatus ippsFBankFree_32f(IppsFBankState_32f* pFBank);
```

## Parameters

*pFBank*                      Pointer to the filter bank structure.

## Description

The function `ippsFBankFree` is declared in the `ippsr.h` file. This function destroys the filter bank structure and frees all memory associated with it.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pFBank</i> pointer is <code>NULL</code> .

## FBankGetCenters

*Retrieves the center frequencies of the triangular filter banks.*

---

### Syntax

```
IppStatus ippsFBankGetCenters_16s(const IppsFBankState_16s* pFBank, int* pCenters);  
  
IppStatus ippsFBankGetCenters_32f(const IppsFBankState_32f* pFBank, int* pCenters);
```

### Parameters

<i>pFBank</i>	Pointer to the filter bank structure.
<i>pCenters</i>	Pointer to the output vector that contains the center frequencies.

### Description

The function `ippsGetCenters` is declared in the `ippsr.h` file. This function retrieves the indexes  $y_k$  (in FFT domain points) of the filter bank centers. The resulting array *pCenters* is of length *nFilter*+2. The filter bank structure *pFBank* must be initialized by either `ippsMelFBankInitAlloc` or `ippsMelLinFBankInitAlloc` function.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFBankErr</code>	Indicates an error when filter centers are not valid after filter bank initialization by <code>ippsEmptyFBankInitAlloc</code> function.

## FBankSetCenters

*Sets the center frequencies of the triangular filter banks.*

---

### Syntax

```
IppStatus ippsFBankSetCenters_16s(IppsFBankState_16s* pFBank, const int* pCenters);
```



```
IppStatus ippsFBankSetCenters_32f(IppsFBankState_32f* pFBank, const int*
pCenters);
```

### Parameters

*pFBank*                      Pointer to the filter bank structure.  
*pCenters*                  Pointer to the vector that contains center frequencies.

### Description

The function `ippsSetCenters` is declared in the `ippsr.h` file. This function sets the filter center indexes  $y_k$  in *pFBank*. The indexes must meet the following conditions:

$$0 \leq y_k \leq y_{k+1} \leq \dots \leq N/2, i = 0, \dots, K.$$

If the  $k$ -th center is modified, the filter weight coefficients in the  $(k-1)$ -th,  $k$ -th and  $(k+1)$ -th filter banks must be also modified using the `ippsFBankSetCoeffs` function.

### Return Values

`ippStsNoErr`              Indicates no error.  
`ippStsNullPtrErr`       Indicates an error when one of the specified pointers is `NULL`.

## FBankGetCoeffs

*Retrieves the filter bank weight coefficients.*

---

### Syntax

```
IppStatus ippsFBankGetCoeffs_16s(const IppsFBankState_16s* pFBank, int fIdx,
Ipp32f* pCoeffs);
```

```
IppStatus ippsFBankGetCoeffs_32f(const IppsFBankState_32f* pFBank, int fIdx,
Ipp32f* pCoeffs);
```

### Parameters

*pFBank*                      Pointer to the filter bank structure  
*fIdx*                        Filter index.  
*pCoeffs*                    Pointer to the output vector that contains the filter coefficients.

## Description

The function `ippsFBankGetCoeffs` is declared in the `ippsr.h` file. This function copies the weight coefficients of the filter bank  $fIdx$  to the array  $pCoeffs$  which is of length  $(y_{k+1} - y_{k-1} + 1)$ .

Note that for the function flavor `ippsFBankGetCoeffs_16s`, the filter output weights are represented in floating-point format.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when $fIdx$ is less than 1 or greater than $nFilter$ .
<code>ippStsFBankErr</code>	Indicates an error when $fIdx$ filter coefficients are not available or valid.

## FBankSetCoeffs

*Sets the filter bank weight coefficients.*

---

### Syntax

```
IppStatus ippsFBankSetCoeffs_16s(IppsFBankState_16s* pFBank, int fIdx, const Ipp32f* pCoeffs);
```

```
IppStatus ippsFBankSetCoeffs_32f(IppsFBankState_32f* pFBank, int fIdx, const Ipp32f* pCoeffs);
```

### Parameters

$pFBank$	Pointer to the filter bank structure.
$fIdx$	Filter index.
$pCoeffs$	Pointer to the output coefficients vector.

### Description

The function `ippsFBankGetCoeffs` is declared in the `ippsr.h` file. This function sets the weight coefficients of the filter bank  $fIdx$ . The vector  $pCoeffs$  contains the weight coefficients from  $y_{k-1}$  to  $y_{k+1}$ .

Note that for `ippsFBankSetCoeffs_16s` function flavor, the weight coefficients are represented in floating-point format and are saturated to the `[-1,1]` interval.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>fIdx</code> is less than 1 or greater than <code>nFilter</code> .
<code>ippStsFBankErr</code>	Indicates an error when the weight coefficients are not available or valid.

## EvalFBank

*Performs the filter bank analysis.*

### Syntax

```

IppStatus ippsEvalFBank_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsFBankState_16s* pFBank, int scaleFactor);

IppStatus ippsEvalFBank_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pDst, const
IppsFBankState_16s* pFBank, int scaleFactor);

IppStatus ippsEvalFBank_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pDst, const
IppsFBankState_32s* pFBank, int scaleFactor);

IppStatus ippsEvalFBank_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsFBankState_32f* pFBank);

```

### Parameters

<code>pSrc</code>	Pointer to the source vector $[2^{pFFTOrder[0]}]$ .
<code>pDst</code>	Pointer to the filter bank coefficients vector $[nFilter]$ .
<code>pFBank</code>	Pointer to the filter bank structure.
<code>scaleFactor</code>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsEvalFBank` is declared in the `ippsr.h` file. This function performs the filter bank analysis for the input vector `pSrc`.

The execution mode set during the filter bank structure initialization determines the use of the input vector as follows:

If the `IPP_POWER_SPECTRUM` flag is set, the source vector is the wave signals. The magnitude of input signal spectrum is calculated for the filter bank analysis as follows:

$$x_k = \left| \sum_{i=0}^{N-1} pSrc[i] \cdot \exp\left(-j \cdot 2\pi \frac{ik}{N}\right) \right|,$$

$$0 \leq k \leq N/2.$$

Otherwise, the input vector is used directly for filter bank analysis:  $x_k = pSrc[i]$ ,  $0 \leq k \leq N/2$ .

Depending on the mode, corresponding formula (see [ippsMelFBankInitAlloc](#)) is used to obtain the filter bank coefficients. The input vector *pSrc* is destroyed after the analysis.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsFBankErr</code>	Indicates an error when <i>pFBank</i> structure is not ready for calculation.

## DCTLifterGetSize\_MulC0

*Gets the size of the DCT structure.*

---

### Syntax

```
IppStatus ippsDCTLifterGetSize_MulC0_16s(int lenDCT, int lenCeps, int* pSize);
```

### Parameters

<i>lenDCT</i>	Length of the DCT in the range (0 .. 8192].
<i>lenCeps</i>	Number of the output coefficients not including $C_0$ ( $0 < lenCeps \leq lenDCT$ ).
<i>pSize</i>	Pointer to the variable to contain the size in bytes.

## Description

The function `ippsDCTLifterGetSize_MulC0` is declared in the `ippsr.h` file. This function computes the size in bytes required for the DCT structure and associated storage.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>lenDCT</i> or <i>lenCeps</i> are out of bounds.

## DCTLifterInit\_MulC0

*Initializes the structure to perform DCT and lift the DCT coefficients.*

---

## Syntax

```
IppStatus ippsDCTLifterInit_MulC0_16s(IppsDCTLifterState_16s* pDCTLifter,
int lenDCT, const Ipp32s* pLifterQ15, int lenCeps);
```

## Parameters

<i>pDCTLifter</i>	Pointer to the structure to be initialized for the DCT calculation and liftering.
<i>lenDCT</i>	Length of the DCT in the range $(0 < lenDCT \leq 8192)$ .
<i>pLifterQ15</i>	Pointer to the Q15 format liftering coefficients vector $(0.0_{Q15} \leq pLifterQ15[k] < 512.0_{Q15})$ .
<i>lenCeps</i>	Number of the output coefficients not including $C_0$ $(0 < lenCeps \leq lenDCT)$ .

## Description

The function `ippsDCTLifterInit_MulC0` is declared in the `ippsr.h` file. This function initializes the structure for the DCT calculation and liftering similar to what is done by the `ippsDCTLifterInitAlloc_MulC0` function. However, memory for the structure must be previously allocated, and the size of memory must be determined by the function `ippsDCTLifterGetSize_MulC0` function.

$C_0$  is stored as the last element of the output vector.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>lenDCT</code> , <code>lenCeps</code> , or <code>pLifterQ15[k]</code> is out of the range.

## DCTLifterInitAlloc

*Initializes the structure and allocates memory to perform DCT and lift the DCT coefficients.*

---

### Syntax

```

IppStatus ippSDCTLifterInitAlloc_16s(IppsDCTLifterState_16s** pDCTLifter,
int lenDCT, int lenCeps, int nLifter, Ipp32f val);

IppStatus ippSDCTLifterInitAlloc_C0_16s(IppsDCTLifterState_16s** pDCTLifter,
int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);

IppStatus ippSDCTLifterInitAlloc_Mul_16s(IppsDCTLifterState_16s** pDCTLifter,
int lenDCT, const Ipp32f* pLifter, int lenCeps);

IppStatus ippSDCTLifterInitAlloc_MulC0_16s(IppsDCTLifterState_16s**
pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);

IppStatus ippSDCTLifterInitAlloc_32f(IppsDCTLifterState_32f** pDCTLifter,
int lenDCT, int lenCeps, int nLifter, Ipp32f val);

IppStatus ippSDCTLifterInitAlloc_C0_32f(IppsDCTLifterState_32f** pDCTLifter,
int lenDCT, int lenCeps, int nLifter, Ipp32f val, Ipp32f val0);

IppStatus ippSDCTLifterInitAlloc_Mul_32f(IppsDCTLifterState_32f** pDCTLifter,
int lenDCT, const Ipp32f* pLifter, int lenCeps);

IppStatus ippSDCTLifterInitAlloc_MulC0_32f(IppsDCTLifterState_32f**
pDCTLifter, int lenDCT, const Ipp32f* pLifter, int lenCeps);

```

### Parameters

<code>pDCTLifter</code>	Pointer to the structure to be created for the DCT calculation and lifting.
<code>lenDCT</code>	Length of the DCT.

<i>lenCeps</i>	Number of the output coefficients (not including $C_0$ ).
<i>nLifter</i>	Liftering factor.
<i>pLifter</i>	Pointer to the liftering coefficients vector.
<i>val</i>	The scale factor for the output coefficients (except $C_0$ ).
<i>val0</i>	The scale factor for $C_0$ .

## Description

The function `ippsDCTLifterInitAlloc` is declared in the `ippsr.h` file. This function initializes the structure for the DCT calculation and liftering.

The first DCT coefficient  $C_0$  is usually ignored. Therefore, the output of the `ippsDCTLifter` function contains only  $C_1$  to  $C_{lenCeps}$  coefficients. However, if the `C0` suffix is specified,  $C_0$  is stored as the last element of the output vector.

The liftering coefficients are calculated according to the below formulas:

For functions without the `Mul` and `C0` suffixes,

$$l_i = \left( 1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right) \right) \cdot val ,$$

$i = 1, \dots, lenCeps$ .

For functions with the `C0` suffix,  $l_0 = val0$ ,

$$l_i = \left( 1 + \frac{nLifter}{2} \cdot \sin\left(\frac{\pi \cdot i}{nLifter}\right) \right) \cdot val ,$$

$i = 1, \dots, lenCeps$ .

For functions with the `Mul` suffix,  $l_i = pLifter[i - 1]$ ,  $i = 1, \dots, lenCeps$ .

For functions that have both the `Mul` and `C0` suffixes:  $l_0 = pLifter[lenCeps - 1]$ ,  $l_i = pLifter[i - 1]$ ,  $i = 1, \dots, lenCeps$

## Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when <i>pLifter</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>lenDCT</i> , <i>lenCeps</i> , or <i>nLifter</i> is less than or equal to 0, or when <i>lenDCT</i> is less than <i>lenCeps</i> .
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.

## DCTLifterFree

*Destroys the structure used for the DCT and liftering.*

---

### Syntax

```
ippStatus ippSDCTLifterFree_16s(IppsDCTLifterState_16s* pDCTLifter);
ippStatus ippSDCTLifterFree_32f(IppsDCTLifterState_32f* pDCTLifter);
```

### Parameters

*pDCTLifter*                      Pointer to the DCT and liftering structure.

### Description

The function `ippSDCTLifterFree` is declared in the `ippsr.h` file. This function closes the DCT and liftering structure by freeing all memory associated with it.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDCTLifter</i> pointer is <code>NULL</code> .

## DCTLifter

*Performs the DCT and lifts the DCT coefficients.*

---

### Syntax

```
ippStatus ippSDCTLifter_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);

ippStatus ippSDCTLifter_32s16s_Sfs (const Ipp32s* pSrc, Ipp16s* pDst, const
IppsDCTLifterState_16s* pDCTLifter, int scaleFactor);
```



```
IppStatus ippsDCTLifter_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsDCTLifterState_32f* pDCTLifter);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector [ <i>lenDCT</i> ].
<i>pDst</i>	Pointer to the output vector [ <i>lenCeps</i> ] or [ <i>lenCeps</i> +1].
<i>pDCTLifter</i>	Pointer to the DCT and liftering structure.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsDCTLifter` is declared in the `ippsr.h` file. This function first performs the DCT and then lifts the DCT coefficients.

The DCT coefficients are calculated according to the formula:

$$y_i = \sum_{j=1}^{lenDCT} pSrc[j-1] \cdot \cos\left(\frac{\pi \cdot i \cdot (j-0.5)}{lenDCT}\right),$$

$i = 0, \dots, lenCeps$

The output coefficients are weighted as follows:

$$pDst[i-1] = l_i * y_i, i = 1, \dots, lenCeps$$

If the  $C_0$  coefficient is required (`pDCTLifter` is initialized by functions with the `C0` suffix), it is stored as the last element of the output vector:  $pDst[lenCeps] = l_0 * y_0$

$$pDst[lenCeps] = l_0 y_0$$

Example 8-1 below shows how the feature processing functions can be used for MFCC feature calculation.

## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when *pSrc*, *pDst*, or *pFBank* pointer is `NULL`.

**Example 8-1 MFCC feature calculation**

```
/* Input: samples[] Input samples
sample_number Number of samples
Output: mfccs[12*(sample_num-240)/160] The resulting MFCC coefficients
*/

void Calc_MFCC (Ipp32f *samples, int sample_num, Ipp32f *mfccs) {
    Ipp32f* frame_buffer,fbank_buffer,mfcc_cur;
    IppsFBankState_32f *fbank;
    IppsDCTLifterState_32f *dctl;
    int fft_len,fft_order;
    int i,j;
    float LogEnergy;
```

---

```

/* Initialize the structures */
ippsMelFBankInitAlloc_32f(&fbank, /* return the structure pointer */
    &fft_order, /* return the FFT length */
    400, /* 25ms window/512 point FFT */
    16000, /* sample rate */
    64, /* lowest frequency of interest */
    8000, /* highest frequency of interest */
    24, /* number of filter banks */
    1127.0, /* mel-scale factor 1 */
    700.0, /* mel-scale factor 2 */
    IPP_FBANK_MELWGT | IPP_POWER_SPECTRUM);
ippsDCTLifterInitAlloc_32f(&dctl, /* return the structure pointer */
    24, /* filter bank channels */
    12, /* number of MFCC coefficients */
    22, /* liftering */
    1.0); /* no scaling */

fft_len=1<<fft_order;
frame_buffer=ippsMalloc_32f(fft_len);
fbank_buffer=ippsMalloc_32f(24);
mfcc_cur=mfccs;

/* Calculate MFCC features */
for (i=j=0; i+400<sample_num; i+=160,mfcc_cur+=12,j++) {
    /* Organize the input wave data into a frame */
    ippsCopyWithPadding_32f(&samples[i],400,frame_buffer,fft_len);
    ippsDotProd_32f(frame_buffer,frame_buffer,fft_len,&LogEnergy);
    /* Pre-emphasize the input signal with factor 0.97 */
    ippsPreemphasize_32f(frame_buffer,400,0.97);
    frame_buffer[0]*=(1.0-0.97);
}

```

```

    /* Add the hamming window to the input signal */
    ippsWinHamming_32f_I(frame_buffer,400);
    /* Perform the filter bank analysis */
    ippsEvalFBank_32f(frame_buffer,fbank_buffer,fbank);
    ippsThreshold_LTV_32f_I(fbank_buffer, 24, 1.0, 1.0);
    ippsLn_32f_I(fbank_buffer, 24);

    /* Perform the DCT analysis and liftering */
    ippsDCTLifter_32f(fbank_buffer,mfcc_cur,dctl);
    mfcc_cur[12] = (float) log ((double) LogEnergy);
}
/* Normalize log energy */
ippsNormEnergy_32f(mfccs+11,12,(sample_num-240)/160,50.0,1.0);
/* Destroy the structures after calculation */
ippsFree(fbank_buffer);
ippsFree(frame_buffer);
ippsFBankFree_32f(fbank);
ippsDCTLifterFree_32f(dctl);
}

```

## NormEnergy

*Normalizes a vector of energy values.*

---

### Syntax

```

IppStatus ippsNormEnergy_32f(Ipp32f* pSrcDst, int step, int height, Ipp32f
silFloor, Ipp32f enScale);

IppStatus ippsNormEnergy_16s(Ipp16s* pSrcDst, int step, int height, Ipp16s
silFloor, Ipp16s val, Ipp32f enScale);

IppStatus ippsNormEnergy_RT_32f(Ipp32f* pSrcDst, int step, int height, Ipp32f
silFloor, Ipp32f maxE, Ipp32f enScale);

```

```
IppStatus ippsNormEnergy_RT_16s(Ipp16s* pSrcDst, int step, int height, Ipp16s
silFloor, Ipp16s maxE, Ipp16s val, Ipp32f enScale);
```

## Parameters

<i>pSrcDst</i>	Pointer to the input and output vector [ <i>height*step</i> ].
<i>step</i>	Sample step in the vector <i>pSrcDst</i> .
<i>height</i>	Number of samples for the normalization.
<i>silFloor</i>	Silence floor value.
<i>val</i>	Coefficient value.
<i>maxE</i>	Maximum energy value.
<i>enScale</i>	Energy scale.

## Description

The function `ippsNormEnergy` is declared in the `ippsr.h` file. This function normalizes the input vector that contains energy values. The normalization is performed as follows:

For functions without the RT suffix, the maximum energy value is calculated as

$$\text{maxE} = \max_{0 \leq i < \text{height}-1} p\text{SrcDst}[i \cdot \text{step}]$$

For all functions  $\text{val} = 1$  if not specified,

$$\text{minE} = \text{maxE} - (\text{silFloor} * \ln 10) / 10$$

$$p\text{SrcDst}[i * \text{step}] = \text{val} - (\text{maxE} - \max(p\text{SrcDst}[i * \text{step}], \text{minE})) * \text{enScale},$$

for  $0 \leq i < \text{height}$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.

## SumMeanVar

*Calculates both the sum of a the vector and its square sum.*

---

### Syntax

```

IppStatus ippsSumMeanVar_32f(const Ipp32f* pSrc, int srcStep, int height,
Ipp32f* pDstMean, Ipp32f* pDstVar, int width);

IppStatus ippsSumMeanVar_32f_I(const Ipp32f* pSrc, int srcStep, int height,
Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);

IppStatus ippsSumMeanVar_16s32f(const Ipp16s* pSrc, int srcStep, int height,
Ipp32f* pDstMean, Ipp32f* pDstVar, int width);

IppStatus ippsSumMeanVar_16s32f_I(const Ipp16s* pSrc, int srcStep, int height,
Ipp32f* pSrcDstMean, Ipp32f* pSrcDstVar, int width);

IppStatus ippsSumMeanVar_16s32s_Sfs(const Ipp16s* pSrc, int srcStep, int
height, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);

IppStatus ippsSumMeanVar_16s32s_ISfs(const Ipp16s* pSrc, int srcStep, int
height, Ipp32s* pSrcDstMean, Ipp32s* pSrcDstVar, int width, int scaleFactor);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector [ <i>height*srcStep</i> ].
<i>srcStep</i>	Row step in the vector <i>pSrc</i> .
<i>height</i>	Number of rows in the source vector.
<i>pDstMean</i>	Pointer to the destination vector that contains the sums [ <i>width</i> ].
<i>pDstVar</i>	Pointer to the destination vector that contains the square sums [ <i>width</i> ].
<i>pSrcDstMean</i>	Pointer to the source and destination vector that contains the sums [ <i>width</i> ].
<i>pSrcDstVar</i>	Pointer to the source and destination vector that contains the square sums [ <i>width</i> ].
<i>width</i>	Number of columns in the source vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsSumMeanVar` is declared in the `ippsr.h` file. This function calculates both the sums and the square sums of the source vectors as follows:

$$pDstVar[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pDstMean[j] = \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] ,$$

for  $j = 0, \dots, width - 1$ .

The function `ippsSumMeanVar_I` performs the in-place calculation as given by:

$$pSrcDstVar[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] \cdot pSrc[i \cdot srcStep + j]$$

$$pSrcDstMean[j] += \sum_{i=0}^{height-1} pSrc[i \cdot srcStep + j] ,$$

for  $j = 0, \dots, width - 1$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

`ippStsSizeErr` Indicates an error when *srcStep*, *width*, or *height* is less than or equal to 0, or when *width* is greater than *srcStep*.

## NewVar

*Calculates the variances given the sum and square sum accumulators.*

---

### Syntax

```

IppStatus ippsNewVar_32f(const Ipp32f* pSrcMean, const Ipp32f* pSrcVar,
Ipp32f* pDstVar, int width, Ipp32f val1, Ipp32f val2);

IppStatus ippsNewVar_32f_I(const Ipp32f* pSrcMean, Ipp32f* pSrcDstVar, int
width, Ipp32f val1, Ipp32f val2);

IppStatus ippsNewVar_32s_Sfs(const Ipp32s* pSrcMean, const Ipp32s* pSrcVar,
Ipp32s* pDstVar, int width, Ipp32f val1, Ipp32f val2, int scaleFactor);

IppStatus ippsNewVar_32s_ISfs(const Ipp32s* pSrcMean, Ipp32s* pSrcDstVar,
int width, Ipp32f val1, Ipp32f val2, int scaleFactor);

```

### Parameters

<i>pSrcMean</i>	Pointer to the vector that accumulates sums [ <i>width</i> ].
<i>pSrcVar</i>	Pointer to the vector that accumulates square sums [ <i>width</i> ].
<i>pSrcDstVar</i>	Pointer to the square sum accumulators and the resulting variance vector [ <i>width</i> ].
<i>pDstVar</i>	Pointer to the result variance vector [ <i>width</i> ].
<i>width</i>	Length of the variance vector <i>pDstVar</i> .
<i>val1, val2</i>	Constant coefficients.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsNewVar` is declared in the `ippsr.h` file. This function calculates the variance values given the sum and square sum accumulators as follows:

$$pDstVar[i] = (pSrcVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2 \quad ,$$



$i=0\dots width-1$

whereas the in-place function `ippsNewVar_I` uses the following formula:

$$pSrcDstVar[i] = (pSrcDstVar[i] - pSrcMean[i] \cdot pSrcMean[i] \cdot val1) \cdot val2 ,$$

for  $i = 0, \dots width - 1$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>width</code> is less than or equal to 0.

## RecSqrt

*Calculates square roots of a vector and their reciprocals.*

---

### Syntax

```
IppStatus ippsRecSqrt_32s_Sfs(Ipp32s* pSrcDst, int len, Ipp32s val, int
scaleFactor);
```

```
IppStatus ippsRecSqrt_32f(Ipp32f* pSrcDst, int len, Ipp32f val);
```

```
IppStatus ippsRecSqrt_32s16s_Sfs(const Ipp32s* pSrc, Ipp16s* pDst, int len,
Ipp32s val, int scaleFactor);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector .
<i>len</i>	Length of the source and destination vector.
<i>val</i>	Threshold for processed source values.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsRecSqrt` is declared in the `ippsr.h` file. This function calculates the square root of a vector and then takes the reciprocals. The operations are as follows:

$$pSrcDst[j] = \begin{cases} val & , \text{ if } pSrcDst[j] < val \\ \frac{1}{\sqrt{pSrcDst[j]}} & , \text{ otherwise} \end{cases}$$

for  $j = 0, \dots, len - 1$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0 or <code>val</code> is less than or equal to 0.
<code>ippStsInvZero</code>	Indicates a warning that all elements of the <code>pSrcDst</code> are less than <code>val</code> .

## AccCovarianceMatrix

*Accumulates covariance matrix.*

---

### Syntax

```
IppStatus ippsAccCovarianceMatrix_16s64f_D2L(const Ipp16s** mSrc, int height,
const Ipp16s* pMean, Ipp64f** mSrcDst, int width, Ipp64f val);
```

```
IppStatus ippsAccCovarianceMatrix_32f64f_D2L(const Ipp32f** mSrc, int height,
const Ipp32f* pMean, Ipp64f** mSrcDst, int width, Ipp64f val);
```

```
IppStatus ippsAccCovarianceMatrix_16s64f_D2(const Ipp16s* pSrc, int srcStep,
int height, const Ipp16s* pMean, Ipp64f* pSrcDst, int width, int dstStep,
Ipp64f val);
```

```
IppStatus ippsAccCovarianceMatrix_32f64f_D2(const Ipp32f* pSrc, int srcStep,
int height, const Ipp32f* pMean, Ipp64f* pSrcDst, int width, int dstStep,
Ipp64f val);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height*srcStep</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height*srcStep</i> ] .
<i>srcStep</i>	Row step in <i>pSrc</i> .
<i>pMean</i>	Pointer to the mean vector [ <i>width</i> ].
<i>width</i>	Length of the input matrix row, also length of the mean and variance vectors.
<i>pSrcDst</i>	Pointer to the result vector [ <i>width*dstStep</i> ] .
<i>mSrcDst</i>	Pointer to the result matrix [ <i>width*width</i> ] .
<i>dstStep</i>	Row step in <i>pSrcDst</i> .
<i>height</i>	Number of rows in the input matrix.
<i>val</i>	Value to multiply to each distance.

## Description

The function `ippsAccCovarianceMatrix` is declared in the `ippsr.h` file. This function accumulates destination covariance matrix elements according to formulas below.

For functions with the D2 suffix,

$$pSrcDst[i \cdot dstStep + j] = pSrcDst[i \cdot dstStep + j] + \\ + val \cdot \sum_{k=0}^{height-1} (pSrc[k \cdot srcStep + i] - pMean[j]) \cdot (pSrc[k \cdot srcStep + j] - pMean[j])$$

$$pSrcDst[j \cdot dstStep + i] = pSrcDst[i \cdot dstStep + j] ,$$

for  $i = 0, \dots, width - 1, j = i, \dots, width - 1$ .

For functions with the D2L suffix,

$$mSrcDst[i][j] = mSrcDst[i][j] + val \cdot \sum_{k=0}^{height-1} (mSrc[k][i] - pMean[j]) \cdot (mSrc[k][j] - pMean[j])$$

$$mSrcDst[j][i] = mSrcDst[i][j] ,$$

for  $i = 0, \dots, width - 1, j = i, \dots, width - 1$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>srcStep</i> or <i>dstStep</i> is less than <i>width</i> .

## CompensateOffset

*Removes the DC offset of the input signals.*

---

### Syntax

```

IppStatus ippCompensateOffset_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp32f val);

IppStatus ippCompensateOffset_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
len, Ipp32f* pSrcDst0, Ipp32f dst0, Ipp32f val);

IppStatus ippCompensateOffset_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
pSrcDst0, Ipp16s dst0, Ipp32f val);

IppStatus ippCompensateOffset_32f_I(Ipp32f* pSrcDst, int len, Ipp32f*
pSrcDst0, Ipp32f dst0, Ipp32f val);

IppStatus ippCompensateOffsetQ15_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
len, Ipp16s* pSrcDst0, Ipp16s dst0, Ipp16s valQ15);

```

```
IppStatus ippsCompensateOffsetQ15_16s_I(Ipp16s* pSrcDst, int len, Ipp16s*
pSrcDst0, Ipp16s dst0, Ipp16s valQ15);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector [ <i>len</i> ].
<i>pDst</i>	Pointer to the destination vector [ <i>len</i> ].
<i>pSrcDst</i>	Pointer to the source and destination vector [ <i>len</i> ] for in-place operations.
<i>pSrcDst0</i>	Pointer to the previous source element.
<i>len</i>	Number of elements in the vector.
<i>dst0</i>	Previous destination element.
<i>val</i>	Constant for offset compensation.
<i>valQ15</i>	Real-valued Q15 format constant for offset compensation ( $0.0_{Q31} \leq valQ15 < 1.0_{Q31}$ ).
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsCompensateOffset` is declared in the `ippsr.h` file. This function removes the offset of the input signals. The destination vector is calculated as follows.

For the not-inplace flavors of the `ippsCompensateOffset` function:

$$pDst[0] = pSrc[0] - pSrc[0] + val * dst0$$

$$pDst[i] = pSrc[i] - pSrc[i-1] + val * pDst[i-1], 1 \leq i \leq len - 1$$

$$pSrcDst0[0] = pSrc[len - 1],$$

and for in-place flavors of the `ippsCompensateOffset` function:

$$y = pSrcDst[0] - pSrcDst0[0] + val * dst0, x = pSrcDst[0], pSrcDst[0] = y,$$

$$y = pSrcDst[i] - pSrcDst[i-1] + val * x, x = pSrcDst[i], pSrcDst[i] = y,$$

$$1 \leq i \leq len - 1,$$

$$pSrcDst0[0] = x.$$

## Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0, or when <code>valQ15</code> is out of range.

## Derivative Functions

Functions described in this section are used to calculate first and second derivatives of feature vectors. These functions process the sequence of input feature vectors and generate the corresponding output feature vectors that contain derivatives.

The input sequence  $a_0, \dots, a_{N-1}$  is stored as a one-dimensional array of length  $N \cdot M$ , where  $N$  is the number of feature vectors and  $M$  is the dimension of each feature  $a_i$ . Similarly, the output sequence  $b_0, \dots, b_{N-1}$  is also stored as a one-dimensional array of length  $N \cdot K$ , where  $K$  is the dimension of each feature  $a_j$ . Certain constraints on values of  $M$  and  $K$  must be observed, specifically,

$$K \leq M;$$

$$K \leq 2M, \text{ for generating first derivatives;}$$

$$K \leq 3M, \text{ for generating both first and second derivatives.}$$

The `ippsCopyColumn` function copies the input sequence to the output sequence, that is, places the base features into the output sequence. The base features are placed in the first  $M$  elements of each output vector. The function `ippsEvalDelta` is then used to calculate

the derivatives. While these two functions are used for general derivative calculation, the functions `ippsDelta` and `ippsDeltaDelta` provide combined but specific functionalities.

As the derivative operation accesses feature vectors both in the history and in the future, special treatment is required for the first `winSize` (delta window size) features, as well as for the last `winSize` features. For the first `winSize` features, the first feature vector is usually repeated to provide the history. For the last `winSize` features, the last feature vector is repeated to provide the future information.

## CopyColumn

*Copies the input sequence into the output sequence.*

---

### Syntax

```
IppStatus ippsCopyColumn_16s_D2(const Ipp16s* pSrc, int srcWidth, Ipp16s* pDst, int dstWidth, int height);
```

```
IppStatus ippsCopyColumn_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f* pDst, int dstWidth, int height);
```

### Parameters

<i>pSrc</i>	Pointer to the input feature sequence [ <i>height*srcWidth</i> ].
<i>srcWidth</i>	Length of each input feature vector.
<i>pDst</i>	Pointer to the output feature sequence [ <i>height*dstWidth</i> ].
<i>dstWidth</i>	Length of each output feature vector.
<i>height</i>	Number of features in the sequence.

### Description

The function `ippsCopyColumn` is declared in the `ippsr.h` file. This function copies the input feature sequence into the output sequence as follows:

$$pDst[j*dstWidth+i] + pSrc[j*srcWidth+i], 0 \leq i < srcWidth, 0 \leq j < height.$$

The unspecified elements in the output sequence remain unchanged.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippSsSizeErr</code>	Indicates an error when <i>height</i> or <i>srcWidth</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>dstWidth</i> is less than <i>srcWidth</i> .

## EvalDelta

*Calculates the derivatives of feature vectors.*

---

### Syntax

```
IppStatus ippsEvalDelta_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step, int width, int offset, int winSize, Ipp16s val, int scaleFactor);
```

```
IppStatus ippsEvalDelta_32f_D2(Ipp32f* pSrcDst, int height, int step, int width, int offset, int winSize, Ipp32f val);
```

```
IppStatus ippsEvalDeltaMul_16s_D2Sfs(Ipp16s* pSrcDst, int height, int step, const Ipp16s* pVal, int width, int offset, int winSize, int scaleFactor);
```

```
IppStatus ippsEvalDeltaMul_32f_D2(Ipp32f* pSrcDst, int height, int step, const Ipp32f* pVal, int width, int offset, int winSize);
```

### Parameters

<i>pSrcDst</i>	Pointer to the input and output sequence [ <i>height</i> * <i>step</i> ].
<i>height</i>	Number of features in <i>pSrcDst</i> .
<i>step</i>	Length of each feature in <i>pSrcDst</i> .
<i>width</i>	The number of derivatives to be calculated for each feature.
<i>offset</i>	Offset to place the derivative values.
<i>winSize</i>	The delta window size
<i>val</i>	The delta coefficient.
<i>pVal</i>	Pointer to the delta coefficients vector [ <i>width</i> ].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsEvalDelta` is declared in the `ippsr.h` file. This function calculates the derivatives for the input feature sequence. The base feature ranges from *offset* to (*offset* + *width* - 1) in each feature vector. The output derivatives are stored in each feature vector next to the base features ranging from (*offset* + *width*) to (*offset* + 2 \* *width* - 1).



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , or <i>winSize</i> is less than or equal to 0; or <i>offset</i> is less than 0; or <i>height</i> is less than $2 * \text{winSize}$ .
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than $\text{offset} + 2 * \text{width}$ .

## Delta

*Copies the base features and calculates the derivatives of feature vectors.*

---

### Syntax

```

IppStatus ippsDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);

IppStatus ippsDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth, Ipp16s*
pDst, int dstStep, int height, Ipp16s val, int deltaMode, int scaleFactor);

IppStatus ippsDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
pDst, int dstStep, int height, Ipp32f val, int deltaMode);

IppStatus ippsDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
pDst, int dstStep, int height, Ipp32f val, int deltaMode);

IppStatus ippsDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
scaleFactor);

IppStatus ippsDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s* pVal,
int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode, int
scaleFactor);

IppStatus ippsDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal,
int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);

IppStatus ippsDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f* pVal,
int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);

```

## Parameters

<i>pSrc</i>	Pointer to the input feature sequence [ <i>height*srcWidth</i> ].
<i>srcWidth</i>	Length of the feature vector in the input sequence <i>pSrc</i> .
<i>pDst</i>	Pointer to the output feature sequence
<i>dstStep</i>	Length of the feature vector in the output sequence <i>pDst</i> .
<i>height</i>	Number of feature vectors.
<i>val</i>	Delta coefficient.
<i>deltaMode</i>	Execution mode.
<i>pVal</i>	Pointer to the delta coefficients vector [ <i>width</i> ].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

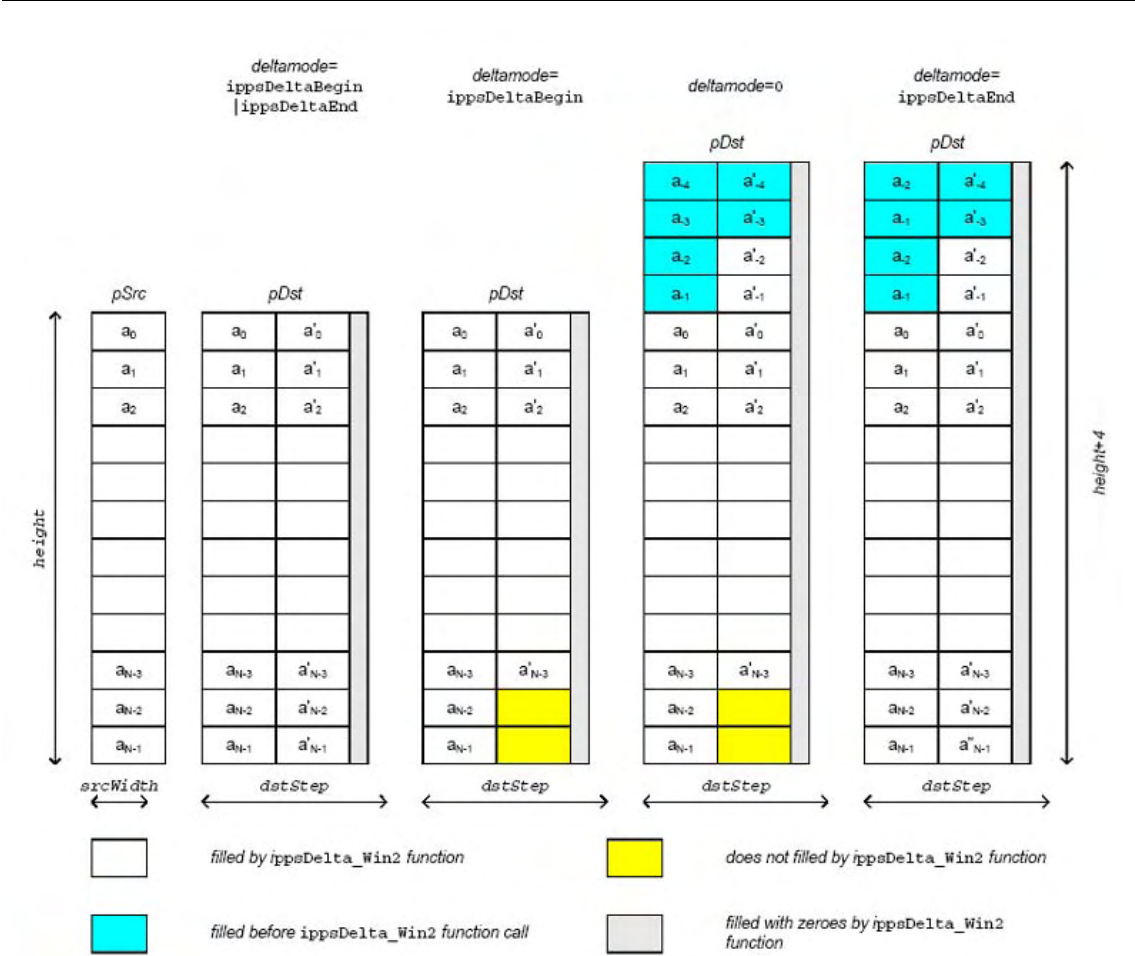
The functions `ippsDelta` and `ippsDeltaMul` are declared in the `ippsr.h` file. These functions provide a combined functionality of `ippsCopyColumn` and `ippsEvalDelta`. First, the input feature vectors are copied to the output sequence. Then the derivatives are calculated.

The execution mode *deltaMode* provides additional controls for the base feature copy and derivative calculation process. The admissible values of *deltaMode* are the following:

1. *deltaMode* is equal to `IPP_DELTA_BEGIN | IPP_DELTA_END`
2. *deltaMode* is equal to 0
3. *deltaMode* is equal to `IPP_DELTA_BEGIN`
4. *deltaMode* is equal to `IPP_DELTA_END`

Figure 8-1 below illustartes the four delta calculation modes.

Figure 8-1 Execution Modes of `ippsDelta_Win2` function



Return Values

- ippStsNoErr
- Indicates no error.
- ippStsNullPtrErr
- Indicates an error when one of the specified pointers is NULL.

<code>ippStsSizeErr</code>	Indicates an error when <i>srcWidth</i> is less than or equal to 0; or <i>height</i> is less than 0; or <i>height</i> is less than $2 * \text{winSize}$ when <i>deltaMode</i> is equal to <code>IPP_DELTA_BEGIN</code> ; or <i>height</i> is equal to 0 when <i>deltaMode</i> is not equal to <code>IPP_DELTA_END</code> .
<code>ippStsStrideErr</code>	Indicates an error when <i>dstStep</i> is less than $2 * \text{srcWidth}$ .

## DeltaDelta

*Copies the base features and calculates their first and second derivatives.*

---

### Syntax

```
IppStatus ippsDeltaDelta_Win1_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
deltaMode, int scaleFactor);
```

```
IppStatus ippsDeltaDelta_Win2_16s_D2Sfs(const Ipp16s* pSrc, int srcWidth,
Ipp16s* pDst, int dstStep, int height, Ipp16s val1, Ipp16s val2, int
deltaMode, int scaleFactor);
```

```
IppStatus ippsDeltaDelta_Win1_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);
```

```
IppStatus ippsDeltaDelta_Win2_32f_D2(const Ipp32f* pSrc, int srcWidth, Ipp32f*
pDst, int dstStep, int height, Ipp32f val1, Ipp32f val2, int deltaMode);
```

```
IppStatus ippsDeltaDeltaMul_Win1_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
int scaleFactor);
```

```
IppStatus ippsDeltaDeltaMul_Win2_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
pVal, int srcWidth, Ipp16s* pDst, int dstStep, int height, int deltaMode,
int scaleFactor);
```

```
IppStatus ippsDeltaDeltaMul_Win1_32f_D2(const Ipp32f* pSrc, const Ipp32f*
pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

```
IppStatus ippsDeltaDeltaMul_Win2_32f_D2(const Ipp32f* pSrc, const Ipp32f*
pVal, int srcWidth, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

## Parameters

<i>pSrc</i>	Pointer to the input feature sequence [ <i>height*srcWidth</i> ].
<i>srcWidth</i>	Length of the input feature in the input sequence <i>pSrc</i> .
<i>pDst</i>	Pointer to the output feature sequence.
<i>dstStep</i>	Length of the output feature in the output sequence <i>pDst</i> .
<i>height</i>	Number of feature vectors.
<i>val1, val2</i>	The first and second delta coefficients.
<i>deltaMode</i>	Execution mode.
<i>pVal</i>	Pointer to the delta coefficients vector [ <i>width</i> ].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

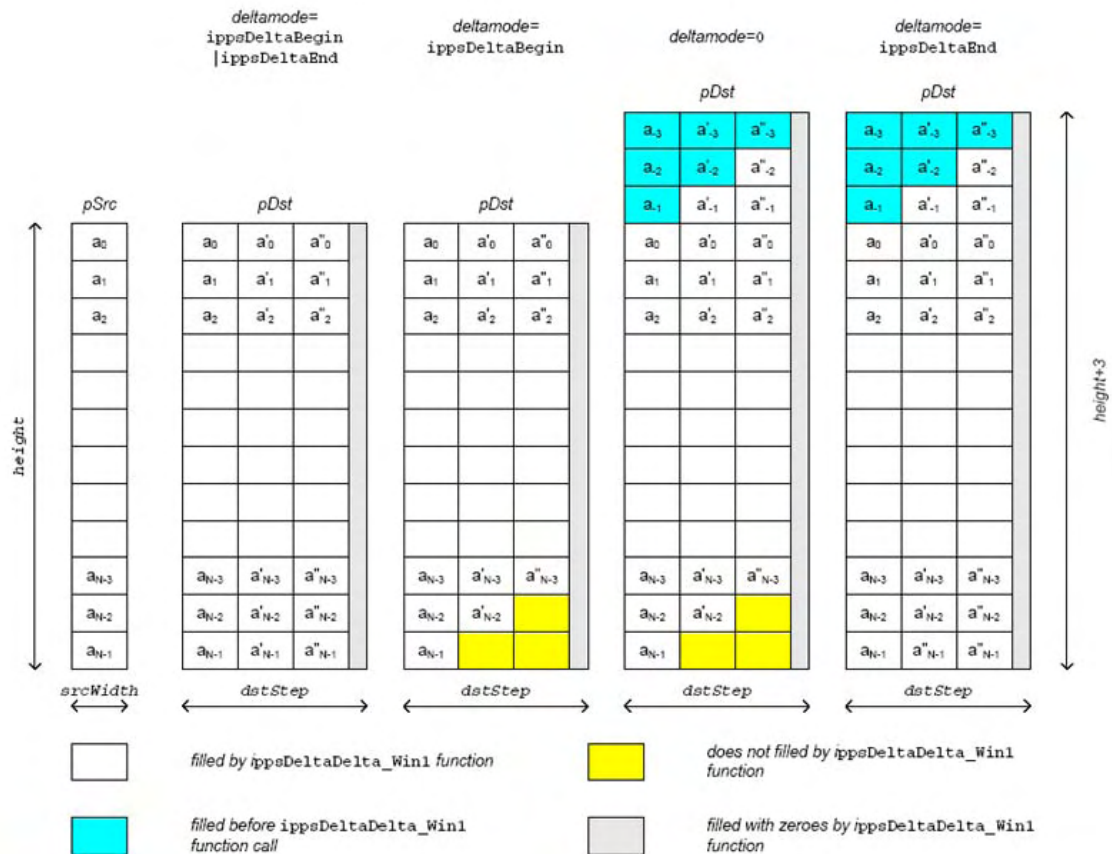
The functions `ippsDeltaDelta` and `ippsDeltaDeltaMul` are declared in the `ippsr.h` file. These functions provide a combined functionality of `ippsCopyColumn` and double operations of `ippsEvalDelta`. First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated.

The execution mode *deltaMode* provides additional controls for the base feature copy and derivative calculation process. The admissible values of *deltaMode* are the following:

1. *deltaMode* is equal to `IPP_DELTA_BEGIN|IPP_DELTA_END`
2. *deltaMode* is equal to 0
3. *deltaMode* is equal to `IPP_DELTA_BEGIN`
4. *deltaMode* is equal to `IPP_DELTA_END`

Figure 8-2 below illustrates the four delta-delta calculation modes.

**Figure 8-2. Execution Modes of `ippsDeltaDelta_Win1` function**



## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is NULL.

<code>ippStsSizeErr</code>	Indicates an error when <i>srcWidth</i> is less than or equal to 0; or <i>height</i> is less than 0; or <i>height</i> is less than $3 * \text{winSize}$ when <i>deltaMode</i> is equal to <code>IPP_DELTA_BEGIN</code> ; or <i>height</i> is equal to 0 when <i>deltaMode</i> is not equal to <code>IPP_DELTA_END</code> .
<code>ippStsStrideErr</code>	Indicates an error when <i>dstStep</i> is less than $3 * \text{srcWidth}$ .

## Pitch Super Resolution

This section describes functions that can be used for implementing the pitch resolution algorithm (see [Med91]).

Some of these functions are used in the Intel® IPP Speech Recognition Sample. See *Speech Processing* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## CrossCorrCoeffDecim

*Calculates vector of cross correlation coefficients with decimation.*

---

### Syntax

```
ippStatus ippsCrossCorrCoeffDecim_16s32f(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, int maxLen, int minLen, Ipp32f* pDst, int dec);
```

### Parameters

<i>pSrc1</i>	Pointer to the first input vector [ <i>maxLen</i> ].
<i>pSrc2</i>	Pointer to the second input vector [ <i>maxLen</i> ].
<i>pDst</i>	Pointer to the output vector [ $(\text{maxLen} - \text{minLen}) / \text{dec} + 1$ ].
<i>maxLen</i>	Maximal length of cross correlation.
<i>minLen</i>	Minimal length of cross correlation.
<i>dec</i>	Decimation step.

### Description

The function `ippsCrossCorrCoeffDecim` is declared in the `ippsr.h` file. This function calculates the vector of cross correlation coefficients with lengths from *minLen* to *maxLen* using the decimation step *dec*.

Computations are performed as given by the following formula:

$$pDst[k] = \begin{cases} 0, & \text{if } (\mathbf{x}^k, \mathbf{x}^k)_{dec} \cdot (\mathbf{y}^k, \mathbf{y}^k)_{dec} = 0 \\ \frac{(\mathbf{x}^k, \mathbf{y}^k)_{dec}}{\sqrt{(\mathbf{x}^k, \mathbf{x}^k)_{dec}} \cdot \sqrt{(\mathbf{y}^k, \mathbf{y}^k)_{dec}}}, & \text{otherwise} \end{cases}$$

where

$$\mathbf{x}_i^k = pSrc1[\maxLen - \minLen - dec \cdot k + i], \quad \mathbf{y}_i^k = pSrc2[i],$$

$i = 0, \dots, \minLen + dec \cdot k - 1, k = 0, \dots, (\maxLen - \minLen)/dec,$

and  $(a, b)_{dec}$  denotes the decimated dot product of two vectors  $a$  and  $b$ :

$$(\mathbf{a}, \mathbf{b})_{dec} = \sum_{i=0}^{(\text{len}-1)/dec} a[i \cdot dec] \times b[i \cdot dec]$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>minLen</code> or <code>dec</code> is less than or equal to 0, or when <code>maxLen</code> is less than <code>minLen</code> .



## CrossCorrCoeff

*Calculates the cross correlation coefficient.*

---

### Syntax

```
IppStatus ippsCrossCorrCoeff_16s32f(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
int len, Ipp32f* pResult);
```

```
IppStatus ippsCrossCorrCoeffPartial_16s32f(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, int len, Ipp32f val, Ipp32f* pResult);
```

### Parameters

<i>pSrc1</i>	Pointer to the first input vector.
<i>pSrc2</i>	Pointer to the second input vector.
<i>len</i>	Length of the input vectors.
<i>pResult</i>	Pointer to the result cross correlation coefficient value.
<i>val</i>	Value used as the squared magnitude of the first vector.

### Description

The functions `ippsCrossCorrCoeff` and `ippsCrossCorrCoeffPartial` are declared in the `ippsr.h` file. These functions calculate the cross correlation coefficient of two vectors according to the formulas given below.

For the `ippsCrossCorrCoeff` function:

$$pResult[0] = \begin{cases} 0, & \text{if } (\mathbf{x}, \mathbf{x}) \cdot (\mathbf{y}, \mathbf{y}) = 0 \\ \frac{(\mathbf{x}, \mathbf{y})}{\sqrt{(\mathbf{x}, \mathbf{x})} \cdot \sqrt{(\mathbf{y}, \mathbf{y})}}, & \text{otherwise} \end{cases}$$

For the `ippsCrossCorrCoeffPartial` function:

$$pResult[0] = \begin{cases} 0, & \text{if } val \cdot (y, y) = 0 \\ \frac{(x, y)}{\sqrt{val} \cdot \sqrt{(y, y)}}, & \text{otherwise} \end{cases}$$

where

$x_i = pSrc1[i - 1]$ ,  $y_i = pSrc2[i - 1]$  for  $i = 1, \dots, len$ ,

and  $(a, b)$  denotes the dot product of two vectors  $a$  and  $b$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates an error when <code>val</code> is less than 0.

## CrossCorrCoeffInterpolation

Calculates interpolated cross correlation coefficient.

### Syntax

```
IppStatus ippCrossCorrCoeffInterpolation_16s32f(const Ipp16s* pSrc1, const
Ipp16s* pSrc2, int len, Ipp32f* pBeta, Ipp32f* pResult);
```

### Parameters

<code>pSrc1</code>	Pointer to the first input vector [ <code>len</code> ].
<code>pSrc2</code>	Pointer to the second input vector [ <code>len+1</code> ].
<code>len</code>	Length of the input vectors.
<code>pResult</code>	Pointer to the result cross correlation coefficient value.
<code>pBeta</code>	Pointer to the result value of fractional part of the pitch period.

## Description

The function `ippsCrossCorrCoeffInterpolation` is declared in the `ippsr.h` file. This function calculates interpolated cross correlation coefficient *pResult* of two vectors and the fractional part *pBeta* of the pitch period according to the following formulas:

$$pBeta[0] = \beta = \frac{(\mathbf{x}, \mathbf{y}') \cdot (\mathbf{y}, \mathbf{y}) - (\mathbf{x}, \mathbf{y}) \cdot (\mathbf{y}, \mathbf{y}')}{(\mathbf{x}, \mathbf{y}') \cdot [(\mathbf{y}, \mathbf{y}) - (\mathbf{y}, \mathbf{y}')] + (\mathbf{x}, \mathbf{y}) \cdot [(\mathbf{y}', \mathbf{y}') - (\mathbf{y}, \mathbf{y}')]},$$

and if  $0 \leq \beta < 1$ , then

$$pResult[0] = \frac{(1 - \beta) \cdot (\mathbf{x}, \mathbf{y}) + \beta \cdot (\mathbf{x}, \mathbf{y}')}{\sqrt{(\mathbf{x}, \mathbf{x}) \cdot ((1 - \beta)^2 \cdot (\mathbf{y}, \mathbf{y}) + 2\beta \cdot (1 - \beta) \cdot (\mathbf{y}, \mathbf{y}') + \beta^2 \cdot (\mathbf{y}', \mathbf{y}'))}},$$

where

$x_i = pSrc1[i - 1]$ ,  $y_i = pSrc2[i - 1]$ ,  $y'_i = pSrc2[i]$  for  $i = 1, \dots, len$ ,

and  $(\mathbf{a}, \mathbf{b})$  denotes the dot product of two vectors  $\mathbf{a}$  and  $\mathbf{b}$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>srcStep</code> or <code>dstStep</code> is less than <code>width</code> .
<code>ippStsDivByZero</code>	Indicates a warning that the denominator in formula for <i>pBeta</i> [0] has zero value. Operation execution is not aborted.

## Model Evaluation

This section describes functions that evaluate the acoustic and language models.

Some of these functions are used in the Intel® IPP Speech Recognition Samples. See *Gaussian Calculations* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## AddNRows

*Adds N vectors from a vector array.*

---

### Syntax

```
IppStatus ippsAddNRows_32f_D2(Ipp32f* pSrc, int height, int offset, int step,
Ipp32s* pInd, Ipp16u* pAddInd, int rows, Ipp32f* pDst, int width, Ipp32f
weight);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector array [ <i>height*step</i> ].
<i>height</i>	Number of rows in the vector array <i>pSrc</i> .
<i>offset</i>	Offset to the vector of interest in the vector array <i>pSrc</i> .
<i>step</i>	Row step in the vector array <i>pSrc</i> .
<i>pInd</i>	Pointer to the index vector [ <i>rows</i> ].
<i>pAddInd</i>	Pointer to the additional index vector [ <i>rows</i> ].
<i>rows</i>	Number of vectors to be added.
<i>pDst</i>	Pointer to the output vector [ <i>width</i> ].
<i>width</i>	Length of the output vector <i>pDst</i> .
<i>weight</i>	Weight value to be added to the output.

### Description

The function `ippsAddNRows` is declared in the `ippsr.h` file. This function calculates the sum of the *rows* number of vectors, according to the sum of the indexing vectors *pInd* and *pAddInd*, from the vector array *pSrc*.

$$pDst[k] = weight + \sum_{i=0}^{rows} pSrc[k + offset + step \cdot (pInd[i] + pAddInd[i])] ,$$

$0 \leq k < width$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pInd</i> , <i>pAddInd</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , or <i>rows</i> is less than or equal to 0; or ( <i>pInd</i> [ <i>i</i> ] + <i>pAddInd</i> [ <i>i</i> ]), or <i>offset</i> is less than 0; or ( <i>pInd</i> [ <i>i</i> ] + <i>pAddInd</i> [ <i>i</i> ]) is greater than or equal to <i>height</i> .
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> + <i>offset</i> .

## ScaleLM

Scales vector elements with thresholding.

### Syntax

```
IppStatus ippScaleLM_16s32s(const Ipp16s* pSrc, Ipp32s* pDst, int len,
Ipp16s floor, Ipp16s scale, Ipp32s base);
```

```
IppStatus ippScaleLM_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f
floor, Ipp32f scale, Ipp32f base);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector .
<i>pDst</i>	Pointer to the output vector.
<i>len</i>	Length of the vectors <i>pSrc</i> or <i>pDst</i> .
<i>floor</i>	Threshold value.
<i>scale</i>	Scaling factor.
<i>base</i>	Additive factor.

### Description

The function `ippScaleLM` is declared in the `ippsr.h` file. This function sets threshold on the input vector *pSrc* and scales the vector elements as follows:

$$pDst[n] = scale * \max(pSrc[n], floor) + base, 0 \leq n < len.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## LogAdd

Adds two vectors in the logarithmic representation.

### Syntax

```
IppStatus ippsLogAdd_32f(const Ipp32f* pSrc, Ipp32f * pSrcDst, int len,
IppHintAlgorithm hint);
```

```
IppStatus ippsLogAdd_64f(const Ipp64f* pSrc, Ipp64f >* pSrcDst, int len,
IppHintAlgorithm hint);
```

```
IppStatus ippsLogAdd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
int scaleFactor , IppHintAlgorithm hint);
```

```
IppStatus ippsLogAdd_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len,
int scaleFactor , IppHintAlgorithm hint);
```

### Parameters

<i>pSrc</i>	Pointer to the first input vector [ <i>len</i> ].
<i>pSrcDst</i>	Pointer to the second input vector and the output vector [ <i>len</i> ].
<i>len</i>	Number of elements in the input and output vectors.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<i>hint</i>	Suggestion for using specific code for logarithmic addition.

### Description

The function `ippsLogAdd` is declared in the `ippsr.h` file. This function adds the *pSrc* and *pSrcDst* vectors, whose elements are in the logarithmic representation. For functions with floating point arguments, the output vector *pSrcDst*, also in the logarithmic representation, is calculated as follows:

$$pSrcDst[i] = \ln(e^{pSrc[i]} + e^{pSrcDst[i]}),$$

$$0 \leq i < len.$$

For functions with integer arguments, the output vector *pSrcDst* is calculated as follows:

$$pSrcDst[i] = 2^{scaleFactor} \cdot \ln\left(e^{pSrc[i] \cdot 2^{-scaleFactor}} + e^{pSrcDst[i] \cdot 2^{-scaleFactor}}\right),$$

$$0 \leq i < len.$$

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The possible values for the parameter *hint* are listed in Table [Hint Arguments](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## LogSub

*Subtracts a vector from another vector, in the logarithmic representation.*

---

### Syntax

```

IppStatus ippsLogSub_32f(const Ipp32f* pSrc, Ipp32f* pSrcDst, int len);
IppStatus ippsLogSub_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len);
IppStatus ippsLogSub_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
int scaleFactor);
IppStatus ippsLogSub_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pSrcDst, int len,
int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the first input vector [ <i>len</i> ].
<i>pSrcDst</i>	Pointer to the second input vector and the output vector [ <i>len</i> ].
<i>len</i>	Number of elements in the input and output vectors.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<i>hint</i>	Suggestion for using specific code for logarithmic addition.

## Description

The function `ippsLogSub` is declared in the `ippsr.h` file. This function subtracts the vector *pSrcDst* from the vector *pSrc*, both of them in the logarithmic representation.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsRangeErr</code>	Indicates an error when <i>pSrc</i> [ <i>i</i> ] is less than or equal to <i>pSrcDst</i> [ <i>i</i> ].

## LogSum

*Sums vector elements in the logarithmic representation.*

---

### Syntax

```

IppStatus ippsLogSum_32f(const Ipp32f* pSrc, Ipp32f* pResult, int len,
IppHintAlgorithm hint);

IppStatus ippsLogSum_64f(const Ipp64f* pSrc, Ipp64f* pResult, int len,
IppHintAlgorithm hint);

IppStatus ippsLogSum_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pResult, int len,
int scaleFactor, IppHintAlgorithm hint);

IppStatus ippsLogSum_32s_Sfs(const Ipp32s* pSrc, Ipp32s* pResult, int len,
int scaleFactor, IppHintAlgorithm hint);

```



## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>len</i> ].
<i>pResult</i>	Pointer to the result value.
<i>len</i>	Number of elements in the input vector.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<i>hint</i>	Suggestion for using specific code for logarithmic addition.

## Description

The function `ippsLogSum` is declared in the `ippsr.h` file. This function sums the source vector elements that are taken in logarithmic representation.

The *hint* argument suggests using special code which provides for faster but less accurate calculation, or more accurate but slower calculation. The possible values for the parameter *hint* are listed in Table [Hint Arguments](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## MahDistSingle

*Calculates the Mahalanobis distance for a single observation vector.*

---

### Syntax

```

IppStatus ippsMahDistSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, int scaleFactor);

IppStatus ippsMahDistSingle_16s32f(const Ipp16s* pSrc, const Ipp16s* pMean,
const Ipp16s* pVar, int len, Ipp32f* pResult);

IppStatus ippsMahDistSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean,
const Ipp32f* pVar, int len, Ipp32f* pResult);

IppStatus ippsMahDistSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean,
const Ipp64f* pVar, int len, Ipp64f* pResult);

```

```
IppStatus ippsMahDistSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
const Ipp32f* pVar, int len, Ipp64f* pResult);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>len</i> ].
<i>pMean</i>	Pointer to the mean vector [ <i>len</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>len</i> ].
<i>len</i>	Number of elements in the input, mean, and variance vectors.
<i>pResult</i>	Pointer to the result.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsMahDistSingle` is declared in the `ippsr.h` file. This function calculates the Mahalanobis distance for the input vector *pSrc*, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows:

$$pResult[0] = \sum_{i=0}^{len-1} pVar[i] \cdot (pSrc[i] - pMean[i])^2$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## MahDist

*Calculates the Mahalanobis distances for multiple observation vectors.*

---

### Syntax

```
IppStatus ippsMahDist_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height);
```

```
IppStatus ippsMahDist_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, const
Ipp32f* pVar, int width, Ipp32f* pDst, int height);
```

```
IppStatus ippsMahDist_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int height);
```

```
IppStatus ippsMahDist_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, const
Ipp64f* pVar, int width, Ipp64f* pDst, int height);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height*step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height*step</i> ].
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pMean</i>	Pointer to the mean vector [ <i>width</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>width</i> ].
<i>width</i>	Length of the mean and variance vectors.
<i>pDst</i>	Pointer to the result vector [ <i>height</i> ].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .

## Description

The function *ippsMahDist* is declared in the *ippsr.h* file. This function calculates the Mahalanobis distances for multiple input vectors, given the mean vector *pMean* and the variance vector *pVar*. The calculation is as follows: For functions with the D2 suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (pSrc[i \cdot step + j] - pMean[j])^2, \quad 0 \leq i < height.$$

,  $0 \leq i < height$ .

For functions with the D2L suffix,

$$pDst[i] = \sum_{j=0}^{width-1} pVar[j] \cdot (mSrc[i][j] - pMean[j])^2, \quad 0 \leq i < height.$$

,  $0 \leq i < height$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## MahDistMultiMix

*Calculates the Mahalanobis distances for multiple means and variances.*

---

### Syntax

```
IppStatus ippsMahDistMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f* pVar,
int step, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
```

```
IppStatus ippsMahDistMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
mVar, const Ipp32f* pSrc, int width, Ipp32f* pDst, int height);
```

```
IppStatus ippsMahDistMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f* pVar,
int step, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
```

```
IppStatus ippsMahDistMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
mVar, const Ipp64f* pSrc, int width, Ipp64f* pDst, int height);
```

### Parameters

<i>pMean</i>	Pointer to the mean vector [ <i>height</i> * <i>step</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>height</i> * <i>step</i> ].
<i>mMean</i>	Pointer to the mean matrix [ <i>height</i> ][ <i>width</i> ].
<i>mVar</i>	Pointer to the variance matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in mean and variance vectors.
<i>pSrc</i>	Pointer to the input vector\ .
<i>width</i>	Length of the input vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the result vector.

*height* Length of the result vector *pDst*.

## Description

The function `ippsMahDistMultiMix` is declared in the `ippsr.h` file. This function calculates the Mahalanobis distances for a single observation vector but multiple mean and variance pairs.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## LogGaussSingle

*Calculates the observation probability for a single Gaussian with an observation vector.*

---

### Syntax

#### Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val, int
scaleFactor);
```

```
IppStatus ippsLogGaussSingle_Scaled_16s32f(const Ipp16s* pSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val, int
scaleFactor);
```

```
IppStatus ippsLogGaussSingle_32f(const Ipp32f* pSrc, const Ipp32f* pMean,
const Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);
```

```
IppStatus ippsLogGaussSingle_32f64f(const Ipp32f* pSrc, const Ipp32f* pMean,
const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```

```
IppStatus ippsLogGaussSingle_64f(const Ipp64f* pSrc, const Ipp64f* pMean,
const Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```

```
IppStatus ippsLogGaussSingle_Low_16s32s_Sfs(const Ipp16s* pSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val, int
scaleFactor);
```

```
IppStatus ippsLogGaussSingle_LowScaled_16s32f(const Ipp16s* pSrc, const
Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val, int
scaleFactor);
```

### **Case 2: Operation for the diagonal covariance matrix**

```
IppStatus ippsLogGaussSingle_DirectVar_16s32s_Sfs(const Ipp16s* pSrc, const
Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32s* pResult, Ipp32s val, int
scaleFactor);
```

```
IppStatus ippsLogGaussSingle_DirectVarScaled_16s32f(const Ipp16s* pSrc, const
Ipp16s* pMean, const Ipp16s* pVar, int len, Ipp32f* pResult, Ipp32f val, int
scaleFactor);
```

```
IppStatus ippsLogGaussSingle_DirectVar_32f(const Ipp32f* pSrc, const Ipp32f*
pMean, const Ipp32f* pVar, int len, Ipp32f* pResult, Ipp32f val);
```

```
IppStatus ippsLogGaussSingle_DirectVar_32f64f(const Ipp32f* pSrc, const
Ipp32f* pMean, const Ipp32f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```

```
IppStatus ippsLogGaussSingle_DirectVar_64f(const Ipp64f* pSrc, const Ipp64f*
pMean, const Ipp64f* pVar, int len, Ipp64f* pResult, Ipp64f val);
```

### **Case 3: Operation for the identity covariance matrix**

```
IppStatus ippsLogGaussSingle_IdVar_16s32s_Sfs(const Ipp16s* pSrc, const
Ipp16s* pMean, int len, Ipp32s* pResult, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGaussSingle_IdVarScaled_16s32f(const Ipp16s* pSrc, const
Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int scaleFactor);
```

```
IppStatus ippsLogGaussSingle_IdVar_32f(const Ipp32f* pSrc, const Ipp32f*
pMean, int len, Ipp32f* pResult, Ipp32f val);
```

```
IppStatus ippsLogGaussSingle_IdVar_32f64f(const Ipp32f* pSrc, const Ipp32f*
pMean, int len, Ipp64f* pResult, Ipp64f val);
```

```
IppStatus ippsLogGaussSingle_IdVar_64f(const Ipp64f* pSrc, const Ipp64f*
pMean, int len, Ipp64f* pResult, Ipp64f val);
```

```
IppStatus ippsLogGaussSingle_IdVarLow_16s32s_Sfs(const Ipp16s* pSrc, const
Ipp16s* pMean, int len, Ipp32s* pResult, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGaussSingle_IdVarLowScaled_16s32f(const Ipp16s* pSrc, const
Ipp16s* pMean, int len, Ipp32f* pResult, Ipp32f val, int scaleFactor);
```

#### Case 4: Operation for the block diagonal covariance matrix

```
IppStatus ippsLogGaussSingle_BlockDVar_16s32s_Sfs(const Ipp16s* pSrc, const
Ipp16s* pMean, const IppsBlockDMatrix_16s* pBlockVar, int len, Ipp32s*
pResult, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGaussSingle_BlockDVarScaled_16s32f(const Ipp16s* pSrc, const
Ipp16s* pMean, const IppsBlockDMatrix_16s* pBlockVar, int len, Ipp32f*
pResult, Ipp32f val, int scaleFactor);
```

```
IppStatus ippsLogGaussSingle_BlockDVar_32f(const Ipp32f* pSrc, const Ipp32f*
pMean, const IppsBlockDMatrix_32f* pBlockVar, int len, Ipp32f* pResult,
Ipp32f val);
```

```
IppStatus ippsLogGaussSingle_BlockDVar_32f64f(const Ipp32f* pSrc, const
Ipp32f* pMean, const IppsBlockDMatrix_32f* pBlockVar, int len, Ipp64f*
pResult, Ipp64f val);
```

```
IppStatus ippsLogGaussSingle_BlockDVar_64f(const Ipp64f* pSrc, const Ipp64f*
pMean, const IppsBlockDMatrix_64f* pBlockVar, int len, Ipp64f* pResult,
Ipp64f val);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector .
<i>pMean</i>	Pointer to the mean vector.
<i>pVar</i>	Pointer to the variance vector .
<i>pBlockVar</i>	Pointer to the block diagonal variance matrix.
<i>len</i>	Number of elements in the input, mean, and variance vectors.
<i>pResult</i>	Pointer to the result.
<i>val</i>	Gaussian constant.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

### Description

The function `ippsLogGaussSingle` is declared in the `ippsr.h` file. This function calculates the observation probability for a single observation vector and a Gaussian mixture component. The result *pResult* is in the logarithmic representation.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value (for <code>_DirectVar</code> function flavors only).

## LogGauss

*Calculates the observation probability for a single Gaussian with multiple observation vectors.*

---

### Syntax

#### Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGauss_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height,
Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGauss_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val,
int scaleFactor);
```

```
IppStatus ippsLogGauss_Scaled_16s32f_D2(const Ipp16s* pSrc, int step, const
Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
Ipp32f val, int scaleFactor);
```

```
IppStatus ippsLogGauss_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val,
int scaleFactor);
```

```
IppStatus ippsLogGauss_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
pMean, const Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);
```



```

IppStatus ippsLogGauss_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
const Ipp32f* pVar, int width, Ipp32f* pDst, int height, Ipp32f val);

IppStatus ippsLogGauss_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
pMean, const Ipp64f* pVar, int width, Ipp64f* pDst, int height, Ipp64f val);

IppStatus ippsLogGauss_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
const Ipp64f* pVar, int width, Ipp64f* pDst, int height, Ipp64f val);

IppStatus ippsLogGauss_Low_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height,
Ipp32s val, int scaleFactor);

IppStatus ippsLogGauss_Low_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int width, Ipp32s* pDst, int height, Ipp32s val,
int scaleFactor);

IppStatus ippsLogGauss_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
Ipp32f val, int scaleFactor);

IppStatus ippsLogGauss_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height, Ipp32f val,
int scaleFactor);

```

### Case 2: Operation for the identity covariance matrix

```

IppStatus ippsLogGauss_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const
Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int
scaleFactor);

IppStatus ippsLogGauss_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int scaleFactor);

IppStatus ippsLogGauss_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int
scaleFactor);

IppStatus ippsLogGauss_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int
scaleFactor);

IppStatus ippsLogGauss_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
Ipp32f* pMean, int width, Ipp32f* pDst, int height, Ipp32f val);

```

```
IppStatus ippsLogGauss_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean, int width, Ipp32f* pDst, int height, Ipp32f val);
```

```
IppStatus ippsLogGauss_IdVar_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f* pMean, int width, Ipp64f* pDst, int height, Ipp64f val);
```

```
IppStatus ippsLogGauss_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean, int width, Ipp64f* pDst, int height, Ipp64f val);
```

```
IppStatus ippsLogGauss_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int step, const Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGauss_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s* pMean, int width, Ipp32s* pDst, int height, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int step, const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int scaleFactor);
```

```
IppStatus ippsLogGauss_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s* pMean, int width, Ipp32f* pDst, int height, Ipp32f val, int scaleFactor);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> * <i>step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pMean</i>	Pointer to the mean vector [ <i>width</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>width</i> ].
<i>width</i>	Length of the mean and variance vectors.
<i>pDst</i>	Pointer to the result vector [ <i>height</i> ].
<i>height</i>	Number of rows in the input matrix, also the length of the result vector <i>pDst</i> .
<i>val</i>	Gaussian constant.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

## Description

The function `ippsLogGauss` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors for a Gaussian mixture component. The result *pResult* is in the logarithmic representation.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## LogGaussMultiMix

*Calculates the observation probability for multiple Gaussian mixture components.*

---

### Syntax

```
IppStatus ippsLogGaussMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean, const Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32f* pSrcDst, int height, int scaleFactor);
```

```

IppStatus ippsLogGaussMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
pVar, int step, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
mVar, const Ipp32f* pSrc, int width, Ipp32f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
pVar, int step, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
mVar, const Ipp64f* pSrc, int width, Ipp64f* pSrcDst, int height);

IppStatus ippsLogGaussMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean, const
Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int
height, int scaleFactor);

IppStatus ippsLogGaussMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean, const
Ipp16s** mVar, const Ipp16s* pSrc, int width, Ipp32s* pSrcDst, int height,
int scaleFactor);

IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2(const Ipp16s* pSrc, int
step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int
height, int scaleFactor);

IppStatus ippsLogGaussMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mSrc,
const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pDst, int height,
int scaleFactor);

```

## Parameters

<i>pMean</i>	Pointer to the mean vector [ <i>height</i> * <i>step</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>height</i> * <i>step</i> ].
<i>mMean</i>	Pointer to the mean matrix [ <i>height</i> ][ <i>width</i> ].
<i>mVar</i>	Pointer to the variance matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in the mean and variance vectors.
<i>pSrc</i>	Pointer to the input vector [ <i>width</i> ].
<i>pSrcDst</i>	Pointer to the input and destination vector [ <i>height</i> ].
<i>width</i>	Number of columns in the mean matrix <i>mMean</i> .
<i>height</i>	Number of rows in the mean matrix <i>mMean</i> .
<i>scaleFactor</i>	Scaling factor for intermediate sums.

## Description

The function `ippsLogGaussMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components. The results are in the logarithmic representation.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## LogGaussMax

*Calculates the likelihood probability given multiple observations and a Gaussian mixture component, using the maximum operation.*

---

### Syntax

#### Case 1: Operation for the inverse diagonal covariance matrix

```
IpplStatus ippsLogGaussMax_16s32s_D2Sfs(const Ippl6s* pSrc, int step, const
Ippl6s* pMean, const Ippl6s* pVar, int width, Ipp32s* pSrcDst, int height,
Ipp32s val, int scaleFactor);
```

```
IpplStatus ippsLogGaussMax_16s32s_D2LSfs(const Ippl6s** mSrc, const Ippl6s*
pMean, const Ippl6s* pVar, int width, Ipp32s* pSrcDst, int height, Ipp32s
val, int scaleFactor);
```

```
IpplStatus ippsLogGaussMax_Scaled_16s32f_D2(const Ippl6s* pSrc, int step,
const Ippl6s* pMean, const Ippl6s* pVar, int width, Ipp32f* pDst, int height,
Ipp32f val, int scaleFactor);
```

```
IppStatus ippsLogGaussMax_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f
val, int scaleFactor);
```

```
IppStatus ippsLogGaussMax_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f*
pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f
val);
```

```
IppStatus ippsLogGaussMax_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);
```

```
IppStatus ippsLogGaussMax_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f
val);
```

```
IppStatus ippsLogGaussMax_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);
```

```
IppStatus ippsLogGaussMax_Low_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int
height, Ipp32s val, int scaleFactor);
```

```
IppStatus ippsLogGaussMax_Low_16s32s_D2LSfs(const Ipp16s** mSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height, Ipp32s
val, int scaleFactor);
```

```
IppStatus ippsLogGaussMax_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
height, Ipp32f val, int scaleFactor);
```

```
IppStatus ippsLogGaussMax_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
Ipp32f val, int scaleFactor);
```

### **Case 2: Operation for the identity covariance matrix**

```
IppStatus ippsLogGaussMax_IdVar_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s val, int
scaleFactor);
```

```
IppStatus ippsLogGaussMax_IdVar_16s32s_D2LSfs(const Ipp16s** mSrc, const
Ipp16s* pMean, int width, Ipp32s* pSrcDst, int height, Ipp32s val, int
scaleFactor);
```

```

IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
scaleFactor);

IppStatus ippsLogGaussMax_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
scaleFactor);

IppStatus ippsLogGaussMax_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussMax_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussMax_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussMax_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2Sfs(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int
height, Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussMax_IdVarLow_16s32s_D2LSfs(const Ipp16s** mSrc, const
Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32s* pSrcDst, int height,
Ipp32s val, int scaleFactor);

IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
step, const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst,
int height, Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussMax_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc,
const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
height, Ipp32f val, int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> * <i>step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pMean</i>	Pointer to the mean vector [ <i>width</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>width</i> ].

<i>width</i>	Length of the mean and variance vectors.
<i>pSrcDst</i>	Pointer to the likelihood vector [ <i>height</i> ].
<i>height</i>	Length of the vector <i>pSrcDst</i> .
<i>val</i>	Gaussian constant.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

## Description

The function `ippsLogGaussMax` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector *pSrcDst* using the “maximum” operation.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## LogGaussMaxMultiMix

*Calculate the likelihood probability for multiple Gaussian mixture components, using the maximum operation.*

---

### Syntax

```
IppStatus ippsLogGaussMaxMultiMix_16s32s_D2Sfs(const Ipp16s* pMean, const
Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
Ipp32s* pSrcDst, int height, int scaleFactor);
```



```

IppStatus ippsLogGaussMaxMultiMix_16s32s_D2LSfs(const Ipp16s** mMean, const
Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal, Ipp32s*
pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
Ipp32f* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean,
const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
Ipp32f* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f*
pSrcDst, int height);

IppStatus ippsLogGaussMaxMultiMix_32f_D2L(const Ipp32f* pMean, const Ipp32f**
mVar, const Ipp32f* pSrc, int width, const Ipp32f* pVal, Ipp32f* pSrcDst,
int height);

IppStatus ippsLogGaussMaxMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f*
pSrcDst, int height);

IppStatus ippsLogGaussMaxMultiMix_64f_D2L(const Ipp64f* pMean, const Ipp64f**
mVar, const Ipp64f* pSrc, int width, const Ipp64f* pVal, Ipp64f* pSrcDst,
int height);

IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2Sfs(const Ipp16s* pMean, const
Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
Ipp32s* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_Low_16s32s_D2LSfs(const Ipp16s** mMean,
const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32s* pVal,
Ipp32s* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2(const Ipp16s* pMean,
const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f*
pVal, Ipp32f* pSrcDst, int height, int scaleFactor);

IppStatus ippsLogGaussMaxMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mMean,
const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
Ipp32f* pSrcDst, int height, int scaleFactor);

```

## Parameters

<i>pMean</i>	Pointer to the mean vector [ <i>height</i> * <i>step</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>height</i> * <i>step</i> ].
<i>mMean</i>	Pointer to the mean matrix [ <i>height</i> ][ <i>width</i> ].
<i>mVar</i>	Pointer to the variance matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in the mean and variance vectors.
<i>pSrc</i>	Pointer to the input vector [ <i>width</i> ].
<i>width</i>	Number of columns in the mean and variance matrices.
<i>pVal</i>	Pointer to the weight constant vector [ <i>height</i> ].
<i>pSrcDst</i>	Pointer to the likelihood vector [ <i>height</i> ].
<i>height</i>	Number of rows in the mean and variance matrices.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

## Description

The function `ippsLogGaussMaxMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst* using the “maximum” operation. The covariance matrix is assumed to be inverse diagonal.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## LogGaussAdd

*Calculates the likelihood probability for multiple observation vectors.*

---

### Syntax

#### Case 1: Operation for the inverse diagonal covariance matrix

```

IppStatus ippsLogGaussAdd_Scaled_16s32f_D2(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
height, Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussAdd_Scaled_16s32f_D2L(const Ipp16s** mSrc, const Ipp16s*
pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f
val, int scaleFactor);

IppStatus ippsLogGaussAdd_32f_D2(const Ipp32f** pSrc, int step, const Ipp32f*
pMean, const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32fval);

IppStatus ippsLogGaussAdd_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pMean,
const Ipp32f* pVar, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussAdd_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f*
pMean, const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f
val);

IppStatus ippsLogGaussAdd_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pMean,
const Ipp64f* pVar, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int
height, Ipp32f val, int scaleFactor);

IppStatus ippsLogGaussAdd_LowScaled_16s32f_D2L(const Ipp16s** mSrc, const
Ipp16s* pMean, const Ipp16s* pVar, int width, Ipp32f* pSrcDst, int height,
Ipp32f val, int scaleFactor);

```

#### Case 2: Operation for the identity covariance matrix

```

IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, int step,
const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
scaleFactor);

```

```

IppStatus ippsLogGaussAdd_IdVarScaled_16s32f_D2L(const Ipp16s** mSrc, const
Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
scaleFactor);

IppStatus ippsLogGaussAdd_IdVar_32f_D2(const Ipp32f* pSrc, int step, const
Ipp32f* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussAdd_IdVar_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val);

IppStatus ippsLogGaussAdd_IdVar_64f_D2(const Ipp64f* pSrc, int step, const
Ipp64f* pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussAdd_IdVar_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
pMean, int width, Ipp64f* pSrcDst, int height, Ipp64f val);

IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc, int
step, const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f
val, int scaleFactor);

IppStatus ippsLogGaussAdd_IdVarLowScaled_16s32f_D2L(const Ipp16s** mSrc,
const Ipp16s* pMean, int width, Ipp32f* pSrcDst, int height, Ipp32f val, int
scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> * <i>step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pMean</i>	Pointer to the mean vector [ <i>width</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>width</i> ].
<i>width</i>	Length of the mean and variance vectors.
<i>pSrcDst</i>	Pointer to the likelihood vector [ <i>height</i> ].
<i>height</i>	Number of observation vectors.
<i>val</i>	Weight constant.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

## Description

The function `ippsLogGaussAdd` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple observation vectors and accumulates the resulting probabilities in the vector `pSrcDst`.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## LogGaussAddMultiMix

*Calculates the likelihood probability for multiple Gaussian mixture components.*

---

### Syntax

```
IppStatus ippsLogGaussAddMultiMix_Scaled_16s32f_D2(const Ipp16s* pMean, const
Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f pVal,
Ipp32f* pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussAddMultiMix_Scaled_16s32f_D2L(const Ipp16s** mMean,
const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f pVal, Ipp32f*
pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussAddMultiMix_32f_D2(const Ipp32f* pMean, const Ipp32f*
pVar, int step, const Ipp32f* pSrc, int width, const Ipp32f pVal, Ipp32f*
pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussAddMultiMix_32f_D2L(const Ipp32f** mMean, const Ipp32f**
mVar, const Ipp32f* pSrc, int width, const Ipp32f pVal, Ipp32f* pSrcDst, int
height, int scaleFactor);
```

```
IppStatus ippsLogGaussAddMultiMix_64f_D2(const Ipp64f* pMean, const Ipp64f*
pVar, int step, const Ipp64f* pSrc, int width, const Ipp64f pVal, Ipp64f*
pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussAddMultiMix_64f_D2L(const Ipp64f** mMean, const Ipp64f**
mVar, const Ipp64f* pSrc, int width, const Ipp64f pVal, Ipp64f* pSrcDst, int
height, int scaleFactor);
```

```
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2(const Ipp16s* pMean,
const Ipp16s* pVar, int step, const Ipp16s* pSrc, int width, const Ipp32f*
pVal, Ipp32f* pSrcDst, int height, int scaleFactor);
```

```
IppStatus ippsLogGaussAddMultiMix_LowScaled_16s32f_D2L(const Ipp16s** mMean,
const Ipp16s** mVar, const Ipp16s* pSrc, int width, const Ipp32f* pVal,
Ipp32f* pSrcDst, int height, int scaleFactor);
```

## Parameters

<i>pMean</i>	Pointer to the mean vector [ <i>height</i> * <i>step</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>height</i> * <i>step</i> ].
<i>mMean</i>	Pointer to the mean matrix [ <i>height</i> ][ <i>width</i> ].
<i>mVar</i>	Pointer to the variance matrix [ <i>height</i> ][ <i>width</i> ].
<i>step</i>	Row step in the mean and variance vectors.
<i>pSrc</i>	Pointer to the input vector [ <i>width</i> ].
<i>width</i>	Number of columns in the mean and variance matrices.
<i>pVal</i>	Pointer to the weight constant vector [ <i>height</i> ].
<i>pSrcDst</i>	Pointer to the likelihood vector [ <i>height</i> ].
<i>height</i>	Number of Gaussian mixture components.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

## Description

The function `ippsLogGaussAddMultiMix` is declared in the `ippsr.h` file. This function calculates the observation probability for multiple Gaussian mixture components and accumulates the resulting probabilities in the vector *pSrcDst*. The covariance matrix is assumed to be inverse diagonal.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## LogGaussMixture

*Calculates the likelihood probability for the Gaussian mixture.*

---

### Syntax

#### Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMixture_Scaled_16s32f_D2(const Ipp16s* pSrc, const
Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int width, const
Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_Scaled_16s32f_D2L(const Ipp16s* pSrc, const
Ipp16s** mMean, const Ipp16s** mVar, int height, int width, const Ipp32f*
pVal, Ipp32f* pResult, int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2(const Ipp16s* pSrc, const
Ipp16s* pMean, const Ipp16s* pVar, int height, int step, int width, const
Ipp32f* pVal, Ipp32f* pResult, int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_LowScaled_16s32f_D2L(const Ipp16s* pSrc, const
Ipp16s** mMean, const Ipp16s** mVar, int height, int width, const Ipp32f*
pVal, Ipp32f* pResult, int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_32f_D2(const Ipp32f* pSrc, const Ipp32f* pMean,
const Ipp32f* pVar, int height, int step, int width, const Ipp32f* pVal,
Ipp32f* pResult);
```

```
IppStatus ippsLogGaussMixture_32f_D2L(const Ipp32f* pSrc, const Ipp32f**
mMean, const Ipp32f** mVar, int height, int width, const Ipp32f* pVal, Ipp32f*
pResult);
```

```
IppStatus ippsLogGaussMixture_64f_D2(const Ipp64f* pSrc, const Ipp64f* pMean,
const Ipp64f* pVar, int height, int step, int width, const Ipp64f* pVal,
Ipp64f* pResult);
```

```
IppStatus ippsLogGaussMixture_64f_D2L(const Ipp64f* pSrc, const Ipp64f**  
mMean, const Ipp64f** mVar, int height, int width, const Ipp64f* pVal, Ipp64f*  
pResult);
```

### Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2(const Ipp16s* pSrc, const
Ipp16s* pMean, int height, int step, int width, const Ipp32f* pVal, Ipp32f*
pResult, int scaleFactor);
```

```

IppStatus ippsLogGaussMixture_IdVarScaled_16s32f_D2L(const Ipp16s* pSrc,
const Ipp16s** mMean, int height, int width, const Ipp32f* pVal, Ipp32f*
pResult, int scaleFactor);

```

```
IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2(const Ipp16s* pSrc,
const Ipp16s* pMean, int height, int step, int width, const Ipp32f* pVal,
Ipp32f* pResult, int scaleFactor);
```

```

IppStatus ippsLogGaussMixture_IdVarLowScaled_16s32f_D2L(const Ipp16s* pSrc,
const Ipp16s** mMean, int height, int width, const Ipp32f* pVal, Ipp32f*
pResult, int scaleFactor);

```

```

IppStatus ippsLogGaussMixture_IdVar_32f_D2(const Ipp32f* pSrc, const Ipp32f*
pMean, int height, int step, int width, const Ipp32f* pVal, Ipp32f* pResult);

```

```

IppStatus ippsLogGaussMixture_IdVar_32f_D2L(const Ipp32f* pSrc, const Ipp32f**
mMean, int height, int width, const Ipp32f* pVal, Ipp32f* pResult);

```

```

IppStatus ippsLogGaussMixture_IdVar_64f_D2(const Ipp64f* pSrc, const Ipp64f*
pMean, int height, int step, int width, const Ipp64f* pVal, Ipp64f* pResult);

```

```
IppStatus ippsLogGaussMixture_IdVar_64f_D2L(const Ipp64f* pSrc, const Ipp64f**  
mMean, int height, int width, const Ipp64f* pVal, Ipp64f* pResult);
```

## Parameters

*pSrc* Pointer to the input vector [*width*].



<i>pMean</i>	Pointer to the mean vector [ <i>height</i> * <i>step</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>height</i> * <i>step</i> ].
<i>mMean</i>	Pointer to the mean matrix [ <i>height</i> ][ <i>width</i> ].
<i>mVar</i>	Pointer to the variance matrix [ <i>height</i> ][ <i>width</i> ].
<i>height</i>	Number of Gaussian mixture components.
<i>step</i>	Row step in mean and variance vectors (in <i>pMean</i> elements).
<i>width</i>	Length of the mean and variance matrix rows; also length of the <i>pSrc</i> vector.
<i>pVal</i>	Pointer to the vector of weight constants [ <i>height</i> ].
<i>pResult</i>	Pointer to the output mixture value.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

## Description

The function `ippsLogGaussMixture` is declared in the `ippsr.h` file. This function calculates observation probability for the Gaussian mixture and returns it in *pResult*.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

Functions with `Low` suffix in their name are intended for low input values and provide faster calculations. They yield the correct result if input 16-bit data are represented by 12 bits and input vectors are of length less than 128.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## LogGaussMixture\_Select

*Calculates the likelihood probability for the Gaussian mixture using Gaussian selection.*

---

### Syntax

#### Case 1: Operation for the inverse diagonal covariance matrix

```
IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2(const Ipp16s* pSrc,
const Ipp16s* pMean, const Ipp16s* pVar, int step, int width, const Ipp32s*
pVal, const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f
none, int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_SelectScaled_16s32f_D2L(const Ipp16s** mSrc,
const Ipp16s** mMean, const Ipp16s** mVar, int width, const Ipp32s* pVal,
const Ipp8u* pSign, int height, Ipp32s* pResult, int frames, Ipp32f none,
int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_Select_32f_D2(const Ipp32f* pSrc, const Ipp32f*
pMean, const Ipp32f* pVar, int step, int width, const Ipp32f* pVal, const
Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);
```

```
IppStatus ippsLogGaussMixture_Select_32f_D2L(const Ipp32f** mSrc, const
Ipp32f** mMean, const Ipp32f** mVar, int width, const Ipp32f* pVal, const
Ipp8u* pSign, int height, Ipp32f* pResult, int frames, Ipp32f none);
```

#### Case 2: Operation for the identity covariance matrix

```
IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2(const Ipp16s* pSrc,
const Ipp16s* pMean, int step, int width, const Ipp32s* pVal, const Ipp8u*
pSign, int height, Ipp32s* pResult, int frames, Ipp32f none, int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_SelectIdVarScaled_16s32f_D2L(const Ipp16s**
mSrc, const Ipp16s** mMean, int width, const Ipp32s* pVal, const Ipp8u*
pSign, int height, Ipp32s* pResult, int frames, Ipp32f none, int scaleFactor);
```

```
IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2(const Ipp32f* pSrc, const
Ipp32f* pMean, int step, int width, const Ipp32f* pVal, const Ipp8u* pSign,
int height, Ipp32f* pResult, int frames, Ipp32f none);
```

```
IppStatus ippsLogGaussMixture_SelectIdVar_32f_D2L(const Ipp32f** mSrc, const
Ipp32f** mMean, int width, const Ipp32f* pVal, const Ipp8u* pSign, int height,
Ipp32f* pResult, int frames, Ipp32f none);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>frames*step</i> ].
<i>pMean</i>	Pointer to the mean vector [ <i>height*step</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>height*step</i> ].
<i>mSrc</i>	Pointer to the input vector [ <i>frames*width</i> ].
<i>mMean</i>	Pointer to the mean matrix [ <i>height</i> ][ <i>width</i> ].
<i>mVar</i>	Pointer to the variance matrix [ <i>height</i> ][ <i>width</i> ].
<i>pVal</i>	Pointer to the vector of weight constants [ <i>height</i> ].
<i>height</i>	Number of Gaussian mixture components.
<i>pSign</i>	Pointer to the vector of Gaussian calculation signs [ <i>frames*(height+7)/8</i> ] (in bytes).
<i>step</i>	Row step in mean, variance and input vectors (in <i>pMean</i> elements).
<i>width</i>	Length of the mean, variance and input matrix rows.
<i>pResult</i>	Pointer to the vector of output mixture values [ <i>frames</i> ] .
<i>frames</i>	Number of input vectors; also the length of <i>pSign</i> row.
<i>none</i>	Result value if no Gaussian is calculated.
<i>scaleFactor</i>	Scaling factor for intermediate sums.

## Description

The function `ippsLogGaussMixture_Select` is declared in the `ippsr.h` file. This function calculates observation probabilities for the Gaussian mixture and *frames* input vectors and returns them in *pResult* vector. Gaussians with corresponding signs equal to zero are not calculated. If no Gaussian is calculated for the input vector, *none* value is returned for this vector.

Function flavors that perform integer scaling apply the multiplier  $2^{-scaleFactor}$  to intermediate sums, rather than to output results.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>height</i> or <i>frames</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .
<code>ippStsNoGaussian</code>	Indicates a warning when no Gaussian is calculated for one of the input vectors.

## BuildSignTable

*Fills sign table for Gaussian mixture calculation.*

---

### Syntax

```
IppStatus ippsBuildSignTable_8ulu(const Ipp32s* pIndx, int num, const Ipp8u** mShortlist, int clust, int width, int shift, Ipp8u* pSign, int frames, int comps);
```

```
IppStatus ippsBuildSignTable_Var_8ulu(const Ipp32s* pIndx, const int* pNum, const Ipp8u** mShortlist, int clust, int width, int shift, Ipp8u* pSign, int frames, int comps);
```

### Parameters

<i>pIndx</i>	Pointer to the cluster indexes vector ( $[frames * num]$ or sum of <i>pNum</i> elements).
<i>num</i>	Number of clusters for each input vector.
<i>pNum</i>	Pointer to the number of clusters vector [ <i>frames</i> ].
<i>mShortlist</i>	Pointer to the shortlist matrix [ <i>clust</i> * <i>width</i> ] (in bytes).
<i>clust</i>	Number of rows in shortlist vector (equal to the codebook size).
<i>width</i>	Row length in shortlist matrix in bytes.
<i>shift</i>	First element displacement in shortlist vector rows (in bits).
<i>pSign</i>	Pointer to the output vector of Gaussian calculation signs $[frames * (comps + 7) / 8]$ (in bytes).
<i>frames</i>	Number of input vectors (rows in the vector <i>pSign</i> ).
<i>comps</i>	Number of Gaussian mixture components (bit columns in the vector <i>pSign</i> ).

## Description

The functions `ippsBuildSignTable` are declared in the `ippsr.h` file. These functions build the sign table for Gaussian mixture calculation using Gaussian selection technique. The sign table is used as the input argument for `ippsLogGaussMixture_Select` functions. The input vectors can activate one or several codebook clusters. Shortlist table rows correspond to clusters, and its columns – to Gaussians. Row substring of length `clust` indicates if the cluster is active for a mixture component.

If  $c_k$  is the bit substring of the  $k$ -th row of `mShortlist`, consisting of bits `shift`,... `shift+comps-1`, for  $k = 0, \dots, clust - 1$ .

Then elements  $l(i), \dots, l(i+1)-1$  of the `pIndx` vector are active cluster numbers for  $i$ -th input vector. The  $i$ -th row of `pSign` is set to the logical sum of substrings

$c_{l(i)}, \dots, c_{l(i+1)-1}$ ,  $i = 0, \dots, frames - 1$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>clust</code> , <code>width</code> , <code>frames</code> , <code>comps</code> , <code>num</code> , or one of <code>pNum</code> vector elements is less than or equal to 0, or when <code>shift</code> is less than 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>width</code> is less than $(shift + comps + 7) / 8$ .
<code>ippStsBadArgErr</code>	Indicates an error when one of <code>pIndx</code> vector elements is less than 0, or greater than or equal to <code>clust</code> .

## FillShortlist\_Row

Fills row-wise shortlist table for Gaussian selection.

### Syntax

```

IppStatus ippsFillShortlist_Row_lu(const Ipp32s* pIndx, int height, int num,
Ipp8u** mShortlist, int clust, int width, int shift);

IppStatus ippsFillShortlist_RowVar_lu(const Ipp32s* pIndx, const int* pNum,
int height, Ipp8u** mShortlist, int clust, int width, int shift);

```

## Parameters

<i>pIndx</i>	Pointer to the cluster indexes vector ( $[num*height]$ or sum of <i>pNum</i> elements).
<i>height</i>	Number of Gaussian mixture components.
<i>num</i>	Number of clusters for each Gaussian mean.
<i>pNum</i>	Pointer to the number of clusters vector [ <i>height</i> ].
<i>mShortlist</i>	Pointer to the shortlist matrix [ <i>clust*width</i> ] (in bytes).
<i>clust</i>	Number of rows in shortlist matrix (equal to the codebook size).
<i>width</i>	Row length in shortlist matrix in bytes.
<i>shift</i>	First element displacement in shortlist vector rows (in bits).

## Description

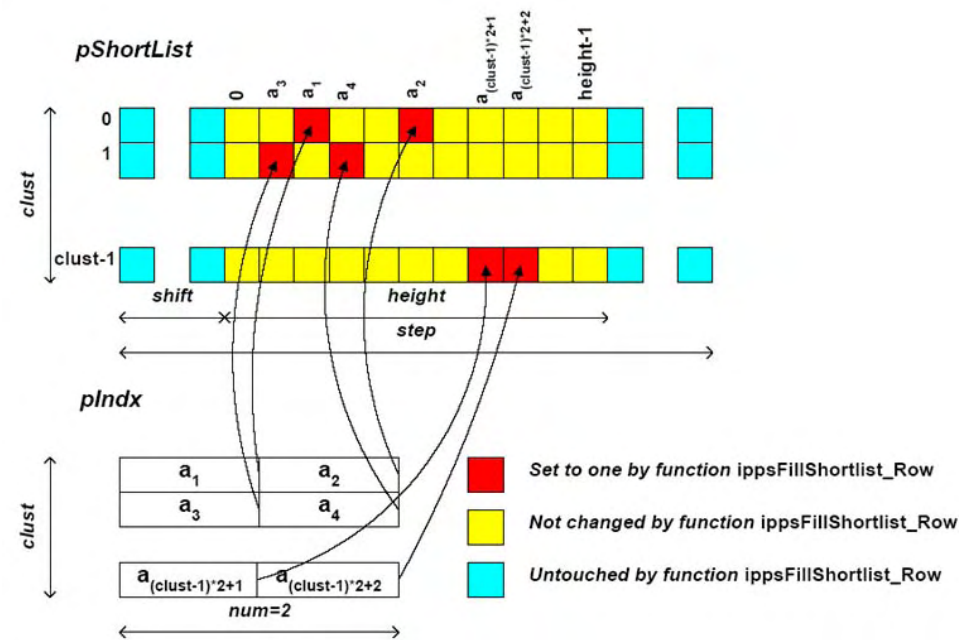
The function `ippsFillShortlist_Rowis` declared in the `ippsr.h` file. The function fills the shortlist table and provides that this table contains at least one Gaussian of the mixture for each cluster. This is done by quantizing of cluster centroids on codebook made of Gaussian mixture means. Quantization results are placed to *pIndx* vector.

Bits in the shortlist table *mShortlist* that correspond to one or more cluster centroids nearest to Gaussian mixture means are set to 1. The shortlist is the bit table with elements indicating whether the *i*-th Gaussian should be calculated if the input vector activates the *k*-th cluster of the codebook. The number of rows is equal to the codebook size, the number of columns is equal to the overall number of Gaussians in the model. Row step is chosen so as to start each row from the byte boundary.

The shortlist table must be zeroed after definition. Components of mixtures have sequential indexes. The *shift* argument is the index of the first mixture component in the model, and the *height* argument is the number of the components in the mixture.

Figure 8-3 illustrates the execution of this function.

**Figure 8-3 Execution of `ippsFillShortlist_Row` function for  $num=2$**



### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>clust</code> , <code>width</code> , <code>height</code> , <code>num</code> , or one of <code>pNum</code> vector elements is less than or equal to 0, or when <code>shift</code> is less than 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>width</code> is less than $(shift+height+7)/8$ .
<code>ippStsBadArgErr</code>	Indicates an error when one of <code>pIdx</code> vector elements is less than 0, or greater than or equal to <code>height</code> .

## FillShortlist\_Column

*Fills column-wise shortlist table for Gaussian selection.*

---

### Syntax

```
IppStatus ippsFillShortlist_Column_lu(const Ipp32s* pIndx, int num, Ipp8u** mShortlist, int clust, int width, int shift, int height);
```

```
IppStatus ippsFillShortlist_ColumnVar_lu(const Ipp32s* pIndx, const int* pNum, Ipp8u** mShortlist, int clust, int width, int shift, int height);
```

### Parameters

<i>pIndx</i>	Pointer to the cluster indexes vector ( $[num*height]$ or sum of <i>pNum</i> elements).
<i>height</i>	Number of Gaussian mixture components.
<i>num</i>	Number of clusters for each Gaussian mean.
<i>pNum</i>	Pointer to the number of clusters vector [ <i>height</i> ].
<i>mShortlist</i>	Pointer to the shortlist matrix [ <i>clust*width</i> ] (in bytes).
<i>clust</i>	Number of rows in shortlist matrix (equal to the codebook size).
<i>width</i>	Row length in shortlist matrix in bytes.
<i>shift</i>	First element displacement in shortlist vector rows (in bits).

### Description

The function `ippsFillShortlist_Column` is declared in the `ippsr.h` file. This function fills the part of the shortlist table corresponding to the Gaussian mixture.

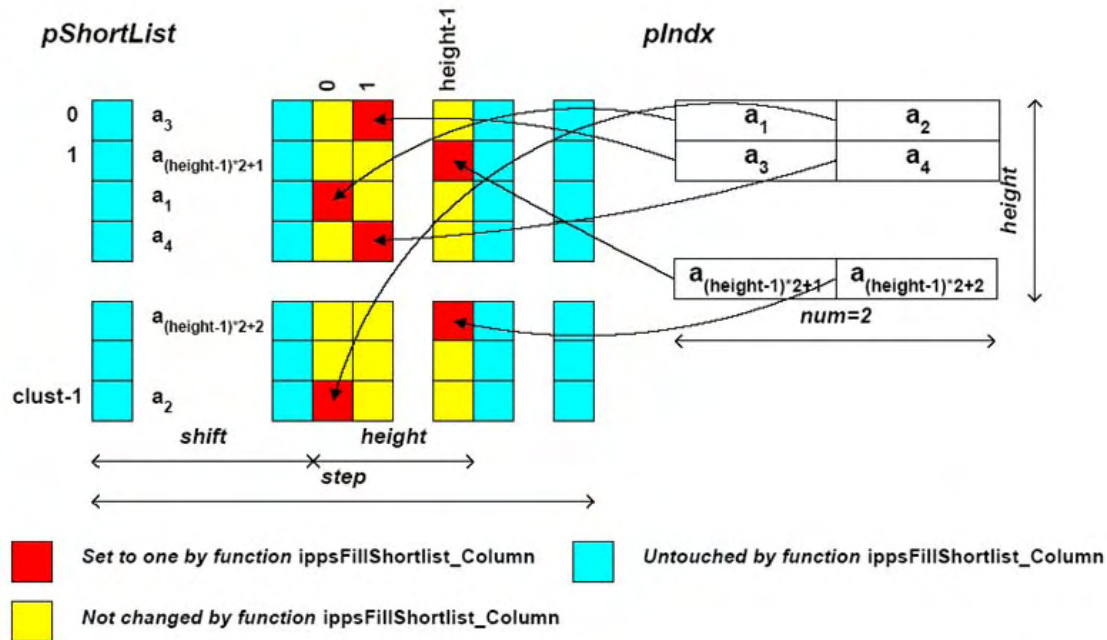
The shortlist is the bit table with elements indicating whether the *i*-th Gaussian must be calculated if the input vector activates the *k*-th cluster of the codebook. The number of rows is equal to the codebook size, the number of columns is equal to the overall number of Gaussians in the model. Row step is chosen so as to start each row from the byte boundary. The shortlist table should be zeroed after definition. Components of mixtures have sequential indexes. The *shift* argument is the index of the first mixture component in the model, and the *height* argument is the number of the components in the mixture.



After means of Gaussian mixture are quantized by one of `ippsVQSingle_Sort`, `ippsVQSingle_Thresh` functions, quantization results can be used to fill the shortlist table. Bits corresponding to active clusters for mixture components are set to 1.

Figure 8-4 illustartes the execution of this function.

**Figure 8-4 Execution of `ippsFillShortlist_Column` function for  $num=2$**



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>clust</code> , <code>width</code> , <code>height</code> , <code>num</code> , or one of <code>pNum</code> vector elements is less than or equal to 0, or when <code>shift</code> is less than 0.
<code>ippStsStrideErr</code>	Indicates an error when <code>width</code> is less than $(shift+height+7)/8$ .
<code>ippStsBadArgErr</code>	Indicates an error when one of <code>pIndx</code> vector elements is less than 0, or greater than or equal to <code>clust</code> .

## DTW

*Computes the distance between observation and reference vector sequences using Dynamic Time Warping algorithm.*

---

### Syntax

```

IppStatus ippsDTW_L2_8u32s_D2Sfs(const Ipp8u* pSrc1, int height1, const
Ipp8u* pSrc2, int height2, int width, int step, Ipp32s* pDist, int delta,
Ipp32s beam, int scaleFactor);

IppStatus ippsDTW_L2_8u32s_D2LSfs(const Ipp8u** mSrc1, int height1, const
Ipp8u** mSrc2, int height2, int width, Ipp32s* pDist, int delta, Ipp32s beam,
int scaleFactor);

IppStatus ippsDTW_L2Low_16s32s_D2Sfs(const Ipp16s* pSrc1, int height1, const
Ipp16s* pSrc2, int height2, int width, int step, Ipp32s* pDist, int delta,
Ipp32s beam, int scaleFactor);

IppStatus ippsDTW_L2Low_16s32s_D2LSfs(const Ipp16s** mSrc1, int height1,
const Ipp16s** mSrc2, int height2, int width, Ipp32s* pDist, int delta,
Ipp32s beam, int scaleFactor);

IppStatus ippsDTW_L2_32f_D2(const Ipp32f* pSrc1, int height1, const Ipp32f*
pSrc2, int height2, int width, int step, Ipp32f* pDist, int delta, Ipp32f
beam);

IppStatus ippsDTW_L2_32f_D2L(const Ipp32f** mSrc1, int height1, const Ipp32f**
mSrc2, int height2, int width, Ipp32f* pDist, int delta, Ipp32f beam);

```

### Parameters

<i>pSrc1</i>	Pointer to the first input (observation) vector [ <i>height1</i> * <i>step</i> ].
<i>pSrc2</i>	Pointer to the second input (reference) vector [ <i>height2</i> * <i>step</i> ].
<i>mSrc1</i>	Pointer to the first input (observation) matrix [ <i>height1</i> ][ <i>step</i> ].
<i>mSrc2</i>	Pointer to the second input (reference) matrix [ <i>height2</i> ][ <i>step</i> ].
<i>height1</i>	Number of rows in the first input matrix.

<i>height2</i>	Number of rows in the second input matrix.
<i>width</i>	Vector dimension. Equal to the length of the input matrices row .
<i>step</i>	Row step in <i>pSrc1</i> and <i>pSrc2</i> .
<i>pDist</i>	Pointer to the distance value.
<i>beam</i>	Beam value, used if positive.
<i>delta</i>	Endpoint constraint relaxation value.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsDTW` is declared in the `ippsr.h` file. This function calculates the distance between two vector sequences using the Dynamic Time Warping (DTW) principle.

Note that if  $height1 < height2$ , then *pSrc1* interchanges with *pSrc2*, that is, *pSrc1* becomes the reference vector and *pSrc2* is the test vector.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr I</code>	ndicates an error when <i>height1</i> , <i>height2</i> , or <i>width</i> is less than or equal to 0, or <i>delta</i> is less than 0 or greater than <i>height2</i> .
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .
<code>ippStsNoOperation</code>	Indicates that there are now admissible paths for <i>height1</i> , <i>height2</i> and <i>delta</i> values.

## Model Estimation

This section describes functions that are needed to estimate the parameters of the acoustic and language models.

Some of these functions are used in the Intel<sup>®</sup> IPP Speech Recognition Samples. See *Gaussian Calculations* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## MeanColumn

*Computes the mean values for the column elements.*

---

### Syntax

```

IppStatus ippsMeanColumn_16s_D2(const Ipp16s* pSrc, int height, int step,
Ipp16s* pDstMean, int width);

IppStatus ippsMeanColumn_16s_D2L(const Ipp16s** mSrc, int height, Ipp16s*
pDstMean, int width);

IppStatus ippsMeanColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
Ipp32f* pDstMean, int width);

IppStatus ippsMeanColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
pDstMean, int width);

```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height*step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>step</i>	Row step in the input vector (measured in <i>pSrc</i> elements).
<i>pDstMean</i>	Pointer to the output mean vector [ <i>width</i> ].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the output mean vector <i>pDstMean</i> .

### Description

The function `ippsMeanColumn` is declared in the `ippsr.h` file. This function calculates the mean values for the column elements of the input matrix.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.

`ippStsStrideErr` Indicates an error when *step* is less than *width*.

## VarColumn

Calculates the variances for the column elements.

### Syntax

```

IppStatus ippsVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int step,
Ipp16s* pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);

IppStatus ippsVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height, Ipp16s*
pSrcMean, Ipp16s* pDstVar, int width, int scaleFactor);

IppStatus ippsVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);

IppStatus ippsVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
pSrcMean, Ipp32f* pDstVar, int width);

```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height*step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pSrcMean</i>	Pointer to the input mean vector [ <i>width</i> ].
<i>pDstVar</i>	Pointer to the output variance vector [ <i>width</i> ].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the input mean vector <i>pSrcMean</i> and the output variance vector <i>pDstVar</i> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsVarColumn` is declared in the `ippsr.h` file. This function calculates the variances for the column elements of the matrix.

### Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## MeanVarColumn

*Calculates the means and variances for the column elements of a matrix.*

---

### Syntax

```

IppStatus ippsMeanVarColumn_16s_D2Sfs(const Ipp16s* pSrc, int height, int
step, Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);

IppStatus ippsMeanVarColumn_16s_D2LSfs(const Ipp16s** mSrc, int height,
Ipp16s* pDstMean, Ipp16s* pDstVar, int width, int scaleFactor);

IppStatus ippsMeanVarColumn_16s32s_D2Sfs(const Ipp16s* pSrc, int height, int
step, Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);

IppStatus ippsMeanVarColumn_16s32s_D2LSfs(const Ipp16s** mSrc, int height,
Ipp32s* pDstMean, Ipp32s* pDstVar, int width, int scaleFactor);

IppStatus ippsMeanVarColumn_32f_D2(const Ipp32f* pSrc, int height, int step,
Ipp32f* pDstMean, Ipp32f* pDstVar, int width);

IppStatus ippsMeanVarColumn_32f_D2L(const Ipp32f** mSrc, int height, Ipp32f*
pDstMean, Ipp32f* pDstVar, int width);

IppStatus ippsMeanVarColumn_16s16s32s_D2(const Ipp16s* pSrc, int height, int
step, Ipp16s* pDstMean, Ipp32s* pDstVar, int width);

IppStatus ippsMeanVarColumn_16s16s32s_D2L(const Ipp16s** mSrc, int height,
Ipp16s* pDstMean, Ipp32s* pDstVar, int width);

```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> * <i>step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>step</i>	Row step in the input vector <i>pSrc</i> .

<i>pDstcMean</i>	Pointer to the output mean vector [ <i>width</i> ].
<i>pDstVar</i>	Pointer to the output variance vector [ <i>width</i> ].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the output vectors <i>pSrcMean</i> and <i>pDstVar</i> .
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> , used only for <i>pDstVar</i> .

### Description

The function `ippsMeanVarColumn` is declared in the `ippsr.h` file. This function calculates both the means and the variances for the column elements of the input matrix.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## WeightedMeanColumn

*Computes the weighted mean values for the column elements.*

---

### Syntax

```
IpplStatus ippsWeightedMeanColumn_32f_D2(const Ipp32f* pSrc, int step, const Ipp32f* pWgt, int height, Ipp32f* pDstMean, int width);
```

```
IpplStatus ippsWeightedMeanColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f* pWgt, int height, Ipp32f* pDstMean, int width);
```

```
IpplStatus ippsWeightedMeanColumn_64f_D2(const Ipp64f* pSrc, int step, const Ipp64f* pWgt, int height, Ipp64f* pDstMean, int width);
```

```
IpplStatus ippsWeightedMeanColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f* pWgt, int height, Ipp64f* pDstMean, int width);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height</i> * <i>step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>pWgt</i>	Pointer to the weight vector [ <i>height</i> ].
<i>height</i>	Number of rows in the input matrix.
<i>step</i>	Row step in the input vector (measured in <i>pSrc</i> elements).
<i>pDstMean</i>	Pointer to the output mean vector [ <i>width</i> ].
<i>width</i>	Number of columns in the input matrix, and also the length of the output mean vector <i>pDstMean</i> .

## Description

The function `ippsWeightedMeanColumn` is declared in the `ippsr.h` file. This function calculates the weighted mean values for the column elements of the input matrix.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## WeightedVarColumn

*Computes the weighted variance values for the column elements.*

---

### Syntax

```

IppStatus ippsWeightedVarColumn_32f_D2(const Ipp32f* pSrc, int step, const
Ipp32f* pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);

IppStatus ippsWeightedVarColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
pWgt, int height, const Ipp32f* pSrcMean, Ipp32f* pDstVar, int width);

IppStatus ippsWeightedVarColumn_64f_D2(const Ipp64f* pSrc, int step, const
Ipp64f* pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar, int width);

```



---

```
IppStatus ippsWeightedVarColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
pWgt, int height, const Ipp64f* pSrcMean, Ipp64f* pDstVar, int width);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height*step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>pWgt</i>	Pointer to the weight vector [ <i>width</i> ].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pSrcMean</i>	Pointer to the input mean vector [ <i>width</i> ].
<i>pDstVar</i>	Pointer to the output variance vector [ <i>width</i> ].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the vectors <i>pSrcMean</i> and <i>pDstVar</i> .

### Description

The function `ippsWeightedVarColumn` is declared in the `ippsr.h` file. This function calculates the weighted variance values for the column elements of the input matrix.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## WeightedMeanVarColumn

*Computes weighted mean and variance values for the column elements.*

---

### Syntax

```
IppStatus ippsWeightedMeanVarColumn_32f_D2(const Ipp32f* pSrc, int step,
const Ipp32f* pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
```

```
IppStatus ippsWeightedMeanVarColumn_32f_D2L(const Ipp32f** mSrc, const Ipp32f*
pWgt, int height, Ipp32f* pDstMean, Ipp32f* pDstVar, int width);
```

```
IppStatus ippsWeightedMeanVarColumn_64f_D2(const Ipp64f* pSrc, int step,
const Ipp64f* pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int width);
```

```
IppStatus ippsWeightedMeanVarColumn_64f_D2L(const Ipp64f** mSrc, const Ipp64f*
pWgt, int height, Ipp64f* pDstMean, Ipp64f* pDstVar, int width);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector [ <i>height*step</i> ].
<i>mSrc</i>	Pointer to the input matrix [ <i>height</i> ][ <i>width</i> ].
<i>pWgt</i>	Pointer to the weight vector [ <i>width</i> ].
<i>height</i>	Number of rows in the input matrix <i>mSrc</i> .
<i>step</i>	Row step in the input vector <i>pSrc</i> .
<i>pDstMean</i>	Pointer to the output mean vector [ <i>width</i> ].
<i>pDstVar</i>	Pointer to the output variance vector [ <i>width</i> ].
<i>width</i>	Number of columns in the input matrix <i>mSrc</i> , and also the length of the output vectors <i>pDstMean</i> and <i>pDstVar</i> .

## Description

The function `ippsWeightedMeanVarColumn` is declared in the `ippsr.h` file. This function calculates both weighted means and weighted variances for the column elements of the input matrix.

Example 8-1 shows how the function `ippsWeightedMeanVarColumn` can be used.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

**Example 8-1 Weights, Means and Variances EM Re-Estimation**

```

/* Input:  int height;                // mixture components number
           int width;                 // observation space dimension
           int step;                  // row step for mean, var and obs (step>=width)
           int num;                   // observations number
           int step1;                 // row step for gamma (step1>=num)
           float obs [num*step]      // observation vectors
Update: float weight[height]         // Gaussian weights
           float mean[height*step]   // Gaussian mean vectors
           float var [height*step]   // Gaussian variance vectors
Output: float result;                // probability logarithms sum */
{
    float gamma [height*step1] // gamma matrix
    float gammaT [height]      // gamma sums vector
    int k; float sum, sumGamma;
    /* adjust determinants for probability calculation */
    ippsLn_32f_I(weight,height);
    for (k=0; k<height; k++) {
        ippsSumLn_32f(var+k*step,width,&sum);
        weight[k]+=0.5f*(sum-width*log(2.0*3.1415926));
    }
}

```

```

/* invert variances for probability calculation */
ippsDivCRev_32f_I(var,height*step);
/* logarithm of weighted Gaussian probabilities */
ippsLogGauss_32f_D2(obs,step,mean,var,width,gamma,num,weight[0]);
ippsCopy_32f(gamma,probs,num);
for (k=1; k<height; k++) {
    ippsLogGauss_32f_D2(obs,step,mean+k*step,var+k*step,width,
                        gamma+k*step1,num,weight[k]);
    ippsLogAdd_32f(gamma+k*step1,probs,num,ippAlgHintNone);
}
ippsSum_32f(probs,num,&result,ippAlgHintNone);
/* gamma matrix and sum calculation */
ippsExp_32f_I(gamma,height*step1);
ippsSumRow_32f_D2(gamma,height,step1,gammaT,num);
ippsSum_32f(gammaT,height,&sumGamma,ippAlgHintNone);
/* weights update */
ippsDivC_32f(gammaT,sumGamma,weight,height);
/* means and variances update */
for (k=0; k<height; k++) {
    ippsDivC_32f_I(gammaT[k],gamma+k*step1,num);
    ippsWeightedMeanVarColumn_32f_D2(obs,step,gamma+k*step1,num,
                                     mean+k*step,var+k*step,width);
}
}

```

## NormalizeColumn

*Normalizes the matrix columns given the column means and variances.*

---

### Syntax

```

IppStatus ippsNormalizeColumn_16s_D2Sfs(Ipp16s* pSrcDst, int step, int height,
const Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);

IppStatus ippsNormalizeColumn_16s_D2LSfs(Ipp16s** mSrcDst, int height, const
Ipp16s* pMean, const Ipp16s* pVar, int width, int scaleFactor);

IppStatus ippsNormalizeColumn_32f_D2(Ipp32f* pSrcDst, int step, int height,
const Ipp32f* pMean, const Ipp32f* pVar, int width);

IppStatus ippsNormalizeColumn_32f_D2L(Ipp32f** mSrcDst, int height, const
Ipp32f* pMean, const Ipp32f* pVar, int width);

```

### Parameters

<i>pSrcDst</i>	Pointer to the input and output vector [ <i>height</i> * <i>step</i> ].
<i>mSrcDst</i>	Pointer to the input and output matrix [ <i>height</i> ][ <i>width</i> ].
<i>pMean</i>	Pointer to the column mean vector [ <i>width</i> ].
<i>pVar</i>	Pointer to the column variance vector [ <i>width</i> ].
<i>width</i>	Length of the mean and variance vectors.
<i>step</i>	Row step in the input and output vector <i>pSrcDst</i> .
<i>height</i>	Number of rows in the input and output matrix <i>mSrcDst</i> .

### Description

The function `ippsNormalizeColumn` is declared in the `ippsr.h` file. This function normalizes the columns of the matrix *mSrcDst*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> or <i>height</i> is less than or equal to 0.
<code>ippStsStrideErr</code>	Indicates an error when <i>step</i> is less than <i>width</i> .

## NormalizeInRange

Normalizes and scales input vector elements.

### Syntax

```
IppStatus ippsNormalizeInRange_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int
len, Ipp32f lowCut, Ipp32f highCut, Ipp8u range);
```

```
IppStatus ippsNormalizeInRange_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int
len, Ipp32f lowCut, Ipp32f highCut, Ipp8u range);
```

```
IppStatus ippsNormalizeInRange_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int
len, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);
```

```
IppStatus ippsNormalizeInRange_16s(const Ipp16s* pSrc, Ipp16s* pDst, int
len, Ipp32f lowCut, Ipp32f highCut, Ipp16su range);
```

```
IppStatus ippsNormalizeInRange_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
len, Ipp32f lowCut, Ipp32f highCut, Ipp32f range);
```

```
IppStatus ippsNormalizeInRange_16s_I(const Ipp16s* pSrcDst, int len, Ipp32f
lowCut, Ipp32f highCut, Ipp16su range);
```

```
IppStatus ippsNormalizeInRange_32f_I(const Ipp32f* pSrcDst, int len, Ipp32f
lowCut, Ipp32f highCut, Ipp32f range);
```

```
IppStatus ippsNormalizeInRangeMinMax_16s8u(const Ipp16s* pSrc, Ipp8u* pDst,
int len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut , Ipp8u
range);
```

```
IppStatus ippsNormalizeInRangeMinMax_32f8u(const Ipp32f* pSrc, Ipp8u* pDst,
int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp8u
range);
```

```
IppStatus ippsNormalizeInRangeMinMax_32f16s(const Ipp32f* pSrc, Ipp16s* pDst,
int len, Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s
range);
```

```

IppStatus ippsNormalizeInRangeMinMax_16s(const Ipp16s* pSrc, Ipp16s* pDst,
int len, Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);

IppStatus ippsNormalizeInRangeMinMax_32f(const Ipp32f* pSrc, Ipp32f* pDst,
int , Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f range);

IppStatus ippsNormalizeInRangeMinMax_16s_I(const Ipp16s* pSrcDst, int len,
Ipp16s valMin, Ipp16s valMax, Ipp32f lowCut, Ipp32f highCut, Ipp16s range);

IppStatus ippsNormalizeInRangeMinMax_32f_I(const Ipp32f* pSrcDst, int len,
Ipp32f valMin, Ipp32f valMax, Ipp32f lowCut, Ipp32f highCut, Ipp32f range);

```

## Parameters

<i>pSrc</i>	Pointer to the input array.
<i>pSrcDst</i>	inter to the input and output array (for the in-place operation).
<i>pDst</i>	Pointer to the output array.
<i>len</i>	Number of elements in the input and output array.
<i>lowCut</i>	Lower cutoff value.
<i>highCut</i>	Higher cutoff value.
<i>range</i>	Upper bound of output data values (lower bound is 0).
<i>valMin</i>	Minimum value of input data.
<i>valMax</i>	Maximum value of input data.

## Description

The function `ippsNormalizeInRange` is declared in the `ippsr.h` file. This function first normalizes the input vector elements to the range from 0 to 1 imposing lower and higher cutoffs, and then scales them by the *range* value. Cutoff values must be in range [0, 1], and *lowCut* must be less than *highCut*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

<code>ippStsBadArgErr</code>	Indicates an error when cutoff values are not valid, or when <i>range</i> is less than 0, or when <i>valMin</i> is greater than <i>valMax</i> .
<code>ippStsInvZero</code>	Indicates a warning if maximum and minimum values are equal. All elements of the output vector are set to 0.

## MeanVarAcc

*Accumulates the estimates for the mean and variance re-estimation.*

---

### Syntax

```
IppStatus ippsMeanVarAcc_32f(Ipp32f const* pSrc, Ipp32f const* pSrcMean,
Ipp32f* pDstMeanAcc, Ipp32f* pDstVarAcc, int len, Ipp32f val);
```

```
IppStatus ippsMeanVarAcc_64f(Ipp64f const* pSrc, Ipp64f const* pSrcMean,
Ipp64f* pDstMeanAcc, Ipp64f* pDstVarAcc, int len, Ipp64f val);
```

### Parameters

<i>pSrc</i>	Pointer to the observation vector .
<i>pSrcMean</i>	Pointer to the old mean vector .
<i>pDstMeanAcc</i>	Pointer to the mean accumulator.
<i>pDstVarAcc</i>	Pointer to the variance accumulator.
<i>len</i>	Length of the vectors.
<i>val</i>	Constant value in the re-estimation.

### Description

The function `ippsMeanVarAcc` is declared in the `ippsr.h` file. This function accumulates the estimates for the mean and variance re-estimation in the forward-backward algorithm.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.



## GaussianDist

Calculates the distance between two Gaussians.

### Syntax

```
IppStatus ippsGaussianDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f
wgt1, Ipp32f det1, Ipp32f wgt2, Ipp32f det2);
```

```
IppStatus ippsGaussianDist_64f(const Ipp64f* pMean1, const Ipp64f* pVar1,
const Ipp64f* pMean2, const Ipp64f* pVar2, int len, Ipp64f* pResult, Ipp64f
wgt1, Ipp64f det1, Ipp64f wgt2, Ipp64f det2);
```

### Parameters

<i>pMean1</i>	Pointer to the input Gaussian mean vector, also the mean vector of the first Gaussian after splitting.
<i>pVar1</i>	Pointer to the input Gaussian variance vector, also the variance vector of the first Gaussian after splitting.
<i>pMean2</i>	Pointer to the mean vector of the second Gaussian after splitting.
<i>pVar2</i>	Pointer to the variance vector of the second Gaussian after splitting.
<i>len</i>	Length of the mean and variance vectors.
<i>pResult</i>	Pointer to the distance value.
<i>wgt1</i>	Weight of the first Gaussian.
<i>det1</i>	Determinant of the first Gaussian (in the logarithmic representation).
<i>wgt2</i>	Weight of the second Gaussian.
<i>det2</i>	Determinant of the second Gaussian (in the logarithmic representation).

### Description

The function `ippsGaussianDist` is declared in the `ippsr.h` file. This function calculates the distance between two Gaussians.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>-Inf</code> .
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>NaN</code> .

## GaussianSplit

*Splits a single Gaussian component into two with the same variance.*

---

### Syntax

```
IppStatus ippsGaussianSplit_32f(Ipp32f* pMean1, Ipp32f* pVar1, Ipp32f* pMean2,
Ipp32f* pVar2, int len, Ipp32f val);
```

```
IppStatus ippsGaussianSplit_64f(Ipp64f* pMean1, Ipp64f* pVar1, Ipp64f* pMean2,
Ipp64f* pVar2, int len, Ipp64f val);
```

### Parameters

<code>pMean1</code>	Pointer to the input Gaussian mean vector, also the mean vector of the first Gaussian after splitting.
<code>pVar1</code>	Pointer to the input Gaussian variance vector, also the variance vector of the first Gaussian after splitting.
<code>pMean2</code>	Pointer to the mean vector of the second Gaussian after splitting.
<code>pVar2</code>	Pointer to the variance vector of the second Gaussian after splitting.
<code>len</code>	Length of the mean and variance vectors.
<code>val</code>	Variance perturbation value.

## Description

The function `ippsGaussianSplit` is declared in the `ippsr.h` file. This function splits the Gaussian component into two Gaussian mixture components as follows:

$$pMean1[i] = pMean1[i] + val * (pVar1[i])^{1/2},$$

$$pMean2[i] = pMean1[i] - val * (pVar1[i])^{1/2},$$

$$pVar2[i] = pVar1[i]$$

$$0 \leq i < len.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## GaussianMerge

*Merges two Gaussian probability distribution functions.*

---

### Syntax

```
IppStatus ippsGaussianMerge_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
const Ipp32f* pMean2, const Ipp32f* pVar2, Ipp32f* pDstMean, Ipp32f* pDstVar,
int len, Ipp32f* pDstDet, Ipp32f wgt1, Ipp32f wgt2);

IppStatus ippsGaussianMerge_64f(const Ipp64f* pMean1, const Ipp64f* pVar1,
const Ipp64f* pMean2, const Ipp64f* pVar2, Ipp64f* pDstMean, Ipp64f* pDstVar,
int len, Ipp64f* pDstDet, Ipp64f wgt1, Ipp64f wgt2);
```

### Parameters

<code>pMean1</code>	Pointer to the input Gaussian mean vector, also the mean vector of the first Gaussian after splitting.
<code>pVar1</code>	Pointer to the input Gaussian variance vector, also the variance vector of the first Gaussian after splitting.
<code>pMean2</code>	Pointer to the mean vector of the second Gaussian after splitting.

<i>pVar2</i>	Pointer to the variance vector of the second Gaussian after splitting.
<i>len</i>	Length of the mean and variance vectors.
<i>pDstMean</i>	Pointer to the mean vector of the merged Gaussian.
<i>pDstVar</i>	Pointer to the variance vector of the merged Gaussian.
<i>pDstDet</i>	Pointer to the determinant of the merged Gaussian.
<i>wgt1</i>	Weight of the first Gaussian.
<i>wgt2</i>	Weight of the second Gaussian.

## Description

The function `ippsGaussianMerge` is declared in the `ippsr.h` file. This function merges two Gaussian probability distribution functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>-Inf</code> .
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The value of the destination vector element for floating-point operations is set to <code>NaN</code> .

## Entropy

*Calculates entropy of the input vector.*

---

### Syntax

```

IppStatus ippsEntropy_32f(const Ipp32f* pSrc, int len, Ipp32f* pResult);
IppStatus ippsEntropy_16s32s_Sfs(const Ipp16s* pSrc, int srcShiftVal, int
len, Ipp32s* pResult, int scaleFactor);

```

## Parameters

<code>pSrc</code>	Pointer to the input vector.
<code>pResult</code>	Pointer to the destination entropy value.
<code>len</code>	Length of the input vector.
<code>srcShiftVal</code>	Scale factor, refer to <a href="#">Integer Scaling</a> .
<code>scaleFactor</code>	Scale factor, refer to <a href="#">Integer Scaling</a> .

## Description

The function `ippsEntropy` is declared in the `ippsr.h` file. This function calculates entropy of the input vector as given by:

$$pResult[0] = \sum_{i=0}^{len-1} pSrc[i] \cdot \log_2 pSrc[i]$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsLnNegArgI</code>	Indicates a warning that some input vector elements are less than 0. Operation execution is not aborted. For floating-point operations the destination value is set to <code>NaN</code> , and for integer operations it is set to 0.

## SinC

*Calculates sine divided by its argument.*

---

### Syntax

```
IppStatus ippsSinC_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsSinC_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsSinC_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
```

```
IppStatus ippsSinC_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsSinC_64f_I(Ipp64f* pSrcDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector .
<i>pSrcDst</i>	Pointer to the input and destination vector.
<i>pDst</i>	Pointer to the destination vector .
<i>len</i>	Length of the input and output vectors.

### Description

The function `ippsSinC` is declared in the `ippsr.h` file. This function calculates destination vector elements according to the formula:

$$pDst[k] = \sin(pSrc[k]) / pSrc[k], k = 0, \dots, len - 1.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## ExpNegSqr

*Calculates exponential of the squared argument taken with the inverted sign.*

---

### Syntax

```
IppStatus ippsExpNegSqr_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);
IppStatus ippsExpNegSqr_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNegSqr_32f64f(const Ipp32f* pSrc, Ipp64f* pDst, int len);
IppStatus ippsExpNegSqr_32f_I(Ipp32f* pSrcDst, int len);
IppStatus ippsExpNegSqr_64f_I(Ipp64f* pSrcDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector .
-------------	-------------------------------

<i>pSrcDst</i>	Pointer to the input and destination vector.
<i>pDst</i>	Pointer to the destination vector .
<i>len</i>	Length of the input and output vectors.

### Description

The function `ippsExpNegSqr` is declared in the `ippsr.h` file. This function calculates destination vector elements according to the following formula:

$$pDst[k] = \exp(-pSrc[k]^2), k = 0, \dots, len - 1.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## BhatDist

*Calculates the Bhattacharia distance between two Gaussians.*

---

### Syntax

```
IppStatus ippsBhatDist_32f(const Ipp32f* pMean1, const Ipp32f* pVar1, const
Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult);
```

```
IppStatus ippsBhatDist_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1,
const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult);
```

```
IppStatus ippsBhatDistSLog_32f(const Ipp32f* pMean1, const Ipp32f* pVar1,
const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp32f* pResult, Ipp32f
sumLog1, Ipp32f sumLog2);
```

```
IppStatus ippsBhatDistSLog_32f64f(const Ipp32f* pMean1, const Ipp32f* pVar1,
const Ipp32f* pMean2, const Ipp32f* pVar2, int len, Ipp64f* pResult, Ipp32f
sumLog1, Ipp32f sumLog2);
```

### Parameters

<i>pMean1</i>	Pointer to the first mean vector .
<i>pVar1</i>	Pointer to the first variance vector.

<i>pMean2</i>	Pointer to the second mean vector.
<i>pVar2</i>	Pointer to the second variance vector.
<i>pResult</i>	Pointer to the result.
<i>len</i>	Length of the input mean and variance vectors.
<i>sumLog1</i>	Sum of the first Gaussian variance in the logarithmic representation.
<i>sumLog2</i>	Sum of the second Gaussian variance in the logarithmic representation.

## Description

The functions `ippsBhatDist` and `ippsBhatDistSLog` are declared in the `ippsr.h` file.

These functions calculate the Bhattacharia distance between two Gaussians.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning that a zero value was detected in the input vector. The execution is not aborted. The result value is set to <code>-Inf</code> if there is no negative element in the vector.
<code>ippStsLnNegArg</code>	Indicates a warning that negative values were detected in the input vector. The execution is not aborted. The result value is set to <code>NaN</code> .

## UpdateMean

*Updates the mean vector in the EM training algorithm.*

---

### Syntax

```

IppStatus ippsUpdateMean_32f(const Ipp32f* pMeanAcc, Ipp32f* pMean, int len,
Ipp32f meanOcc);

IppStatus ippsUpdateMean_64f(const Ipp64f* pMeanAcc, Ipp64f* pMean, int len,
Ipp64f meanOcc);

```



## Parameters

<i>pMeanAcc</i>	Pointer to the mean accumulator [ <i>len</i> ] .
<i>pMean</i>	Pointer to the mean vector [ <i>len</i> ] .
<i>len</i>	Length of the mean vector.
<i>meanOcc</i>	Occupation sum of the Gaussian mixture.

## Description

The function `ippsUpdateMean` is declared in the `ippsr.h` file. This function calculates the updated mean vector in the EM (Expectation-Maximization) training algorithm.

Note that if  $meanOcc \leq 0$ , the mean vector *pMean* is not updated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsNegOccErr</code>	Indicates an error when <i>meanOcc</i> is less than 0.
<code>ippStsZeroOcc</code>	Indicates a warning that <i>meanOcc</i> is equal to 0.

## UpdateVar

*Updates the variance vector in the EM training algorithm.*

---

### Syntax

```
IppStatus ippsUpdateVar_32f(const Ipp32f* pMeanAcc, const Ipp32f* pVarAcc,
const Ipp32f* pVarFloor, Ipp32f* pVar, int len, Ipp32f meanOcc, Ipp32f
varOcc);
```

```
IppStatus ippsUpdateVar_64f(const Ipp64f* pMeanAcc, const Ipp64f* pVarAcc,
const Ipp64f* pVarFloor, Ipp64f* pVar, int len, Ipp64f meanOcc, Ipp64f
varOcc);
```

## Parameters

<i>pMeanAcc</i>	Pointer to the mean accumulator [ <i>len</i> ].
-----------------	---

<i>pVarAcc</i>	Pointer to the variance accumulator [ <i>len</i> ].
<i>pVarFloor</i>	Pointer to the variance floor vector [ <i>len</i> ].
<i>pVar</i>	Pointer to the variance vector [ <i>len</i> ].
<i>len</i>	Length of the variance.
<i>meanOcc</i>	Occupation sum of the Gaussian mixture.
<i>varOcc</i>	Square occupation sum of the variance mixture.

## Description

The function `ippUpdateVar` is declared in the `ippsr.h` file. This function calculates the updated variance vector in the EM algorithm. The covariance matrix is assumed to be diagonal. The accumulators are calculated from the training data.

Note that if  $meanOcc \leq 0$ , or  $varOcc \leq 0$ , the variance vector *pVar* is not updated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsNegOccErr</code>	Indicates an error when <i>meanOcc</i> and <i>varOcc</i> is less than 0.
<code>ippStsZeroOcc</code>	Indicates a warning that <i>meanOcc</i> or <i>varOcc</i> is equal to 0.
<code>ippStsResFloor</code>	Indicates a warning that all variances are floored.

## UpdateWeight

*Updates the weight values of Gaussian mixtures in the EM training algorithm.*

---

### Syntax

```

IppStatus ippsUpdateWeight_32f(const Ipp32f* pWgtAcc, Ipp32f* pWgt, int len,
Ipp32f* pWgtSum, Ipp32f wgtOcc, Ipp32f wgtThresh);

IppStatus ippsUpdateWeight_64f(const Ipp64f* pWgtAcc, Ipp64f* pWgt, int len,
Ipp64f* pWgtSum, Ipp64f wgtOcc, Ipp64f wgtThresh);

```

## Parameters

<code>pWgtAcc</code>	Pointer to the weight accumulator [ <code>len</code> ].
<code>pWgt</code>	Pointer to the weight vector [ <code>len</code> ].
<code>len</code>	Number of Gaussian mixture components.
<code>pWgtSum</code>	Pointer to the output sum of weight values.
<code>wgtOcc</code>	Nominator of the weight update equation.
<code>wgtThresh</code>	Threshold for the weight values.

## Description

The function `ippsUpdateWeight` is declared in the `ippsr.h` file. This function calculates the updated weight values for a Gaussian mixture. The accumulators are calculated from the training data.

Note that if  $wgtOcc \leq 0$ , the weight vector `pWgt` is not updated.

This function can also be used to update the HMM transition matrix.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsNegOccErr</code>	Indicates an error when <code>wgtOcc</code> is less than 0.
<code>ippStsZeroOcc</code>	Indicates a warning that <code>wgtOcc</code> is equal to 0.
<code>ippStsResFloor</code>	Indicates a warning that all weights are floored.

## UpdateGConst

*Updates the fixed constant in the Gaussian output probability density function.*

---

### Syntax

```

IppStatus ippsUpdateGConst_32f(const Ipp32f* pVar, int len, Ipp32f* pDet);
IppStatus ippsUpdateGConst_64f(const Ipp64f* pVar, int len, Ipp64f* pDet);
IppStatus ippsUpdateGConst_DirectVar_32f(const Ipp32f* pVar, int len, Ipp32f*
pDet);

```

```
IppStatus ippsUpdateGConst_DirectVar_64f(const Ipp64f* pVar, int len, Ipp64f* pDet);
```

## Parameters

<i>pVar</i>	Pointer to the variance vector.
<i>len</i>	Dimension of the variance vector.
<i>pDet</i>	Pointer to the result value.

## Description

The function `ippsUpdateGConst` is declared in the `ippsr.h` file. This function calculates the fixed variance constant.

For functions without the `DirectVar` suffix, the Gaussian covariance matrix is inverse diagonal, for functions with the `DirectVar` suffix, the Gaussian covariance matrix is diagonal.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsLnZeroArg</code>	Indicates a warning that a zero value was detected in the input vector. The execution is not aborted. The result value is set to <code>-Inf</code> if there is no negative element in the vector.
<code>ippStsLnNegArg</code>	Indicates a warning that negative values were detected in the input vector. The execution is not aborted. The result value is set to <code>NaN</code> .

## OutProbPreCalc

*Pre-calculates the part of Gaussian mixture output probability that is irrelevant to observation vectors.*

### Syntax

```
IppStatus ippsOutProbPreCalc_32s(const Ipp32s* pWeight, const Ipp32s* pSrc, Ipp32s* pDst, int len);
```

```
IppStatus ippsOutProbPreCalc_32s_I(const Ipp32s* pWeight, Ipp32s* pSrcDst, int len);
```

```

IppStatus ippsOutProbPreCalc_32f(const Ipp32f* pWeight, const Ipp32f* pSrc,
Ipp32f* pDst, int len);

IppStatus ippsOutProbPreCalc_64f(const Ipp64f* pWeight, const Ipp64f* pSrc,
Ipp64f* pDst, int len);

IppStatus ippsOutProbPreCalc_32f_I(const Ipp32f* pWeight, Ipp32f* pSrcDst,
int len);

IppStatus ippsOutProbPreCalc_64f_I(const Ipp64f* pWeight, Ipp64f* pSrcDst,
int len);

```

## Parameters

<i>pWeight</i>	Pointer to the Gaussian mixture weight vector [ <i>len</i> ].
<i>pSrc</i>	Pointer to the input vector calculated by the function <code>ippsUpdateGConst</code> .
<i>pSrcDst</i>	Pointer to the input and output vector calculated by the function <code>ippsUpdateGConst</code> .
<i>pDst</i>	Pointer to the result vector [ <i>len</i> ] .
<i>len</i>	Number of mixtures in the HMM state.

## Description

The function `ippsOutProbPreCalc` is declared in the `ippsr.h` file. This function pre-calculates the part of the Gaussian mixture output probability that is irrelevant to the observation vectors.

For the function `ippsOutProbPreCalc`,

$$pDst[i] = pWeight[i] - 0.5 * pSrc[i], 0 \leq i < len.$$

For the function `ippsOutProbPreCalc_I`,

$$pSrcDst[i] = pWeight[i] - 0.5 * pSrcDst[i], 0 \leq i < len.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## DcsClustLAccumulate

*Updates the accumulators for calculating the state-cluster likelihood in the decision-tree clustering algorithm.*

---

### Syntax

```

IppStatus ippsDcsClustLAccumulate_32f( const Ipp32f* pMean, const Ipp32f*
pVar, Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);

IppStatus ippsDcsClustLAccumulate_64f(const Ipp64f* pMean, const Ipp64f*
pVar, Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);

IppStatus ippsDcsClustLAccumulate_DirectVar_32f(const Ipp32f* pMean, const
Ipp32f* pVar, Ipp32f* pDstSum, Ipp32f* pDstSqr, int len, Ipp32f occ);

IppStatus ippsDcsClustLAccumulate_DirectVar_64f(const Ipp64f* pMean, const
Ipp64f* pVar, Ipp64f* pDstSum, Ipp64f* pDstSqr, int len, Ipp64f occ);

```

### Parameters

<i>pMean</i>	Pointer to the mean vector of an HMM state in the cluster [ <i>len</i> ].
<i>pVar</i>	Pointer to the variance vector of an HMM state in the cluster [ <i>len</i> ].
<i>pDstSum</i>	Pointer to the summation part of the accumulator [ <i>len</i> ].
<i>pDstSqr</i>	Pointer to the square sum part of the accumulator [ <i>len</i> ].
<i>len</i>	Length of the mean and variance vectors.
<i>occ</i>	Occupation counts of the HMM state

### Description

The function `ippsDcsClustLAccumulate` is declared in the `ippsr.h` file. This function updates the accumulators in the decision-tree clustering algorithm. The accumulators are used to calculate the likelihood of an HMM state cluster.

For functions without the `DirectVar` modifier, the Gaussian covariance matrix must be inverse diagonal.

For functions with the `DirectVar` modifier, the Gaussian covariance matrix must be diagonal.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## DcsClustLCompute

*Calculates the likelihood of an HMM state cluster in the decision-tree state-clustering algorithm.*

### Syntax

```
ippStatus ippsDcsClustLCompute_64f(const Ipp64f* pSrcSum, const Ipp64f*
pSrcSqr, int len, Ipp64f* pDst, Ipp64f occ);

ippStatus ippsDcsClustLCompute_32f64f(const Ipp32f* pSrcSum, const Ipp32f*
pSrcSqr, int len, Ipp64f* pDst, Ipp32f occ);
```

### Parameters

<code>pSrcSum</code>	Pointer to the summation part of the accumulator [ <code>len</code> ].
<code>pSrcSqr</code>	Pointer to the square sum part of the accumulator [ <code>len</code> ].
<code>len</code>	Length of the <code>pSrcSum</code> and <code>pSrcSqr</code> vectors.
<code>pDst</code>	Pointer to the result likelihood value.
<code>occ</code>	Occupation sum of the HMM state cluster.

### Description

The function `ippsDcsClustLCompute` is declared in the `ippsr.h` file. This function calculates the likelihood of an HMM state cluster, according to the accumulators computed by `ippsDcsClustLAccumulate` function. The likelihood value is used to determine the splitting of a decision tree node in the decision-tree state-clustering algorithm.

Note that if `occ = 0`, `pDst[0]` is set to `IPPLOGZERO`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> or <code>occ</code> is less than or equal to 0.
<code>ippStsZeroOcc</code>	Indicates a warning that <code>occ</code> is equal to 0.
<code>ippStsNegOccErr</code>	Indicates an error when <code>occ</code> is less than 0.
<code>ippStsLnZeroArg</code>	Indicates a warning for zero-valued input vector elements. Operation execution is not aborted. The result value is set to <code>-Inf</code> if there are no negative elements in the vector.
<code>ippStsLnNegArg</code>	Indicates a warning for negative input vector elements. Operation execution is not aborted. The result value is set to <code>NaN</code> .

## Model Adaptation

This section describes functions that can be used to adapt the acoustic and language models. The adaptation algorithms adjust the parameters of existing models to match the characteristics set by the users, given a few learning samples.

### AddMulColumn

*Adds a weighted matrix column to the other column.*

---

#### Syntax

```
IppStatus ippsAddMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
int col1, int col2, int row1, const Ipp64f val);
```

#### Parameters

<code>mSrc</code>	Pointer to the source matrix [ <code>height</code> ][ <code>width</code> ].
<code>width</code>	Number of columns in the matrix <code>mSrc</code> .
<code>height</code>	Number of rows in the matrix <code>mSrc</code> .
<code>col1</code>	Column number of the first operand.
<code>col2</code>	Column number of the second operand.
<code>row1</code>	First row number.
<code>val</code>	Weight factor.



## Description

The function `ippsAddMulColumn` is declared in the `ippsr.h` file. This function adds a matrix column weighted by `val` to the other matrix column. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2] + mSrcDst[i][col1] * val, \text{ row1} \leq i < \text{height}.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>width</code> , <code>height</code> , <code>col1</code> , <code>col2</code> , or <code>row1</code> is less than or equal to 0; or <code>row1</code> is greater than or equal to <code>height</code> ; or <code>col1</code> or <code>col2</code> is greater than or equal to <code>width</code> .

## AddMulRow

*Adds a weighted vector to the other vector.*

---

### Syntax

```
ippsAddMulRow_64f(const Ipp64f* pSrc, Ipp64f* pSrcDst, int len, const Ipp64f val);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pSrcDst</code>	Pointer to the source and destination vector.
<code>len</code>	Length of the source and destination vectors.
<code>val</code>	Weight factor.

### Description

The function `ippsAddMulRow` is declared in the `ippsr.h` file. This function adds a vector weighted by `val` to the other vector. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$pSrcDst[i] = pSrcDst[i] + pSrc[i] * val, 0 \leq i < len.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## QRTransColumn

*Performs the QR transformation.*

---

### Syntax

```
ippStatus ippsQRTransColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height,
int col1, int col2, const Ipp64f val1, const Ipp64f val2);
```

### Parameters

<code>mSrc</code>	Pointer to the source matrix [ <code>height</code> ][ <code>width</code> ].
<code>width</code>	Number of columns in the matrix <code>mSrc</code> .
<code>height</code>	Number of rows in the matrix <code>mSrc</code> .
<code>col1</code>	Column number of the first operand.
<code>col2</code>	Column number of the second operand.
<code>val1</code>	First weight factor.
<code>val2</code>	Second weight factor.

### Description

The `ippsQRTransColumn` is declared in the `ippsr.h` file. This function performs the QR transform. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][col2] = mSrcDst[i][col2]*val1 + mSrcDst[i][col1] * val2 ,$$

$$mSrcDst[i][col1] = mSrcDst[i][col1]*val1 + mSrcDst[i][col2] * val2 , 0 \leq i < height.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

`ippStsSizeErr` Indicates an error when *width*, *height*, *col1*, *col2*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *col1* or *col2* is greater than or equal to *width*.

## DotProdColumn

Calculates the dot product of two matrix columns.

### Syntax

```
IppStatus ippsDotProdColumn_64f_D2L(const Ipp64f** mSrc, int width, int
height, Ipp64f* pSum, int col1, int col2, int row1);
```

### Parameters

<i>mSrc</i>	Pointer to the source matrix [ <i>height</i> ][ <i>width</i> ].
<i>width</i>	Number of columns in the matrix <i>mSrc</i> .
<i>height</i>	Number of rows in the matrix <i>mSrc</i> .
<i>pSum</i>	Pointer to the computed sum.
<i>col1</i>	First column number.
<i>col2</i>	Second column number.
<i>row1</i>	First row number.

### Description

The function `ippsDotProdColumn` is declared in the `ippsr.h` file. This function calculates the dot product of two matrix columns. It is used for fast execution of the SVD algorithm.

$$pSum[0] = \sum_{i=row1}^{height-1} mSrc[i][col1] \cdot mSrc[i][col2]$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

`ippStsSizeErr` Indicates an error when *width*, *height*, *coll*, or *row1* is less than or equal to 0; or *row1* is greater than or equal to *height*; or *coll* is greater than or equal to *width*.

## MulColumn

*Multiplies a matrix column by a value.*

---

### Syntax

```
IppStatus ippMulColumn_64f_D2L(Ipp64f** mSrcDst, int width, int height, int
coll, int row1, const Ipp64f val);
```

### Parameters

<i>mSrcDst</i>	Pointer to the source and destination matrix [ <i>height</i> ][ <i>width</i> ].
<i>width</i>	Number of columns in the matrix <i>mSrc</i> .
<i>height</i>	Number of rows in the matrix <i>mSrc</i> .
<i>coll</i>	First column number.
<i>row1</i>	First row number.
<i>val</i>	Weight factor.

### Description

The function `ippMulColumn` is declared in the `ippsr.h` file. This function multiplies a column of the *mSrcDst* matrix by *val*. It is used for fast execution of the SVD algorithm. The calculation is as follows:

$$mSrcDst[i][coll] = mSrcDst[i][coll] * val, row1 \leq i < height.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>height</i> , <i>coll</i> , or <i>row1</i> is less than or equal to 0; or <i>row1</i> is greater than or equal to <i>height</i> ; or <i>coll</i> is greater than or equal to <i>width</i> .

## SumColumnAbs

*Calculates the absolute sum of matrix column elements.*

---

### Syntax

```
IppStatus ippsSumColumnAbs_64f_D2L(const Ipp64f** mSrc, int width, int height,
Ipp64f* pSum, int coll, int row1);
```

### Parameters

<i>mSrc</i>	Pointer to the source matrix [ <i>height</i> ][ <i>width</i> ].
<i>width</i>	Number of columns in the matrix <i>mSrc</i> .
<i>height</i>	Number of rows in the matrix <i>mSrc</i> .
<i>pSum</i>	Pointer to the computed sum.
<i>coll</i>	First column number.
<i>row1</i>	First row number.

### Description

The function `ippsSumColumnAbs` is declared in the `ippsr.h` file. This function calculates the absolute sum of the matrix column elements. It is used for fast execution of the SVD algorithm.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>height</i> , <i>coll</i> , or <i>row1</i> is less than or equal to 0; or <i>row1</i> is greater than or equal to <i>height</i> ; or <i>coll</i> is greater than or equal to <i>width</i> .

## SumColumnSqr

*Calculates the square sums of weighted matrix column elements.*

---

### Syntax

```
IppStatus ippSumColumnSqr_64f_D2L(Ipp64f** mSrcDst, int width, int height,
Ipp64f* pSum, int coll, int row1, const Ipp64f val);
```

### Parameters

<i>mSrcDst</i>	Pointer to the source and destination matrix [ <i>height</i> ][ <i>width</i> ].
<i>width</i>	Number of columns in the matrix <i>mSrc</i> .
<i>height</i>	Number of rows in the matrix <i>mSrc</i> .
<i>pSum</i>	Pointer to the computed sum.
<i>coll</i>	First column number.
<i>row1</i>	First row number.
<i>val</i>	Weight factor.

### Description

The function `ippSumColumnSqr` is declared in the `ippsr.h` file. This function multiplies the columns of the matrix *mSrcDst* by *val* and calculates the square sums of the column elements. It is used for fast execution of the SVD algorithm.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>height</i> , <i>coll</i> , or <i>row1</i> is less than or equal to 0; or <i>row1</i> is greater than or equal to <i>height</i> ; or <i>coll</i> is greater than or equal to <i>width</i> .

## SumRowAbs

*Calculates the absolute sum of the vector elements.*

---

### Syntax

```
IppStatus ippsSumRowAbs_64f(const Ipp64f* pSrc, int len, Ipp64f* pSum);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSum</i>	Pointer to the value of the computed sum.
<i>len</i>	Length of the source vector <i>pSrc</i> .

### Description

The function `ippsSumRowAbs` is declared in the `ippsr.h` file. This function calculates the absolute sum of the vector elements.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## SumRowSqr

*Calculates the square sum of weighted vector elements.*

---

### Syntax

```
IppStatus ippsSumRowSqr_64f(Ipp64f* pSrcDst, int len, Ipp64f* pSum, const Ipp64f val);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>len</i>	Length of the source vector <i>pSrcDst</i> .
<i>pSum</i>	Pointer to the value of the computed sum.
<i>val</i>	Weight factor.

## Description

The function `ippsSumRowSqr` is declared in the `ippsr.h` file. This function multiplies the vector `pSrcDst` by `val` and calculates the square sum of the vector elements.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## SVD, SVDSort

Performs single value decomposition on a matrix.

### Syntax

```

IppStatus ippsSVD_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int height,
Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int nIter);

IppStatus ippsSVD_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int height,
Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);

IppStatus ippsSVD_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW,
Ipp64f* pDstV, int width, int step, int nIter);

IppStatus ippsSVD_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f* pDstW,
Ipp64f** mDstV, int width, int nIter);

IppStatus ippsSVDSort_64f_D2(const Ipp64f* pSrcA, Ipp64f* pDstU, int height,
Ipp64f* pDstW, Ipp64f* pDstV, int width, int step, int nIter);

IppStatus ippsSVDSort_64f_D2L(const Ipp64f** mSrcA, Ipp64f** mDstU, int
height, Ipp64f* pDstW, Ipp64f** mDstV, int width, int nIter);

IppStatus ippsSVDSort_64f_D2_I(Ipp64f* pSrcDstA, int height, Ipp64f* pDstW,
Ipp64f* pDstV, int width, int step, int nIter);

IppStatus ippsSVDSort_64f_D2L_I(Ipp64f** mSrcDstA, int height, Ipp64f* pDstW,
Ipp64f** mDstV, int width, int nIter);

```

### Parameters

`pSrcA`                      Pointer to the input vector `A` [`height*step`].



<i>pDstU</i>	Pointer to the output vector $U$ [ $height*step$ ] .
<i>pSrcDstA</i>	Pointer to the input matrix $A$ and output matrix $U$ [ $height*step$ ] .
<i>pDstV</i>	Pointer to the output vector $V$ [ $width*step$ ] .
<i>mSrcA</i>	Pointer to the input matrix $A$ [ $height$ ][ $width$ ] .
<i>mDstU</i>	Pointer to the output matrix $U$ [ $height$ ][ $width$ ] .
<i>mSrcDstA</i>	Pointer to the input matrix $A$ and output matrix $U$ [ $height$ ][ $width$ ].
<i>pDstW</i>	Pointer to the output vector $W$ [ $width$ ].
<i>mDstV</i>	Pointer to the output matrix $V$ [ $width$ ][ $width$ ].
<i>height</i>	Number of rows in the input matrix.
<i>width</i>	Number of columns in the input matrix.
<i>step</i>	Row step in the vector <i>pSrcA</i> , <i>pSrcDstA</i> , or <i>pDstV</i> .
<i>nIter</i>	Number of iterations for diagonalization.

## Description

The functions `ippsSVD` and `ippsSVDSort` are declared in the `ippsr.h` file. This function performs Single Value Decomposition (SVD) on the input matrix  $A$ . The output matrices  $U$ ,  $W$ , and  $V$  meet the following condition:

$$A = U \square W \square V^T,$$

where the matrix  $U$  is column-orthogonal, the matrix  $W$  is diagonal (stored as a vector), and the matrix  $V$  is orthogonal.  $V^T$  is the transpose of the matrix  $V$ .

The function `ippsSVDSort` transforms output matrices so that elements of the matrix  $W$  are sorted in descending order of their absolute values.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> , <i>width</i> , <i>step</i> , or <i>nIter</i> is less than or equal to 0, or <i>width</i> is greater than <i>step</i> .
<code>ippStsSVDConvErr</code>	Indicates an error when the SVD algorithm has not converged after <i>nIter</i> iterations.

## WeightedSum

*Calculates the weighted sums of two input vector elements.*

---

### Syntax

```

IppStatus ippsWeightedSum_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);

IppStatus ippsWeightedSum_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);

IppStatus ippsWeightedSum_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);

IppStatus ippsWeightedSumHalf_16s(const Ipp16s* pSrc1, const Ipp16s* pSrc2,
Ipp16s* pDst, int len, Ipp32f weight1, Ipp32f weight2);

IppStatus ippsWeightedSumHalf_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2,
Ipp32f* pDst, int len, Ipp32f weight1, Ipp32f weight2);

IppStatus ippsWeightedSumHalf_64f(const Ipp64f* pSrc1, const Ipp64f* pSrc2,
Ipp64f* pDst, int len, Ipp64f weight1, Ipp64f weight2);

```

### Parameters

<i>pSrc1</i>	Pointer to the first input vector.
<i>pSrc2</i>	Pointer to the second input vector.
<i>pDst</i>	Pointer to the output vector.
<i>len</i>	Length of the input and output vectors.
<i>weight1</i>	First weight value.
<i>weight2</i>	Second weight value.

### Description

The functions `ippsWeightedSum` and `ippsWeightedSumHalf` are declared in the `ippsr.h` file. The function `ippsWeightedSum` calculates the weighted sum as follows:

$$pDst[i] = (weight1 * pSrc[i] + weight2 * pSrc[i]) / (weight1 + weight2),$$

and the function `ippsWeightedSumHalf` calculates the weighted sum as follows:

$$pDst[i] = (pSrc[i] + weight2 * pSrc[i]) / (weight1 + weight2).$$

Here  $i = 0, \dots, len-1$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDivByZero</code>	Indicates a warning that a divisor vector element has zero value. The execution is not aborted. The value of the destination vector element for the floating-point operations is set as follows: <code>NaN</code> for zero-valued dividend vector element; <code>+Inf</code> for positive dividend vector element; <code>-Inf</code> for negative dividend vector element.

## Vector Quantization

This section describes some functions for vector quantization and codebook operations. These functions are commonly used in acoustic and language model compressions.

### FormVector

*Constructs an output vector of multiple streams from codebook entries.*

---

#### Syntax

```
IppStatus ippsFormVector_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc, const
Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
Ipp16s* pDst);
```

```
IppStatus ippsFormVector_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
const Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int
nStream, Ipp16s* pDst);
```

```
IppStatus ippsFormVector_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc, const
Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int nStream,
Ipp32f* pDst);
```

```
IppStatus ippsFormVector_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
const Ipp32s* pHeights, const Ipp32s* pWidths, const Ipp32s* pSteps, int
nStream, Ipp32f* pDst);
```

```
IppStatus ippsFormVector_2i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc,
const Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_2i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
const Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_2i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc,
const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

```
IppStatus ippsFormVector_2i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

```
IppStatus ippsFormVector_4i_8u16s(const Ipp8u* pInd, const Ipp16s** mSrc,
const Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_4i_16s16s(const Ipp16s* pInd, const Ipp16s** mSrc,
const Ipp32s* pHeights, Ipp16s* pDst, int len);
```

```
IppStatus ippsFormVector_4i_8u32f(const Ipp8u* pInd, const Ipp32f** mSrc,
const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

```
IppStatus ippsFormVector_4i_16s32f(const Ipp16s* pInd, const Ipp32f** mSrc,
const Ipp32s* pHeights, Ipp32f* pDst, int len);
```

## Parameters

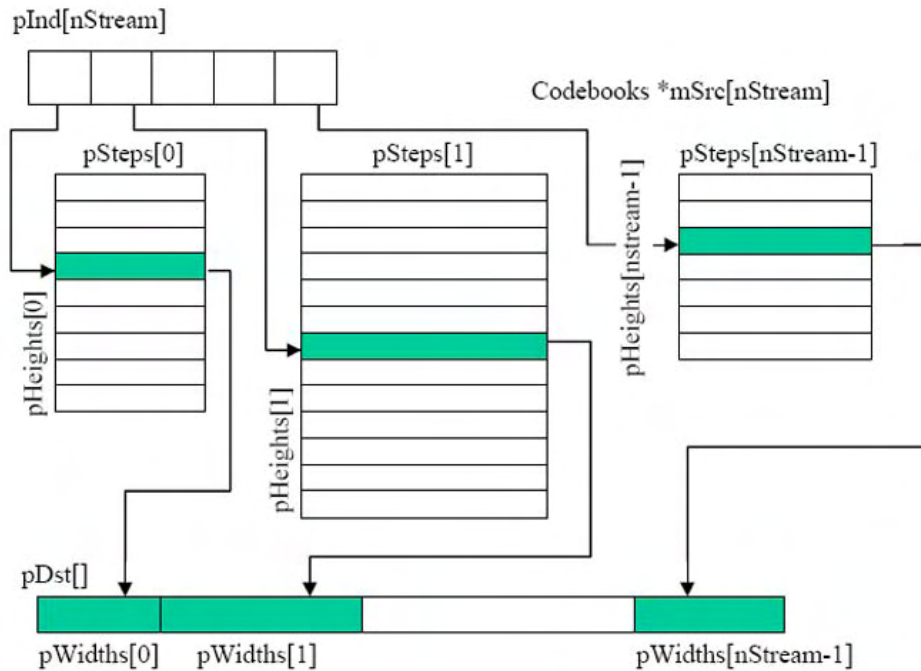
<i>pInd</i>	Pointer to the indexing vector [ <i>nStream</i> ].
<i>mSrc</i>	Pointer to the array of pointers to the codebooks [ <i>nStream</i> ] .
<i>pHeights</i>	Pointer to the codebook lengths [ <i>nStream</i> ] .
<i>pWidths</i>	Pointer to the stream lengths [ <i>nStream</i> ] .
<i>pSteps</i>	Pointer to the codevector lengths [ <i>nStream</i> ] .
<i>nStream</i>	Number of codebooks.
<i>pDst</i>	Pointer to the output vector.
<i>len</i>	Length of the output vector.

## Description

The function `ippsFormVector` is declared in the `ippsr.h` file. This function constructs an output vector of multiple streams. Each stream of size `pWeights[]`, is a codebook entry indexed by `pInd[]`. The codebooks are referenced by `mSrc[]`, and have `pHeights[]` number of codevectors, each of which is of size `pSteps[]`.

Figure 8-5 illustartes the stream layout.

**Figure 8-5 Stream Layout for the function `ippsFormVector`**



The function `ippsFormVector_2i` simplifies the extraction process by posting the constraints of `pWidths[] = pSteps[] = 2` and `nStream = len/2`.

Similarly, the function `ippsFormVector_4i` implies the constraints of `pWidths[ ] = pSteps[ ] = 4` and `nStream = len/4`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>pInd[k]</code> is less than 0, or when <code>len</code> , <code>nStream</code> , <code>pWidths[k]</code> , or <code>pSteps[k]</code> is less than or equal to 0; or when <code>pHeights[k]</code> is less than or equal to <code>pInd[k]</code> .

## CdbkGetSize

*Calculates the size in bytes of the codebook.*

---

### Syntax

```
IppStatus ippsCdbkGetSize_16s(int width, int step, int height, int cdbkSize,
IppCdbk_Hint hint, int* pSize);
```

### Parameters

<code>width</code>	Length of the input vectors (0,.., 512).
<code>step</code>	Row step in the source vector <code>pSrc</code> (0,.., 512).
<code>height</code>	Number of rows in the source vector <code>pSrc</code> , now only <code>height = cdbkSize</code> is supported.
<code>cdbkSize</code>	Size of the codebook (0,.., 8192] .
<code>hint</code>	Flag indicating format of codebook. See <code>ippsCdbkInit</code> for a complete description.
<code>pSize</code>	Pointer to the size in bytes of the codebook structure and associated storage.

### Description

The function `ippsCdbkGetSize` is declared in the `ippsr.h` file. This function calculates the size in bytes of the codebook and additional information to be used for fast search.




---

**CAUTION.** Only `hint = IPP_CDBK_FULL` is currently supported for this function. In this case the parameter `height` is not needed, and its value is ignored.

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCdbk</i> or <i>pSrc</i> pointers are null.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>step</i> , <i>height</i> , or <i>cdbkSize</i> is less than or equal to 0; or <i>width</i> is greater than <i>step</i> ; or <i>cdbkSize</i> is greater than 8192.
<code>ippStsCdbkFlagErr</code>	Indicates an error when the <i>hint</i> value is incorrect or not supported.

## CdbkInit

Initializes the structure that contains the codebook.

### Syntax

```
IppStatus ippCdbkInit_L2_16s(IppsCdbkState_16s* pCdbk, const Ipp16s* pSrc,
int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);
```

### Parameters

<i>pCdbk</i>	Pointer to the codebook structure to be created.
<i>pSrc</i>	Pointer to the source vector [ <i>height</i> * <i>step</i> ].
<i>width</i>	Length of the input vectors (0,.., 512).
<i>step</i>	Row step in the source vector <i>pSrc</i> (0,.., 512).
<i>height</i>	Number of rows in the source vector <i>pSrc</i> (0,.., <i>cdbkSize</i> ) .
<i>cdbkSize</i>	Size of the codebook (0,.., 8192] .
<i>hint</i>	One of the following values: IPP_CDBK_FULLL - the source data are entries of a codebook, <i>height</i> must be greater or equal to <i>cdbkSize</i> . The nearest codebook entry is located through a full search. IPP_CDBK_KMEANS_LONG - LBG algorithm with splitting of the most extensional cluster is used for the codebook building. The nearest codebook entry is located through a logarithmical search.

IPP\_CDBK\_KMEANS\_NUM - LBG algorithm with splitting of the most numerous clusters is used for the codebook building. The nearest codebook entry is located through a logarithmical search.

## Description

The function `ippsCdbkInit` is declared in the `ippsr.h` file. This function initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by the `ippsSplitVQ` function. The Euclidean distance is used to measure the similarity between two vectors.

The only *hint* value that is currently supported for this function is `IPP_CDBK_FULL`. State memory address stored in *pCdbk* must be aligned to 32-bit word boundary.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCdbk</i> or <i>pSrc</i> pointers are null.
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>step</i> , <i>height</i> , or <i>cdbkSize</i> is less than or equal to 0; or width is greater than <i>step</i> ; or <i>cdbkSize</i> is greater than 8192.
<code>ippStsCdbkFlagErr</code>	Indicates an error when the <i>hint</i> value is incorrect or not supported.

## CdbkInitAlloc

*Initializes the codebook structure.*

---

### Syntax

```

IppStatus ippsCdbkInitAlloc_L2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);

IppStatus ippsCdbkInitAlloc_L2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
pSrc, int width, int step, int height, int cdbkSize, Ipp_Cdbk_Hint hint);

IppStatus ippsCdbkInitAlloc_WgtL2_16s(IppsCdbkState_16s** pCdbk, const Ipp16s*
pSrc, const Ipp16s* pWgt, int width, int step, int height, int cdbkSize,
Ipp_Cdbk_Hint hint);

```



```
IppStatus ippsCdbkInitAlloc_WgtL2_32f(IppsCdbkState_32f** pCdbk, const Ipp32f*
pSrc, const Ipp32f* pWgt, int width, int step, int height, int cdbkSize,
Ipp_Cdbk_Hint hint);
```

## Parameters

<i>pCdbk</i>	Pointer to the codebook structure to be created.
<i>pSrc</i>	Pointer to the source vector [ <i>height</i> * <i>step</i> ].
<i>pWgt</i>	Pointer to the weight vector.
<i>width</i>	Length of the input vectors.
<i>step</i>	Row step in the source vector <i>pSrc</i> .
<i>height</i>	Number of rows in the source vector <i>pSrc</i> .
<i>cdbkSize</i>	Size of the codebook.
<i>hint</i>	One of the following values: IPP_CDBK_FULL - the source data are entries of a codebook, <i>height</i> must be greater or equal to <i>cdbkSize</i> . The nearest codebook entry is located through a full search. IPP_CDBK_KMEANS_LONG - LBG algorithm with splitting of the most extensional cluster is used for the codebook building. The nearest codebook entry is located through a logarithmical search. IPP_CDBK_KMEANS_NUM - LBG algorithm with splitting of the most numerous clusters is used for the codebook building. The nearest codebook entry is located through a logarithmical search.

## Description

The function `ippsCdbkInitAlloc` is declared in the `ippsr.h` file. This function initializes the structure that contains the codebook and additional information to be used for fast search. The structure is used during vector quantization by `ippsVQ` or `ippsSplitVQ` functions. The `ippsCdbkInitAlloc_L2` function uses the Euclidean distance to measure the similarity between two vectors, while the `ippsCdbkInitAlloc_WgtL2` function uses the weighted Euclidean distance for these purposes.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>width</i> , <i>step</i> , or <i>cdbkSize</i> is less than or equal to 0; or <i>cdbkSize</i> is greater than <i>height</i> ; or <i>width</i> is greater than <i>step</i> ; or <i>cdbkSize</i> is greater than 16383.
<code>ippStsCdbkFlagErr</code>	Indicates an error when the <i>hint</i> value is incorrect.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory was allocated.
<code>ippStsBadArgErr</code>	Indicates an error when one of <i>pWgt</i> [ <i>i</i> ] is less than 0.
<code>ippStsSmallerCodebook</code>	Indicates a warning when size of the created codebook is less than the <i>cdbkSize</i> argument value.

## CdbkFree

*Destroys the codebook structure.*

---

### Syntax

```
IppStatus ippCdbkFree_16s(IppsCdbkState_16s* pCdbk);
IppStatus ippCdbkFree_32f(IppsCdbkState_32f* pCdbk);
```

### Parameters

*pCdbk*                                      Pointer to the codebook structure.

### Description

The function `ippCdbkFree` is declared in the `ippsr.h` file. This function destroys the codebook structure and frees all memory associated with it.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pCdbk</i> pointer is <code>NULL</code> .

## GetCdbkSize

*Retrieves the number of codevectors in the codebook.*

---

### Syntax

```
IppStatus ippsGetCdbkSize_16s(const IppsCdbkState_16s* pCdbk, int* pNum);  
IppStatus ippsGetCdbkSize_32f(const IppsCdbkState_32f* pCdbk, int* pNum);
```

### Parameters

<i>pCdbk</i>	Pointer to the codebook structure.
<i>pNum</i>	Pointer to the result number of codevectors.

### Description

The function `ippsGetCdbkSize` is declared in the `ippsr.h` file. This function retrieves the number of codevectors in the codebook *pCdbk*. This number can be less than *cdbkSize* if the number of different vectors in *pSrc* is less than *cdbkSize*. The codebook structure *pCdbk* is initialized by the `ippsCdbkInitAlloc` function.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## GetCodebook

*Retrieves the codevectors from the codebook.*

---

### Syntax

```
IppStatus ippsGetCodebook_16s(const IppsCdbkState_16s* pCdbk, Ipp16s* pDst,  
int step);  
IppStatus ippsGetCodebook_32f(const IppsCdbkState_32f* pCdbk, Ipp32f* pDst,  
int step);
```

### Parameters

<i>pCdbk</i>	Pointer to the codebook structure.
--------------	------------------------------------

<i>pDst</i>	Pointer to the destination vector for codevectors [ <i>pNum</i> [0]* <i>step</i> ].
<i>step</i>	Row step in the destination vector <i>pDst</i> .

## Description

The function `ippsGetCodebook` is declared in the `ippsr.h` file. This function retrieves the codevectors from the codebook structure *pCdbk* and stores them in the *pDst* vector with row step *step*.

The codebook structure *pCdbk* is initialized by the function `ippsCdbkAlloc`. The number of clusters can be obtained by the function `ippsGetCdbkSize`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or equal to 0 or <i>step</i> is less than <i>width</i> .

## VQ

Quantizes the input vectors given a codebook.

### Syntax

```
ippStatus ippsVQ_16s(const Ipp16s* pSrc, int step, Ipp32s* pIndx, int height,
const IppsCdbkState_16s* pCdbk);
```

```
ippStatus ippsVQ_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx, int height,
const IppsCdbkState_32f* pCdbk);
```

```
ippStatus ippsVQDist_16s32s_Sfs(const Ipp16s* pSrc, int step, Ipp32s* pIndx,
Ipp32s* pDist, int height, const IppsCdbkState_16s* pCdbk, int scaleFactor);
```

```
ippStatus ippsVQDist_32f(const Ipp32f* pSrc, int step, Ipp32s* pIndx, Ipp32f*
pDist, int height, const IppsCdbkState_32f* pCdbk);
```

### Parameters

<i>pCdbk</i>	Pointer to the codebook structure.
<i>pSrc</i>	Pointer to the source vector [ <i>height</i> * <i>step</i> ].

<i>step</i>	Row step in the source vector <i>pSrc</i> .
<i>height</i>	Number of rows in the source vector <i>pSrc</i> .
<i>pIndx</i>	Pointer to the result index vector of the closest codevectors [ <i>height</i> ].
<i>pDist</i>	Pointer to the result quantization distances from the source vector [ <i>height</i> ].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The functions `ippsVQ` and `ippsVQDist` are declared in the `ippsr.h` file. The function `ippsVQ` performs Vector Quantization (VQ) on the input vectors. The resulting indexes of the closest codevectors are stored in the vector *pIndx*. The function `ippsVQDist` also stores the distances (scaled with *scaleFactor* for the integer versions) to the output distance vector *pDist*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.

## VQSingle\_Sort, VQSingle\_Thresh

*Quantizes the input vector given a codebook and gets several closest clusters.*

---

### Syntax

```

IppStatus ippsVQSingle_Sort_32f(const Ipp32f* pSrc, Ipp32s* pIndx, const
IppsCdbkState_32f* pCdbk, int num);

IppStatus ippsVQSingle_Sort_16s(const Ipp16s* pSrc, Ipp32s* pIndx, const
IppsCdbkState_16s* pCdbk, int num);

IppStatus ippsVQDistSingle_Sort_32f(const Ipp32f* pSrc, Ipp32s* pIndx, Ipp32f*
pDist, const IppsCdbkState_32f* pCdbk, int num);

IppStatus ippsVQDistSingle_Sort_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s* pIndx,
Ipp32s* pDist, const IppsCdbkState_16s* pCdbk, int num, int scaleFactor);

```

```

IppStatus ippsVQSingle_Thresh_32f(const Ipp32f* pSrc, Ipp32s* pIndx, const
IppsCdbkState_32f* pCdbk, Ipp32f val, int* pnum);

IppStatus ippsVQSingle_Thresh_16s(const Ipp16s* pSrc, Ipp32s* pIndx, const
IppsCdbkState_16s* pCdbk, Ipp32f val, int* pnum);

IppStatus ippsVQDistSingle_Thresh_32f(const Ipp32f* pSrc, Ipp32s* pIndx,
Ipp32f* pDist, const IppsCdbkState_32f* pCdbk, Ipp32f val, int* pnum);

IppStatus ippsVQDistSingle_Thresh_16s32s_Sfs(const Ipp16s* pSrc, Ipp32s*
pIndx, Ipp32s* pDist, const IppsCdbkState_16s* pCdbk, Ipp32f val, int* pnum,
int scaleFactor);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pIndx</i>	Pointer to the destination indexes vector.
<i>pDist</i>	Pointer to the destination distances vector.
<i>pCdbk</i>	Pointer to the codebook structure.
<i>val</i>	Relative threshold value.
<i>num</i>	Number of closest clusters to search for each input vector.
<i>pNum</i>	Pointer to the number of clusters within threshold [1].
<i>scaleFactor</i>	Scaling factor for intermediate sums, refer to <a href="#">Integer Scaling</a> .

## Description

The functions `ippsVQSingle_Sort` and `ippsVQSingle_Thresh` are declared in the `ippsr.h` file. These functions perform multiple vector quantization (VQ) for the input vector. Functions with `Sort` modifier provide *num* indexes of closest codebook centroids sorted in distance ascending order.

Functions with `Thresh` modifier provide indexes of codebook centroids with distance less than or equal to minimum multiplied by *val* and the number of such clusters.

Functions with `Dist` suffix provide distance values of closest clusters. The length of vectors *pSrc* is equal to codevector length, the length of output vectors *pIndx* and *pDist* is equal to codebook size. Full search is done for all codebook types.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>num</i> is less than or equal to 0, or greater than the codebook size.
<code>ippStsBadArgErr</code>	Indicates an error when <i>val</i> is less than 1.

## SplitVQ

*Quantizes a multiple-stream vector given the codebooks.*

---

### Syntax

```

IppStatus ippsSplitVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s* pDst,
int dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);

IppStatus ippsSplitVQ_16s8u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst,
int dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);

IppStatus ippsSplitVQ_16s1u(const Ipp16s* pSrc, int srcStep, Ipp8u* pDst,
int dstBitStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);

IppStatus ippsSplitVQ_32f16s(const Ipp32f* pSrc, int srcStep, Ipp16s* pDst,
int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);

IppStatus ippsSplitVQ_32f8u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst,
int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);

IppStatus ippsSplitVQ_32f1u(const Ipp32f* pSrc, int srcStep, Ipp8u* pDst,
int dstBitStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);

```

### Parameters

<i>pCdbks</i>	Pointer to the codebook structures [ <i>nStream</i> ].
<i>pSrc</i>	Pointer to the source vector [ <i>height*srcStep</i> ].
<i>srcStep</i>	Row step in the source vector <i>pSrc</i> .
<i>pDst</i>	Pointer to the destination indexing vector [ <i>height*dstStep</i> ].
<i>dstStep</i>	Row step in the destination vector <i>pDst</i> .

<i>height</i>	Number of rows in the source and destination vectors.
<i>dstBitStep</i>	Row step in the destination vector (in bits).
<i>nStream</i>	Number of streams in the source vectors.

## Description

The functions `ippsSplitVQ` is declared in the `ippsr.h` file. This function quantizes the multiple-stream vectors *pSrc* against given codebooks *pDbks*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors. The outputs *pDst* are indexes to the codebook entries.

For functions with the `1u` data, the output indexes are packed in bits. Each stream takes the least number of bits sufficient to represent its codebook indexes.

See also `ippsFormVector` , `ippsFormVectorVQ` functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>srcStep</i> , <i>dstStep</i> , <i>height</i> , or <i>nStream</i> is less than or equal to 0; or the sum of the stream length is greater than <i>srcStep</i> ; or <i>nStream</i> is greater than <i>dstStep</i> for functions for <code>16s</code> or <code>8u</code> data; or the number of bits sufficient to represent the indexes is greater than <i>dstStep</i> for functions for <code>1u</code> data; or the codebook size is greater than 256 for functions for <code>8u</code> data.

## FormVectorVQ

*Constructs multiple-stream vectors from codebooks, given indexes.*

---

### Syntax

```
IppStatus ippsFormVectorVQ_16s16s(const Ipp16s* pSrc, int srcStep, Ipp16s*
pDst, int dstStep, int height, const IppsCdbkState_16s** pDbks, int nStream);

IppStatus ippsFormVectorVQ_8u16s(const Ipp8u* pSrc, int srcStep, Ipp16s*
pDst, int dstStep, int height, const IppsCdbkState_16s** pDbks, int nStream);
```



```

IppStatus ippsFormVectorVQ_1u16s(const Ipp8u* pSrc, int srcBitStep, Ipp16s*
pDst, int dstStep, int height, const IppsCdbkState_16s** pCdbks, int nStream);

IppStatus ippsFormVectorVQ_16s32f(const Ipp16s* pSrc, int srcStep, Ipp32f*
pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);

IppStatus ippsFormVectorVQ_8u32f(const Ipp8u* pSrc, int srcStep, Ipp32f*
pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);

IppStatus ippsFormVectorVQ_1u32f(const Ipp8u* pSrc, int srcBitStep, Ipp32f*
pDst, int dstStep, int height, const IppsCdbkState_32f** pCdbks, int nStream);

```

## Parameters

<i>pCdbks</i>	Pointer to the codebook structures [ <i>nStream</i> ].
<i>pSrc</i>	Pointer to the indexing vectors [ <i>height*srcStep</i> ].
<i>srcStep</i>	Row step in the indexing vectors <i>pSrc</i> .
<i>pDst</i>	Pointer to the constructed vectors [ <i>height*dstStep</i> ].
<i>dstStep</i>	Row step in the constructed vectors <i>pDst</i> .
<i>height</i>	Number of rows in the indexing vectors.
<i>srcBitStep</i>	Row step in the indexing vectors (in bits).
<i>nStream</i>	Number of streams.

## Description

The function `ippsFormVectorVQ` is declared in the `ippsr.h` file. This function constructs multiple-stream vectors *pDst* from the codebooks *pCdbks* given indexes *pSrc*. The length of each stream is assumed to be equal to that of the corresponding codebook vectors. For functions for 1u data, each stream index is assumed to be in a packed format. Each stream takes the number of bits sufficient to represent its codebook indexes. See also `ippsFormVector`, `ippsSplitVQ` functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>srcStep</i> , <i>dstStep</i> , <i>height</i> , or <i>nStream</i> is less than or equal to 0; or the sum of the stream length is greater than <i>srcStep</i> ; or <i>nStream</i> is greater than <i>dstStep</i> for functions for 16s or 8u data; or the number of bits sufficient to

represent the indexes is greater than *dstStep* for functions for *1u* data; or the codebook size is greater than 256 for functions for *8u* data.

## Polyphase Resampling

The Intel IPP functions described in this section build, apply and free Kaiser-windowed polyphase filters for data resampling. Functions with *Fixed* suffix are intended for fixed rational resampling factor and provide faster speed. Functions without this suffix build universal resampling filter with linear interpolation of filter coefficients and allow to use a variable factor. The general description of the polyphase resampling algorithm can be found, for example, in "*Multirate Digital Signal Processing*" by R. Crochiere and L. Rabiner, Prentice Hall, 1983.

Some of these functions are used in the Intel® IPP Speech Recognition Samples. See *Speech Processing* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

### ResamplePolyphaseInit

*Initializes the structure for polyphase resampling without calculating the filter coefficients.*

---

#### Syntax

```

IppStatus ippsResamplePolyphaseFixedInit_16s(
    IppsResamplingPolyphaseFixed_16s* pSpec, int inRate, int outRate, int len,
    IppHintAlgorithm hint);

IppStatus ippsResamplePolyphaseFixedInit_32f(
    IppsResamplingPolyphaseFixed_32f* pSpec, int inRate, int outRate, int len,
    IppHintAlgorithm hint);

```

#### Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
<i>inRate</i>	The input rate for fixed factor resampling.
<i>outRate</i>	The output rate for fixed factor resampling.
<i>len</i>	The filter length for fixed factor resampling.

*hint*                      Suggests using specific code (must be equal to `ippAlgHintFast`). The possible values for the parameter *hint* are listed in [Hint Arguments](#).

## Description

This function is declared in the `ippsr.h` file. The function `ippsResamplePolyphaseFixedInit` initializes structures for data resampling with the factor equal to  $inRate/outRate$  without calculating the filter coefficients. This function is designed for using pre-calculated filter coefficients for polyphase resampling.

First, you need to allocate memory for the structure. Then use the function `ippsResamplePolyphaseSetFixedFilter` to import filter coefficients. To calculate the memory size, filter length and the number of filters, use the `ippsResamplePolyphaseFixedGetSize` function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>inRate</i> , <i>outRate</i> or <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates that <i>hint</i> is not equal to <code>ippAlgHintFast</code> .

## ResamplePolyphaseGetSize

*Gets the size of the polyphase resampling structure.*

---

### Syntax

```
IppStatus ippsResamplePolyphaseFixedGetSize_16s(int inRate, int outRate, int len, int* pSize, int* pLen, int* pHeight, IppHintAlgorithm hint);
IppStatus ippsResamplePolyphaseFixedGetSize_32f(int inRate, int outRate, int len, int* pSize, int* pLen, int* pHeight, IppHintAlgorithm hint);
```

### Parameters

<i>inRate</i>	The input rate for fixed factor resampling.
<i>outRate</i>	The output rate for fixed factor resampling.
<i>len</i>	The filter length for fixed factor resampling.

<i>pSize</i>	The pointer to the variable that contains the size of the polyphase resampling structure.
<i>pLen</i>	The pointer to the variable that contains the real filter length.
<i>pHeight</i>	The pointer to the variable that contains the number of filters.
<i>hint</i>	Suggests using specific code (must be equal to <code>ippAlgHintFast</code> ). The possible values for the parameter <i>hint</i> are given in <a href="#">Hint Arguments</a> .

### Description

The function `ippsResamplePolyphaseFixedGetSize` is declared in the `ippsr.h` file. This function determines the size required for the fixed rate polyphase resampling structure and associated storage, the filter length and the number of filters in the filter bank. The returned length of the filter is equal to  $\min\{l \geq len, l\%4\}$ , the filter length for zero phase is greater by 1. These values can be used for export and import of fixed polyphase resampling filter.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>inRate</i> , <i>outRate</i> or <i>len</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates that <i>hint</i> is not equal to <code>ippAlgHintFast</code> .

## ResamplePolyphaseSetFilter

Sets polyphase resampling filter coefficients.

### Syntax

```

IppStatus ippsResamplePolyphaseSetFixedFilter_16s(
    IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int step, int
    height);

IppStatus ippsResamplePolyphaseSetFixedFilter_32f(
    IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int step, int
    height);

```

## Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
<i>pSrc</i>	The pointer to the input vector of filter coefficients.
<i>step</i>	The row step in <i>pSrc</i> vector.
<i>height</i>	The number of filters (the number of rows in <i>pSrc</i> vector).

## Description

The function `ippsResamplePolyphaseSetFixedFilter` is declared in the `ippsr.h` file. This function imports pre-calculated filter coefficients into the polyphase resampling structure. If the *step* value is less than the filter length, trailing filter coefficients are zeroed.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates that <i>height</i> is greater than the number of filters in <i>pSpec</i> structure.

## ResamplePolyphaseGetFilter

*Gets polyphase resampling filter coefficients.*

### Syntax

```
IppStatus ippsResamplePolyphaseGetFixedFilter_16s(
    IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int step, int
    height);

IppStatus ippsResamplePolyphaseGetFixedFilter_32f(
    IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int step, int
    height);
```

## Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
--------------	--

<i>pSrc</i>	The pointer to the output vector of filter coefficients.
<i>step</i>	The row step in <i>pSrc</i> vector.
<i>height</i>	The number of filters (the number of rows in <i>pSrc</i> vector).

### Description

The function `ippsResamplePolyphaseSetFixedFilter` is declared in the `ippsr.h` file. This function exports filter coefficients from the polyphase resampling structure. If the *step* value is less than the filter length, only first *step* coefficients are exported.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.
<code>ippStsBadArgErr</code>	Indicates that <i>height</i> is greater than the number of filters in <i>pSpec</i> structure.

### Example 8-3. Export and Import of the Polyphase Resampling Filter Bank

```

int inRate=16000; // input frequency
int outRate=8000; // output frequency
int history; // half of filter length
char fname[]="filter.flt\0";
// coefficient file name
{
    int size,len,height;
    FILE *file;
    short *pFilter;
    IppsresamplingPolyphaseFixed_16s *state;
    history=(int)(64.0f*0.5*IPP_MAX(1.0,1.0/(double)outRate/(double)inRate))+1;
    ippsResamplePolyphaseFixedInitAlloc_16s(&state,inRate,outRate,2*(history-1),
    0.95f,9.0f,ippAlgHintFast);
    ippsResamplePolyphaseFixedGetSize_16s(inRate,outRate,2*(history-1),&size,&len,
    &height,ippAlgHintFast);
    pFilter=ippsMalloc_16s(len*height);
    ippsResamplePolyphaseGetFixedFilter_16s(state,pFilter,len,height);
    file=fopen(fname,"wb");
    fwrite(&size,sizeof(int),1,file);
    fwrite(&len,sizeof(int),1,file);
    fwrite(&height,sizeof(int),1,file);
    fwrite(pFilter,sizeof(short),len*height,file);
    fclose(file);
    ippsFree(pFilter);
    ippsResamplePolyphaseFixedFree_16s(state);
}
{

```

```

int size,len,height;
FILE *file;
short *pFilter;
IppsresamplingPolyphaseFixed_16s *state;
history=(int)(64.0f*0.5*IPP_MAX(1.0,1.0/(double)outRate/(double)inRate))+1;
file=fopen(fname,"rb");
fread(&size,sizeof(int),1,file);
fread(&len,sizeof(int),1,file);
fread(&height,sizeof(int),1,file);
pFilter=ippsMalloc_16s(len*height);
fread(pFilter,sizeof(short),len*height,file);
fclose(file);
state=(IppsresamplingPolyphaseFixed_16s*)ippsMalloc_8u(size);
ippsResamplePolyphaseFixedInit_16s(state,inRate,outRate,2*(history-1), ippAlgHintFast);
ippsResamplePolyphaseSetFixedFilter_16s((IppsresamplingPolyphaseFixed_16s*)
state, (const Ipp16s*) pFilter, len, height);
ippsFree(pFilter);
// use of polyphase filter
...

ippsFree(state);
}

```

## ResamplePolyphaseInitAlloc

*Initializes the structure for polyphase data resampling.*

---

### Syntax

```

IppStatus ippsResamplePolyphaseInitAlloc_16s(IppsResamplingPolyphase_16s**
pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm
hint);

```



```

IppStatus ippsResamplePolyphaseInitAlloc_32f(IppsResamplingPolyphase_32f**
pSpec, Ipp32f window, int nStep, Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm
hint);

IppStatus ippsResamplePolyphaseFixedInitAlloc_16s(
IppsResamplingPolyphaseFixed_16s** pSpec, int inRate, int outRate, int len,
Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm hint);

IppStatus ippsResamplePolyphaseFixedInitAlloc_32f(
IppsResamplingPolyphaseFixed_32f** pSpec, int inRate, int outRate, int len,
Ipp32f rollf, Ipp32f alpha, IppHintAlgorithm hint);

```

## Parameters

<i>pSpec</i>	The pointer to the resampling state structure to be created.
<i>window</i>	The size of the ideal lowpass filter window.
<i>nStep</i>	The discretization step for filter coefficients.
<i>rollf</i>	The roll-off frequency of the filter.
<i>alpha</i>	The parameter of the Kaiser window.
<i>inRate</i>	The input rate for resampling with fixed factor.
<i>outRate</i>	The output rate for resampling with fixed factor.
<i>len</i>	The filter length for resampling with fixed factor.
<i>hint</i>	Suggests using specific code. The possible values for the parameter <i>hint</i> are given in <a href="#">Hint Arguments</a> .

## Description

These functions are declared in the `ippsr.h` file. The functions `ippsResamplePolyphaseInitAlloc` create structures for data resampling using the ideal lowpass filter and applies the Kaiser window with *alpha* parameter and width *window* to it.

This means that the values of the ideal lowpass filtering function are calculated (discretized) for all *i* such that  $|i / nStep| \leq window$ .

The structure created can be used to resample input samples by the function `ippsResample` with arbitrary resampling factor. In this case, filter coefficients for each output sample are calculated on the fly by using linear interpolation between two nearest values. The filter size depends on the resampling factor.

The functions `ippsResamplePolyphaseFixedInitAlloc` create structures for data resampling with the factor equal to  $inRate/outRate$ . If we denote the number of filters created in the *IppsResamplingPolyphaseFixed* structure for input and output frequencies by  $fnum$ , then

$$fnum = outRate / GCD(inRate, outRate),$$

where  $GCD(a, b)$  stands for the greatest common divisor of  $a$  and  $b$ . For example, if  $inRate = 8000$  and  $outRate = 11025$ , then the number of filters will be  $fnum = 11025 / GCD(8000, 11025) = 441$ .

Functions with the `Fixed` suffix in their name pre-calculate filter coefficients for each phase and store them in the data structure for better performance. These functions can be used for cases when the ratio  $inRate/outRate$  is rational only. They can be considerably faster but may require large data structures for some input and output rates.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	For <code>ippsResamplePolyphaseInitAlloc</code> functions: Indicates an error when $inRate$ , $outRate$ , $nStep$ or $len$ is less than or equal to 0. For <code>ippsResamplePolyphaseFixedInitAlloc</code> functions: Indicates an error when $inRate$ , $outRate$ , $nStep$ or $len$ is less than or equal to 0, or when $inRate / LCD * (outRate + 1) / LCD$ is greater than $2^{29}$ (here $LCD$ stands for the lowest common denominator of $inRate$ and $outRate$ ).
<code>ippStsBadArgErr</code>	Indicates an error when $rollf$ is less than or equal to 0 or is greater than 1, or if $alpha$ is less than 1, or if $window$ is less than $2/nStep$ .
<code>ippStsMemAllocErr</code>	Indicates a memory allocation error.

## ResamplePolyphaseFree

Frees structure for polyphase data resampling.

### Syntax

```

IppStatus ippsResamplePolyphaseFree_16s(IppsResamplingPolyphase_16s* pSpec);
IppStatus ippsResamplePolyphaseFree_32f(IppsResamplingPolyphase_32f* pSpec);

```

```
IppStatus ippsResamplePolyphaseFixedFree_16s(IppsResamplingPolyphaseFixed_16s*
pSpec);
```

```
IppStatus ippsResamplePolyphaseFixedFree_32f(IppsResamplingPolyphaseFixed_32f*
pSpec);
```

## Parameters

*pSpec*                                      The pointer to the resampling state structure.

## Description

The functions `ippsResamplePolyphaseFree` and `ippsResamplePolyphaseFixedFree` are declared in the `ippsr.h` file. These functions close the resampling structure by freeing all memory allocated for it by `ippsResamplePolyphaseInitAlloc` or `ippsResamplePolyphaseFixedInitAlloc` functions.

## Return Values

`ippStsNoErr`                              Indicates no error.  
`ippStsNullPtrErr`                      Indicates an error when one of the specified pointers is `NULL`.

# ResamplePolyphase

*Resamples input data using polyphase filters.*

## Syntax

```
IppStatus ippsResamplePolyphase_16s(const IppsResamplingPolyphase_16s* pSpec,
const Ipp16s* pSrc, int len, Ipp16s* pDst, Ipp64f factor, Ipp32f norm, Ipp64f*
pTime, int* pOutlen);
```

```
IppStatus ippsResamplePolyphase_32f(const IppsResamplingPolyphase_32f* pSpec,
const Ipp32f* pSrc, int len, Ipp32f* pDst, Ipp64f factor, Ipp32f norm, Ipp64f*
pTime, int* pOutlen);
```

```
IppStatus ippsResamplePolyphaseFixed_16s(const
IppsResamplingPolyphaseFixed_16s* pSpec, const Ipp16s* pSrc, int len, Ipp16s*
pDst, Ipp32f norm, Ipp64f* pTime, int* pOutlen);
```

```
IppStatus ippsResamplePolyphaseFixed_32f(const
IppsResamplingPolyphaseFixed_32f* pSpec, const Ipp32f* pSrc, int len, Ipp32f*
pDst, Ipp32f norm, Ipp64f* pTime, int* pOutlen);
```

## Parameters

<i>pSpec</i>	The pointer to the resampling state structure.
<i>pSrc</i>	The pointer to the input vector.
<i>pDst</i>	The pointer to the output vector.
<i>len</i>	The number of input vector elements to resample.
<i>norm</i>	The norm factor for output samples.
<i>factor</i>	The resampling factor.
<i>pTime</i>	The pointer to the start time of resampling (in input vector elements). Keeps the input sample number and the phase for the first output sample from the next input data portion.
<i>pOutlen</i>	The number of calculated output vector elements.

## Description

The functions `ippsResamplePolyphase` and `ippsResamplePolyphaseFixed` are declared in the `ippsr.h` file. These functions convert data from the input vector changing their frequency and compute all output samples that can be correctly calculated for the given input and the filter length. The ratio of output and input frequencies is defined by the *factor* argument for the function `ippsResamplePolyphase`. For the function `ippsResamplePolyphaseFixed`, this ratio is defined during creation of the resampling structure. The value `pTime[0]` defines the time value for which the first output sample is calculated.

The history data of filters are in the input vector with indexes less than `pTime[0]`. The history length is equal to  $flen/2$  for function `ippsResamplePolyphaseFixed`, and  $[1/2window * \max(1, 1/factor)] + 1$  for function `ippsResamplePolyphase`. Here *flen* is filter length and *window* is the size of the ideal lowpass filter window (see [ippsResamplePolyphaseInitAlloc](#)). The input vector must contain the same number of elements with indexes greater than `pTime[0] + len` for the right filter wing for the last element.

After function execution, the time value is updated and `pOutlen[0]` contains the number of calculated output samples.

The output samples are multiplied by  $norm * \min(1, factor)$  before saturation.

See Example 8-4.

## Return Values

*ippStsNoErr* Indicates no error.

*ippStsNullPtrErr*  
Indicates an error when *pSpec*, *pSrc*, *pDst*, *pTime* or *pOutlen* pointer is *NULL*.

*ippStsSizeErr*  
Indicates an error when *len* is less than or equal to 0.

*ippStsBadArgErr*  
Indicates an error when *factor* is less than or equal to 0.

**Example 8-4 Resampling of the Input Mono pcm File**

```

void resampleIPP(
    int      inRate,    // input frequency
    int      outRate,   // output frequency
    FILE     *infd,     // input pcm file
    FILE     *outfd)    // output pcm file
{
    short *inBuf,*outBuf;
    int bufsize=4096;
    int history=128;
    double time=history;
    int lastread=history;
    int inCount=0,outCount=0,inLen,outLen;
    IppsResamplingPolyphaseFixed_16s *state;
    ippsResamplePolyphaseFixedInitAlloc_16s(&state,inRate,outRate,2*history,
                                             0.95f,9.0f,ippAlgHintAccurate);
    inBuf=ippsMalloc_16s(bufsize+history+2);
    outBuf=ippsMalloc_16s((int)((bufsize-history)*outRate/(float)inRate+2));
    ippsZero_16s(inBuf,history);
    while ((inLen=fread(inBuf+lastread,sizeof(short),bufsize-lastread,infd))>0) {
        inCount+=inLen;
        lastread+=inLen;
        ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-history-(int)time,
                                         outBuf,0.98f,&time,&outLen);
        fwrite(outBuf,outLen,sizeof(short),outfd);
        outCount+=outLen;
        ippsMove_16s(inBuf+(int)time-history,inBuf,lastread+history-(int)time);
        lastread-=(int)time-history;
    }
}

```

```

        time--=(int)time-history;
    }
    ippsZero_16s(inBuf+lastread,history);
    ippsResamplePolyphaseFixed_16s(state,inBuf,lastread-(int)time,
                                   outBuf,0.98f,&time,&outLen);
    fwrite(outBuf,outLen,sizeof(short),outfd);
    outCount+=outLen;
    printf("%d inputs resampled to %d outputs\n",inCount,outCount);
    ippsFree(outBuf);
    ippsFree(inBuf);
    ippsResamplePolyphaseFixedFree_16s(state);
}

```

## Advanced Aurora Functions

The Intel IPP functions discussed later in this section are implemented to support the speech processing and compression algorithm described in ETSI ES 202 050 V1.1.1 standard (see [ES202]).

The use of these functions is demonstrated in the Intel® IPP Speech Recognition Samples. See *Advanced Aurora Encoder-Decoder Sample* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## SmoothedPowerSpectrumAurora

Calculates smoothed magnitude of the FFT output.

### Syntax

```

IppStatus ippsSmoothedPowerSpectrumAurora_16s(const Ipp16s* pSrc, Ipp16s*
pDst, int len);

IppStatus ippsSmoothedPowerSpectrumAurora_32f(const Ipp32f* pSrc, Ipp32f*
pDst, int len);

IppStatus ippsSmoothedPowerSpectrumAurora_32s64s_Sfs(Ipp32s* pSrc, Ipp64s*
pDst, Ipp32s len, int scaleFactor);

```

```
IppStatus ippsSmoothedPowerSpectrumAurora_32s_Sfs(Ipp32s* pSrc, Ipp32s* pDst,
Ipp32s len, int scaleFactor);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector in PERM format.
<i>pDst</i>	Pointer to the output vector [ <i>len</i> /4+1].
<i>len</i>	The input vector length (multiple of 4).
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsSmoothedPowerSpectrumAurora` is declared in the `ippsr.h` file. This function calculates the smoothed square magnitude for the Fast Fourier Transform (FFT) output vector in PERM format ([[ES202](#)] , 5.1.3-5.1.4).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or <i>len</i> is not a multiple of 4.

## NoiseSpectrumUpdate\_Aurora

*Updates the noise spectrum.*

---

### Syntax

```
IppStatus ippsNoiseSpectrumUpdate_Aurora_32f(const Ipp32f* pSrc, const Ipp32f*
pSrcNoise, Ipp32f* pDst, int len);

IppStatus ippsNoiseSpectrumUpdate_Aurora_16s_Sfs(const Ipp16s* pSrc, const
Ipp16s* pSrcNoise, Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsNoiseSpectrumUpdate_Aurora_32s_Sfs(const Ipp32s* pSrc, const
Ipp32s* pSrcNoise, Ipp32s* pDst, int len, int scaleFactor);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector of the power spectral density mean.
-------------	---



<i>pSrcNoise</i>	Pointer to the input vector of the previous noiseless signal spectrum.
<i>pDst</i>	Pointer to the output vector of the improved transfer function.
<i>len</i>	The length of input and output vectors.
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsNoiseSpectrumUpdate` is declared in the `ippsr.h` file. This function updates the noise spectrum estimate on the second stage of noise reduction according to the ([ES202], 5.1.5).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## WienerFilterDesign\_Aurora

*Calculates an improved transfer function of the adaptive Wiener filter.*

### Syntax

```
IppStatus ippsWienerFilterDesign_Aurora_32f(const Ipp32f* pSrc, const Ipp32f* pNoise, const Ipp32f* pDen, Ipp32f* pDst, int len);
```

```
IppStatus ippsWienerFilterDesign_Aurora_16s(const Ipp16s* pSrc, const Ipp16s* pNoise, const Ipp16s* pDen, Ipp16s* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector of square roots of the power spectral density mean.
<i>pNoise</i>	Pointer to the input vector of square roots of the noise spectrum estimate.
<i>pDen</i>	Pointer to the input vector of square roots of the previous noiseless signal spectrum.

<i>pDst</i>	Pointer to the output vector of the improved transfer function (in Q14).
<i>len</i>	The length of input and output vectors.

### Description

The function `ippsWienerFilterDesign_Aurora` is declared in the `ippsr.h` file. This function calculates an improved transfer function of the adaptive Wiener filter according to the ([ES202], 5.1.5).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsBadArg</code>	Indicates an error when a negative or NaN argument of a square root operation is detected.

## MelBankInitAlloc\_Aurora

*Initializes the structure for performing the Mel-frequency filter bank analysis.*

---

### Syntax

```
IppStatus ippsMelBankInitAllocLow_Aurora_16s(IppsFBankState_16s** pFBank);
IppStatus ippsMelBankInitAllocLow_Aurora_32f(IppsFBankState_32f** pFBank);
IppStatus ippsMelBankInitAllocHigh_Aurora_16s(IppsFBankState_16s** pFBank);
IppStatus ippsMelBankInitAllocHigh_Aurora_32f(IppsFBankState_32f** pFBank);
```

### Parameters

<i>pFBank</i>	Pointer to the Mel-scale filter bank structure to be created.
---------------	---

## Description

The function `ippsMelFBankInitAlloc_Aurora` is declared in the `ippsr.h` file. This function initializes the triangular filter banks for the Mel-frequency filter bank analysis. The function with *Low* suffix builds a filter bank of 23 filters for 8 KHz data processing. The function with *High* suffix builds a filter bank of 3 filters for high frequency band of 16 KHz data processing ([ES202], 5.1.7).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## TabsCalculation\_Aurora

Calculates filter coefficients for residual filter.

### Syntax

```
ippStatus ippsTabsCalculation_Aurora_16s(const Ipp16s* pSrc, Ipp16s* pDst);
ippStatus ippsTabsCalculation_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector of Mel-scaled filter bank output [10].
<i>pDst</i>	Pointer to the output filter coefficients vector [17].

### Description

The function `ippsTabsCalculation_Aurora` is declared in the `ippsr.h` file. This function calculates residual filter coefficients for the Mel-scaled filter bank output ([ES202], 5.1.9-5.1.10).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not equal to 23 or 26.

## ResidualFilter\_Aurora

*Calculates a denoised waveform signal.*

---

### Syntax

```
IppStatus ippsResidualFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
, const Ipp16s* pTabs , int scaleFactor);
```

```
IppStatus ippsResidualFilter_Aurora_32f(const Ipp32f* pSrc , Ipp32f* pDst,
const Ipp32f* pTabs);
```

### Parameters

<i>pTabs</i>	Pointer to the filter coefficients vector [17].
<i>pSrc</i>	Pointer to the input vector [96].
<i>pDst</i>	Pointer to the output vector [80].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsResidualFilter_Aurora` is declared in the `ippsr.h` file. This function filters the input vector to get the low band and high band parts ([\[ES202\]](#), 5.1.10).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## WaveProcessing\_Aurora

*Processes waveform data after noise reduction.*

---

### Syntax

```
IppStatus ippsWaveProcessing_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDst);
```

```
IppStatus ippsWaveProcessing_Aurora_16s(const Ipp16s* pSrc, Ipp16s* pDst);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [200].
-------------	------------------------------------

*pDst* Pointer to the output vector [200].

### Description

The function `ippsWaveProcessing_Aurora` is declared in the `ippsr.h` file. This function processes the input vector according to the ([ES202], 5.2).

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## LowHighFilter\_Aurora

*Calculates low band and high band filters.*

### Syntax

```
IppStatus ippsLowHighFilter_Aurora_16s_Sfs(const Ipp16s* pSrc, Ipp16s*
pDstLow, Ipp16s* pDstHigh, int len, const Ipp16s* pTabs, int tapsLen, int
scaleFactor);
```

```
IppStatus ippsLowHighFilter_Aurora_32f(const Ipp32f* pSrc, Ipp32f* pDstLow,
Ipp32f* pDstHigh, int len, const Ipp32f* pTabs, int tapsLen);
```

### Parameters

*pSrc* Pointer to the input vector [ $len + tapsLen - 1$ ].  
*pDstLow* Pointer to the low frequency band output vector [ $len/2$ ].  
*pDstHigh* Pointer to the high frequency band output vector [ $len/2$ ].  
*len* The input samples number (even).  
*pTabs* Pointer to the filter coefficients vector [ $tapsLen$ ].  
*tapsLen* The filter taps number (even).  
*scaleFactor* Scale factor, refer to [Integer Scaling](#).

### Description

The function `ippsLowHighFilter_Aurora` is declared in the `ippsr.h` file. This function filters the input vector to get the low band and high band parts ([ES202], 5.5.1).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>pDstLow</i> , <i>pDstHigh</i> , or <i>pTabs</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or <i>tapsLen</i> is less than or equal to 0, or <i>len</i> or <i>tapsLen</i> is not even.

## HighBandCoding\_Aurora

*Codes and decodes the high frequency band energy values.*

---

### Syntax

```
IppStatus ippsHighBandCoding_Aurora_32f(const Ipp32f* pSrcHFB, const Ipp32f* pInSWP, const Ipp32f* pDSWP, Ipp32f* pDstHFB);
```

```
IppStatus ippsHighBandCoding_Aurora_32s_Sfs(const Ipp32s* pSrcHFB, const Ipp32s* pInSWP, const Ipp32s* pDSWP, Ipp32s* pDstHFB, int scaleFactor);
```

### Parameters

<i>pSrcHFB</i>	Pointer to the input high frequency band energy vector [3].
<i>pInSWP</i>	Pointer to the input signal smoothed power spectrum vector [65].
<i>pDSWP</i>	Pointer to the denoised signal power spectrum vector [129].
<i>pDstHFB</i>	Pointer to the output coded high frequency band log energy vector [3].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The function `ippsHighBandCoding_Aurora` is declared in the `ippsr.h` file. This function performs coding and decoding of high frequency band energies according to the ([ES202], 5.5.3).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBadArg</code>	Indicates an error when a non-positive or <code>NaN</code> argument of the logarithm operation is detected.

## BlindEqualization\_Aurora

*Equalizes the cepstral coefficients.*

### Syntax

```
IppStatus ippsBlindEqualization_Aurora_32f(const Ipp32f* pRefs, Ipp32f*
pCeps, Ipp32f* pBias, int len, Ipp32f val);

IppStatus ippsBlindEqualization_Aurora_16s(const Ipp16s* pRefsQ6, Ipp16s*
pCeps, Ipp16s* pBias, int len, Ipp32s valQ6);
```

### Parameters

<i>pRefs</i>	Pointer to the input vector of reference cepstrum [ <i>len</i> ].
<i>pCeps</i>	Pointer to the input and output vector of cepstrum [ <i>len</i> ] .
<i>pBias</i>	Pointer to the input and output vector of bias [ <i>len</i> ] .
<i>len</i>	The number of cepstral coefficients.
<i>val</i>	The log energy value.

### Description

The function `ippsBlindEqualization_Aurora` is declared in the `ippsr.h` file. This function equalizes cepstral coefficients using the LMS algorithm ([ES202], 5.4).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## DeltaDelta\_Aurora

*Calculates the first and second derivatives according to ETSI ES 202 050 standard.*

---

### Syntax

```
IppStatus ippsDeltaDelta_Aurora_16s_D2Sfs(const Ipp16s* pSrc, Ipp16s* pDst,
int dstStep, int height, int deltaMode, int scaleFactor);
```

```
IppStatus ippsDeltaDeltaMul_Aurora_16s_D2Sfs(const Ipp16s* pSrc, const Ipp16s*
pVal, Ipp16s* pDst, int dstStep, int height, int deltaMode, int scaleFactor);
```

```
IppStatus ippsDeltaDelta_Aurora_32f_D2(const Ipp32f* pSrc, Ipp32f* pDst, int
dstStep, int height, int deltaMode);
```

```
IppStatus ippsDeltaDeltaMul_Aurora_32f_D2(const Ipp32f* pSrc, const Ipp32f*
pVal, Ipp32f* pDst, int dstStep, int height, int deltaMode);
```

### Parameters

<i>pSrc</i>	Pointer to the input feature sequence [ <i>height</i> *14].
<i>pDst</i>	Pointer to the output feature sequence [ <i>height</i> * <i>dstStep</i> ].
<i>dstStep</i>	Length of the output feature in the output sequence <i>pDst</i> .
<i>height</i>	Number of feature vectors.
<i>deltaMode</i>	Execution mode.
<i>pVal</i>	Pointer to the delta coefficients vector [39].
<i>scaleFactor</i>	Scale factor, refer to <a href="#">Integer Scaling</a> .

### Description

The functions `ippsDeltaDelta_Aurora` and `ippsDeltaDeltaMul_Aurora` are declared in the `ippsr.h` file. These functions calculate full feature vectors according to ETSI ES 202 050 standard. The input vectors of length 14 contain values  $c_1, \dots, c_{12}, c_0, \ln E$ . First, the input feature vectors are copied to the output sequence. Then the first and second derivatives are calculated in accordance with ([ES202], 9.1-9.2).

The function implies the following constraints on the delta coefficients:



$val[j] = pVal[j]$  for `ippsDeltaDelta_Aurora` function,  $val[j] = 1$  for `ippsDeltaDeltaMul_Aurora` function.

$vel[-4] = -1.0, vel[-3] = -0.75, vel[-2] = -0.5, vel[-1] = -0.25, vel[0] = 0.0, vel[1] = 0.25, vel[2] = 0.5, vel[3] = 0.75, vel[4] = 1.0$ .

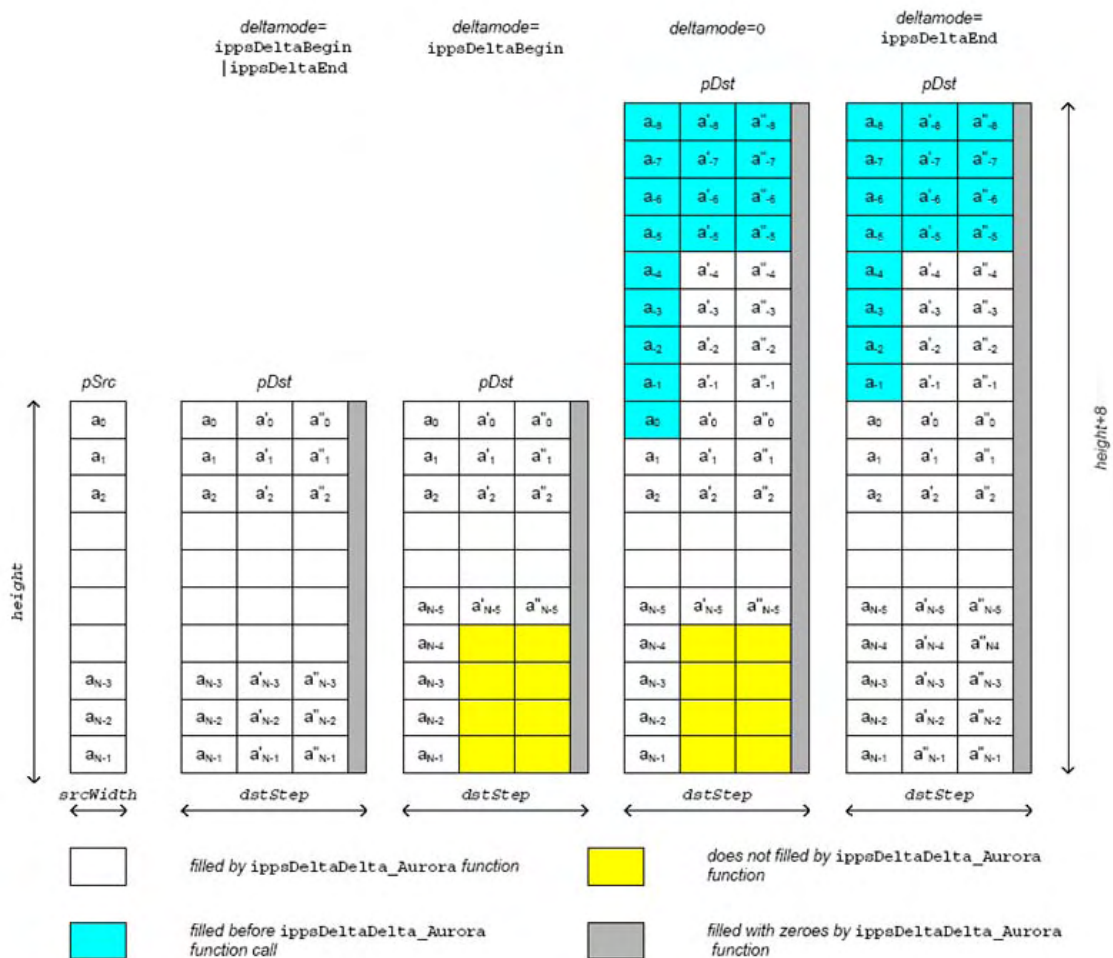
$acc[-4] = 1.0, acc[-3] = 0.25, acc[-2] = -0.285714, acc[-1] = -0.607143, acc[0] = -0.714286, acc[1] = -0.607143, acc[2] = -0.285714, acc[3] = 0.25, acc[4] = 1.0$ .

The execution mode *deltaMode* provides additional controls for the base feature copy and derivative calculation process. The admissible values of *deltaMode* and the corresponding function execution logic are the following:

1. *deltaMode* is equal to `IPP_DELTA_BEGIN|IPP_DELTA_END`
2. *deltaMode* is equal to 0
3. *deltaMode* is equal to `IPP_DELTA_BEGIN`
4. *deltaMode* is equal to `IPP_DELTA_END`

Figure 8-6 illustrates the four delta calculation modes.

**Figure 8-6 Execution Modes of ippsDeltaDelta\_Aurora function**



## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is NULL.

<code>ippStsSizeErr</code>	Indicates an error when <i>height</i> is less than 0; or <i>height</i> is less than 8 when <i>deltaMode</i> is equal to <code>IPP_DELTA_BEGIN</code> ; or <i>height</i> is equal to 0 when <i>deltaMode</i> is not equal to <code>IPP_DELTA_END</code> .
<code>ippStsStrideErr</code>	Indicates an error when <i>dstStep</i> is less than 39.

## VADGetBufSize\_Aurora

Queries the memory size for VAD decision.

### Syntax

```
IppStatus ippSVADGetBufSize_Aurora_32f(int* pSize);
IppStatus ippSVADGetBufSize_Aurora_16s(int* pSize);
```

### Parameters

*pSize*                      Pointer to the output value of the memory size needed for VAD decision.

### Description

The function `ippSVADGetBufSize_Aurora` is declared in the `ippsr.h` file. This function returns the size of memory that should be allocated by user. The memory block of `pSize[0]` bytes is used for VAD algorithm initialization by `ippSVadInit_Aurora` function.

### Return Values

`ippStsNoErr`              Indicates no error.  
`ippStsNullPtrErr`       Indicates an error when *pSize* pointer is `NULL`.

## VADInit\_Aurora

Gets the VAD structure size.

### Syntax

```
IppStatus ippSVADInit_Aurora_32f(char* pVADmem);
IppStatus ippSVADInit_Aurora_16s(char* pVADmem);
```

## Parameters

*pVADmem* Pointer to the VAD decision memory.

## Description

The function `ippsVADInit_Aurora` is declared in the `ippsr.h` file. This function initializes the VAD decision process.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when *pVADmem* pointer is NULL.

## VADDecision\_Aurora

*Takes the VAD decision.*

---

## Syntax

```

IppStatus ippsVADDecision_Aurora_32f(const Ipp32f* pCoeff, const Ipp32f*
pTrans, IppVADDecision_Aurora* pRes, int nbSpeechFrame, char* pVADmem);

IppStatus ippsVADDecision_Aurora_16s(const Ipp16s* pCoeff, const Ipp16s*
pTrans, IppVADDecision_Aurora* pRes, int nbSpeechFrame, char* pVADmem);

```

## Parameters

*pCoeff* Pointer to the input vector of Mel-warped Wiener filter coefficients [25].  
*pTrans* Pointer to the input vector of Wiener filter transfer function [64].  
*pRes* Pointer to VAD decision ("1" if voice is detected, "0" otherwise).  
*nbSpeechFrame* Speech frame hangover counter.  
*pVADmem* Pointer to the VAD decision memory.

## Description

The function `ippsVADDecision_Aurora` is declared in the `ippsr.h` file. This function takes VAD decision for the input frame according to ([[ES202](#)], Annex A).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## VADFlush\_Aurora

*Takes VAD decision for zero input frame.*

---

### Syntax

```
IppStatus ippSVADFlush_Aurora_32f(IppVADDecision_Aurora* pRes, char* pVADmem);
IppStatus ippSVADFlush_Aurora_16s(IppVADDecision_Aurora* pRes, char* pVADmem);
```

### Parameters

<i>pCoeff</i>	Pointer to the input vector of Mel-warped Wiener filter coefficients [25].
<i>pTrans</i>	Pointer to the input vector of Wiener filter transfer function [64].
<i>pRes</i>	Pointer to VAD decision ("1" if voice is detected, "0" otherwise).
<i>pVADmem</i>	Pointer to the VAD decision memory.

### Description

The function `ippSVADDecision_Aurora` is declared in the `ippsr.h` file. This function takes VAD decision for the zero frame. It is used when speech is finished.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Ephraim-Malah Noise Suppressor

This section describes the Intel IPP functions that can be combined to construct an implementation of the Ephraim-Malah Noise Suppressor (EMNS) originally described in [Eph84]. The primitives are primarily concerned with the well-defined, computationally expensive core operations.

The EMNS is a frequency domain noise suppression algorithm that seeks to minimize the minimum mean squared error of the speech short-time spectral amplitude estimate.

Intel IPP EMNS primitives support the following features:

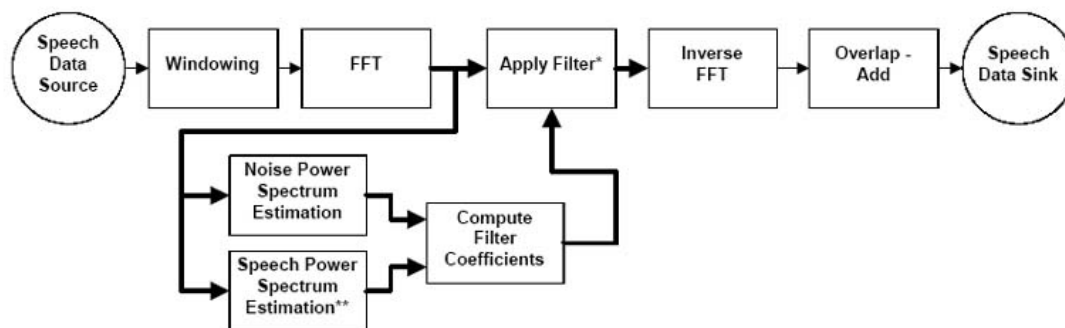
- Filter update
- Noise floor estimation

The use of these functions is demonstrated in the Intel® IPP Speech Recognition Samples. See *Audio Processing Sample* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

### Noise Suppressor Architecture

The major building blocks of the Ephraim-Malah Noise Suppressor are shown in Figure 8-7, where the steps are given to apply noise reduction to a speech signal stream.

**Figure 8-7 Major blocks in an Ephraim-Malah Noise Suppression System**



\* applying the filter consists of multiplying each FFT bin by a real-valued filter coefficient

\*\* speech power spectral estimation is typically accomplished via spectral subtraction

## Data Structures

There is one structure associated with the MCRA noise floor estimator: `IppsMCRAParam`. This structure is used internally and can not be modified by the programmer.

## Filter Update Functions

### FilterUpdateEMNS

Calculates the noise suppression filter coefficients.

#### Syntax

```
IppStatus ippsFilterUpdateEMNS_32s(const Ipp32s* pSrcWienerCoefsQ31 , const
Ipp32s* pSrcPostSNRQ15, Ipp32s* pDstFilterCoefsQ31, int len);

IppStatus ippsFilterUpdateEMNS_32f(const Ipp32f* pSrcWienerCoefs, const
Ipp32f* pSrcPostSNR, Ipp32f* pDstFilterCoefs, int len);
```

#### Parameters

<i>pSrcWienerCoefsQ31</i>	Pointer to a real-valued vector containing the Q31 format Wiener filter coefficients $[0.0_{Q31} .. 1.0_{Q31}]$ .
<i>pSrcWienerCoefs</i>	Pointer to a real-valued vector containing Wiener filter coefficients $[0.0 .. 1.0]$ .
<i>pSrcPostSNRQ15</i>	Pointer to a real-valued vector containing an estimate of the a posteriori signal to noise ratio in Q15 format $(0_{Q15} .. 32768.0_{Q15})$ .
<i>pSrcPostSNR</i>	Pointer to a real-valued vector containing an estimate of the a posteriori signal to noise ratio $(0 .. 32768.0)$ .
<i>pDstFilterCoefsQ31</i>	Pointer to a real-valued vector containing the Q31 format filter coefficients $[0.0_{Q31} .. 1.0_{Q31}]$ .
<i>pDstFilterCoefs</i>	Pointer to a real-valued vector containing the filter coefficients $[0.0 .. 1.0]$ .
<i>len</i>	Number of elements contained in input and output vectors $(0 .. 65536)$ .

## Description

The function `ippsFilterUpdateEMNS` is declared in the `ippsr.h` file. This function calculates the noise suppression filter coefficients. Three filter sizes are typically used: 65, 129, and 257 (corresponding to FFT sizes of 128, 256, and 512). These are recommended for sample rates  $F_s \leq 11025$  Hz,  $11025 \text{ Hz} < F_s \leq 22050$  Hz, and  $22050 \text{ Hz} < F_s \leq 44100$  Hz, respectively. The noise suppression filter coefficients are the gains (scalar values between zero and one) that are applied to each FFT bin.

Power spectral estimates are typically computed using the function `ippsAddWeighted`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error when <code>len</code> has an illegal value.

## FilterUpdateWiener

*Calculates the Wiener filter coefficients.*

---

### Syntax

```
IppStatus ippsFilterUpdateWiener_32s(const Ipp32s* pSrcPriorSNRQ15 , Ipp32s*
pDstFilterCoefsQ31, int len);
```

```
IppStatus ippsFilterUpdateWiener_32f(const Ipp32f* pSrcPriorSNR, Ipp32f*
pDstFilterCoefs, int len);
```

### Parameters

<code>pSrcPriorSNRQ15</code>	Pointer to a real-valued Q15 format vector containing an estimate of the a priori signal to noise ratio ( $0.0 \text{ Q15} < pSrcPriorSNRQ15[k] < 32768.0 \text{ Q15}$ ).
<code>pSrcPriorSNR</code>	Pointer to a real-valued vector containing an estimate of the a priori signal to noise ratio ( $0.0 < pSrcPriorSNRQ15[k] < 32768.0$ ).
<code>pDstFilterCoefsQ31</code>	Pointer to a real-valued vector containing the Q31 format filter coefficients ( $0.0 \text{ Q31} < pDstFilterCoefsQ31[k] < 1.0 \text{ Q31}$ ).



<i>pDstFilterCoefs</i>	Pointer to a real-valued vector containing the filter coefficients ( $0.0 < pDstFilterCoefsQ31[k] < 1.0$ ).
<i>len</i>	Number of elements contained in input and output vectors ( $0 < len < 65536$ ).

## Description

The function `ippsFilterUpdateWiener` is declared in the `ippsr.h` file. This function calculates the Wiener filter coefficients. Three filter sizes are typically used: 65, 129, and 257 (corresponding to FFT sizes of 128, 256, and 512). These are recommended for sample rates  $F_s \leq 11025$  Hz,  $11025 \text{ Hz} < F_s \leq 22050$  Hz, and  $22050 \text{ Hz} < F_s \leq 44100$  Hz, respectively. The Wiener filter coefficients are the gains (scalar values between zero and one) that are applied to each FFT bin.

The function `ippsFilterUpdateWiener` implements a coarse approximation in order to achieve very low execution time. This coarse approximation was found to be sufficient for most noise reduction applications. However, if high accuracy is desired one may choose to call `ippsDiv_32s_Sfs` with the proper parameters instead.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrcPriorSNRQ15</i> , <i>pDstFilterCoefsQ31</i> , <i>pSrcPriorSNR</i> , or <i>pDstFilterCoefs</i> pointer is NULL.
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> has an illegal value.
<code>ippStsBadArgErr</code>	Indicates an error when at least one of the 'a priori' SNR array elements is negative. Zero filter coefficient is returned for such elements.

## Noise Floor Estimation Functions

### GetSizeMCRA

*Calculates the size in bytes required for the MCRA state structure.*

---

#### Syntax

```
IppStatus ippsGetSizeMCRA_32s(int nFFTSIZE, int* pDstSize);  
IppStatus ippsGetSizeMCRA_32f(int nFFTSIZE, int* pDstSize);
```

#### Parameters

<i>nFFTSIZE</i>	Size of the FFT used for noise PSD estimations [8,..,8192]
<i>pDstSize</i>	Pointer to the variable to contain the size in bytes.

#### Description

The function `ippsGetSizeMCRA` is declared in the `ippsr.h` file. This function calculates the size in bytes of the MCRA state structure required by the `ippsUpdateNoisePSDMCRA` function. The function `ippsGetSizeMCRA` must be called before allocating memory and before calling `ippsInitMCRA`.

#### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDstSize</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>nFFTSIZE</i> has an illegal value.

### InitMCRA

*Initializes the MCRA state structure.*

---

#### Syntax

```
IppStatus ippsInitMCRA_32s(int nSamplesPerSec, int nFFTSIZE, IppMCRAState*  
pDst);
```

```
IppStatus ippsInitMCRA_32f(int nSamplesPerSec, int nFFTSIZE, IppMCRAState32f*  
pDst);
```

## Parameters

<i>nSamplesPerSec</i>	Input sample rate [8000,...,48000].
<i>nFFTSIZE</i>	Size of the FFT used to for noise PSD estimation [8,..., 8192].
<i>pDst</i>	Pointer to the MCRA state structure.

## Description

The function `ippsInitMCRA` is declared in the `ippsr.h` file. This function initializes the state structure for the `ippsUpdateNoisePSDMCRA` function.



**NOTE.** State memory address stored in *pDst* must be aligned to 32-bit word boundary.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>nSamplesPerSec</i> is out of range. .
<code>ippStsSizeErr</code>	Indicates an error when <i>nFFTSIZE</i> has an illegal value.

## AltInitMCRA

*Allocates memory and initializes the `IppMCRAState` state structure.*

## Syntax

```
IppStatus ippsAltInitMCRA_32s(int nSamplesPerSec , int nFFTSIZE,  
IppMCRAState** ppDst);  
  
IppStatus ippsAltInitMCRA_32f(int nSamplesPerSec, int nFFTSIZE, int  
nUpdateSamples, IppMCRAState32f* pDst);
```

### Parameters

<i>nSamplesPerSec</i>	Input sample rate (0..48000].
<i>nFFTSize</i>	Size of the FFT used to for noise PSD estimation [8..8192].
<i>nUpdateSamples</i>	Number of new samples per frame.
<i>ppDst</i>	Pointer to the pointer to the state structure.
<i>pDst</i>	Pointer to the state structure.

### Description

The function `ippsInitAllocMCRA` is declared in the `ippsr.h` file. This function allocates memory and initializes the state structure for the `ippsUpdateNoisePSDMCRA` function.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDst</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>nSamplesPerSec</i> is out of range.
<code>ippStsSizeErr</code>	Indicates an error when <i>nFFTSize</i> has an illegal value.

## UpdateNoisePSDMCRA

*Re-estimates the noise power spectrum.*

---

### Syntax

```
ippStatus ippsUpdateNoisePSDMCRA_32s_I(const Ipp32s* pSrcNoisySpeech,
    IppMCRAState* pSrcDstState, Ipp32s* pSrcDstNoisePSD);
```

```
ippStatus ippsUpdateNoisePSDMCRA_32f_I(const Ipp32f* pSrcNoisySpeech,
    IppMCRAState32f* pSrcDstState, Ipp32f* pSrcDstNoisePSD);
```

### Parameters

<i>pSrcNoisySpeech</i>	Pointer to a real-valued vector containing the magnitude squared of the FFT of the noisy speech; must be greater than or equal to zero.
<i>pSrcDstState</i>	Pointer to the state structure.
<i>pSrcDstNoisePSD</i>	Pointer to the noise power spectrum vector.

### Description

The function `ippsUpdateNoisePSDMCRA` is declared in the `ippsr.h` file. This function re-estimates the noise power spectrum given a new measurement of the magnitude squared noisy speech. The algorithm is based on the Minima Controlled Recursive Averaging approach described in [Coh02].

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Voice Activity Detector

This section describes the Intel IPP functions that can be combined to construct a *voice activity detector* (VAD). The functions are primarily concerned with the well-defined, computationally expensive core operations.

The VAD consists of parameter estimation and speech modeling heuristics.

IPP VAD primitives support the following features:

- Peak picking
- Periodicity
- Zero crossing rate

The use of these functions is demonstrated in the Intel® IPP Speech Recognition Samples. See *Audio Processing Sample* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

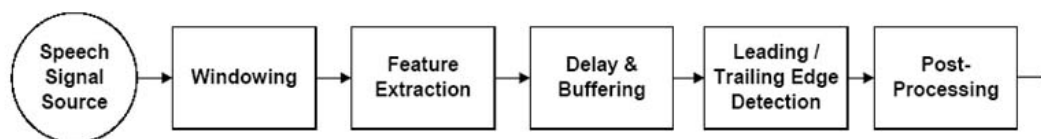
### Voice Activity Detector Architecture

A typical voice activity detector is illustrated below. The speech signal is windowed and separated into (possibly overlapping) frames. Feature extraction is then performed. Feature extraction may include one or more of the following: zero crossing rate calculation, energy calculation, segmental SNR estimation, periodicity detection, and sample or sub-band histogram measurement. Measurements are buffered so that speech onset and ending can be distinguished

from other spurious events. A delay of 400 milliseconds is typical. Speech leading and trailing edge detection is carried out based on heuristic rules. Post-processing (for example, median filtering) may be applied in low-delay implementations to eliminate false detections.

**Figure 8-8. Major blocks in a Voice Activity Detector**

---



## Voice Activity Detection Functions

### FindPeaks

*Identifies peaks in the input vector.*

---

#### Syntax

```
IppStatus ippsFindPeaks_32s8u(const Ipp32s* pSrc, Ipp8u* pDstPeaks, int len,
int searchSize, int movingAvgSize);
```

```
IppStatus ippsFindPeaks_32f8u(const Ipp32f* pSrc, Ipp8u* pDstPeaks, int len,
int searchSize, int movingAvgSize);
```

#### Parameters

<i>pSrc</i>	Pointer to an input vector.
<i>pDstPeaks</i>	Pointer to the output vector containing a one in positions corresponding to a peak in the input vector and zeros elsewhere.
<i>len</i>	Number of elements contained in the input and output vectors ( $0 < len < 65536$ ).

<i>searchSize</i>	Number of elements on either side to consider when picking peak ( $0 < searchSize < 128$ ).
<i>movingAvgSize</i>	Number of elements on either side to include in moving average window that is applied before peak picking ( $0 \leq movingAvgSize < 128$ ).

## Description

The function `ippsFindPeaks` is declared in the `ippsr.h` file. This function identifies peaks in the input vector, places a one in the output vector at the locations of the peaks, and places a zero elsewhere. A peak is defined as a point  $pSrc[i]$  such that  $pSrc[i-L] < pSrc[i-L+1] < \dots < pSrc[i-1] < pSrc[i] > \dots > pSrc[i+L-1] > pSrc[i+L]$ , where *searchSize* is L. If *movingAvgSize* is greater than 0, then the source vector is smoothed via moving average before peaks are selected.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> is out of range.
<code>ippStsSizeErr</code>	Indicates an error when <i>SearchSize</i> or <i>movingAvgSize</i> is out of range.

## PeriodicityLSPE

Computes the periodicity of the input speech frame.

### Syntax

```

IppStatus ippsPeriodicityLSPE_16s(const Ipp16s* pSrc, int len, Ipp16s*
pPeriodicityQ15, int* period, int maxPeriod, int minPeriod);

IppStatus ippsPeriodicityLSPE_32f(const Ipp32f* pSrc, int len, Ipp32f*
pPeriodicity, int* period, int maxPeriod, int minPeriod);

```

### Parameters

<i>pSrc</i>	Pointer to an input speech vector.
<i>len</i>	Number of elements contained in the input vector subject to $6 < len \leq \min(16 * minPeriod, 1024)$

<i>pPeriodicityQ15</i>	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ( $0.0_{Q15} \leq pPeriodicityQ15 \leq 1.0_{Q15}$ ).
<i>pPeriodicity</i>	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ( $0.0 \leq pPeriodicity \leq 1.0$ ).
<i>period</i>	The period (in samples) that minimizes the LSPE cost function.
<i>maxPeriod</i>	Maximum period to search ( $minPeriod < maxPeriod < len$ ).
<i>minPeriod</i>	Minimum period to search ( $6 \leq minPeriod < maxPeriod$ ).

## Description

The function `ippsPeriodicityLSPE` is declared in the `ippsr.h` file. This function computes the periodicity of the input speech frame. The periodicity is calculated according the least squares periodicity estimate (LSPE) algorithm defined in [Tuc92]. The periodicity search is designed for speech signals with sample rate between 2000-24000 Hz and may not perform well on music or other sources. If only periodicity (voicing) is required, it is more efficient to downsample input speech before calling this primitive. For example, at 2000 Hz sample rate, the settings `minPeriod=10`, `maxPeriod=20`, and `len=64`, provide adequate periodicity for some applications.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error when <code>len</code> is out of range.
<code>ippStsRangeErr</code>	Indicates an error when <code>maxPeriod</code> or <code>minPeriod</code> is out of range.



## Periodicity

Computes the periodicity of the input block.

### Syntax

```
IppStatus ippsPeriodicity_32s16s(const Ipp32s* pSrc, int len, Ipp16s*
pPeriodicityQ15, int* period, int maxPeriod, int minPeriod);
```

```
IppStatus ippsPeriodicity_32f (const Ipp32f* pSrc, int len, Ipp32f*
pPeriodicity, int* period, int maxPeriod, int minPeriod);
```

### Parameters

<i>pSrc</i>	Pointer to an input speech vector.
<i>len</i>	Number of elements contained in the input vector subject to $6 < len \leq \min(16 * minPeriod, 1024)$
<i>pPeriodicityQ15</i>	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ( $0.0_{Q15} \leq pPeriodicityQ15 \leq 1.0_{Q15}$ ).
<i>pPeriodicity</i>	Pointer to the Q15 format value corresponding to the normalized sum of the largest periodic sampling ( $0.0$ $\leq pPeriodicity \leq 1.0$ ).
<i>period</i>	The period (in samples) that minimizes the LSPE cost function.
<i>maxPeriod</i>	Maximum period to search ( $minPeriod < maxPeriod$ $< len$ ).
<i>minPeriod</i>	Minimum period to search ( $6 \leq minPeriod <$ $maxPeriod$ ).

### Description

The function `ippsPeriodicity` is declared in the `ippsr.h` file. This function computes the periodicity of the input block. In typical applications, the input block is the magnitude-squared of the discrete Fourier transform of windowed speech. The periodicity is defined as the periodic sampling of the input block that preserves the most energy.

$$\max_{k, T_0} \sum_n x(k + nT_0) \quad \text{where } 0 < k \leq T_0$$

Bias removal is performed prior to the search to ensure an accurate measurement.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> is out of range.
<code>ippStsRangeErr</code>	Indicates an error when <i>pSrc</i> [ <i>k</i> ], <i>maxPeriod</i> , or <i>minPeriod</i> is out of range.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> , <i>period</i> , or <i>pPeriodicityQ15</i> pointer is <code>NULL</code> .

# Speech Coding Functions

This chapter describes the Intel® IPP functions that can be used for implementing speech codecs which follow ITU-T recommendations for G711 (companding functions), G.722, G.722.1, G.723.1, G.726, G.728, G.729.1 and G.729 codecs, the G.167, G.168 for Echo Cancellor, G.169 for Audio Level control, ETSI specifications for GSM-AMR and GSM-FR codecs, as well as 3GPP specification for AMRWB and AMRWB+ codecs, and Microsoft\* Real-Time Audio codec.

When properly built, such speech codecs and other components can be compliant with the bit-exact and other specifications for published test vectors.

The chapter also includes the functions that solely or in combination with other functions can be used for speech/voice quality enhancement, for example, in acoustic or network echo control, background noise artifacts removal, automatic audio level control. In tandem with the speech codec they can increase quality and save bandwidth of voice communication.

The organization of this chapter is the following. The introductory part includes description of rounding modes, defines notational conventions, header files and data structures used by the Intel IPP speech coding functions. This is succeeded by the major functionality sections, which include:

- [Common Functions](#)
- [G.729 Functions](#)
- [G.729.1 Functions](#)
- [G.723.1 Functions](#)
- [GSM-AMR Functions](#)
- [AMR Wideband Functions](#)
- [AMR Wideband Plus Functions](#)
- [GSM Full Rate Functions](#)
- [G.722.1 Functions](#)
- [G.726 Functions](#)
- [G.728 Functions](#)
- [Voice Enhancement Functions](#)
- [G722 Sub-Band ADPCM Speech Codec Functions](#)
- [Companding Functions](#)
- [RT Audio Functions](#)

Each section starts with the table that gives the full list of Intel IPP functions specific for that functional group, followed by detailed description of the respective API.

The use of the Intel IPP speech coding functions is demonstrated in Intel IPP Samples. See *Intel IPP Speech Coding Samples* downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## Rounding Mode

As many speech codecs have to meet the bit-exact requirement, Intel IPP functions described in this chapter use rounding modes that are different from the default rounding mode used in general signal processing functions. For general signal processing functions, the default rounding mode can be described as “nearest even”, so that the fixed point number  $x=N + \alpha$ ,  $0 \leq \alpha < 0.5$ , where  $N$  is an integer number, is rounded as given by:

$$\lceil x \rceil = \begin{cases} N, & 0 \leq \alpha < 0.5 \\ N+1, & 0.5 < \alpha < 1 \\ N, & \alpha = 0.5, N-\text{even} \\ N+1, & \alpha = 0.5, N-\text{odd} \end{cases}$$

For example, 1.5 will be rounded to 2 and 2.5 to 2.

For functions in this chapter, there are two rounding modes.

The default rounding mode is “clipping”, so that the fractional part of the fixed point number is cut off and the result of rounding is always less than the initial value. Specifically, the fixed point number  $x=N + \alpha$ ,  $0 \leq \alpha < 0.5$ , where  $N$  is an integer number, is always rounded to  $N$ . For example, -1.3 will be rounded to -2 and 1.7 to 1. No special suffix is added to the names of functions that use this default mode.

Another rounding mode is “nearest right”, in which the fixed point number  $x=N + \alpha$ ,  $0 \leq \alpha < 0.5$ , where  $N$  is an integer number, is rounded as follows:

$$\lceil x \rceil = \begin{cases} N, & 0 \leq \alpha < 0.5 \\ N+1, & 0.5 \leq \alpha < 1 \end{cases}$$

For example, 1.5 will be rounded to 2 (same as for “nearest even” mode) but -1.5 to -1 (different from “nearest even” mode where -1.5 is rounded to -2).

The suffix “NR” is added to names of functions that use the “nearest right” rounding mode.

## Notational Conventions

This chapter, in addition to conventions used throughout the manual, uses the following notational conventions:

- In the description of function arguments, when an argument refers to a vector, the expression  $[n]$  in square brackets that may be given after the explanation of the argument specifies the number of elements (length) of that vector;
- Most of the speech coding functions for their proper execution interpret their integer and integer array arguments as fixed point numbers, which represent real numbers that vary in their specific ranges. Notation  $Q_n$  used in the argument description means that this argument values are used in integer calculations inside the function as real numbers equal to the integer value multiplied by  $2^{-n}$  (where “ $n$ ” is called a scale factor). For example, if an argument value is described as “4096 in  $Q_{12}$ ”, then it is interpreted as the real number 1.0; the value described as “15565 with scale factor 14” represents the real number 0.95; an argument described as “ $Q_{14}$  in  $[0, 1]$ ” must be passed as an integer value in the range  $[0, 16386]$ .

## Definitions

This section identifies the header files required for using the Intel IPP speech coding API described later in this chapter. For the definition of bit rate specifiers, refer to [Data Structures](#).

The header files `ippdefs.h` and `ippsc.h` must be included in order to link against any of the speech coding primitives, as shown in the following example code:

```
#include "ippdefs.h" #include "ippsc.h"

int main()
{
    ...

    /* call GSM-AMR IPP functions */

    ippLevinsonDurbin_GSMAMR(pSrcAutoCorr, pSrcDstLpc);

    ...
}
```

## Data Structures

Some of the speech coding functions use a bit rate parameter of the enumerated type `IppSpchBitRate`. The set contains one specifier for each of the supported bit rates, as given in the structure below:

```
typedef enum {
    IPP_SPCHBR_4750
    IPP_SPCHBR_5150
    IPP_SPCHBR_5300
    IPP_SPCHBR_5900
    IPP_SPCHBR_6300
    IPP_SPCHBR_6600
    IPP_SPCHBR_6700
    IPP_SPCHBR_7400
    IPP_SPCHBR_7950
    IPP_SPCHBR_8850
    IPP_SPCHBR_9600
    IPP_SPCHBR_10200
    IPP_SPCHBR_12200
    IPP_SPCHBR_12650
    IPP_SPCHBR_12800
    IPP_SPCHBR_14250
    IPP_SPCHBR_15850
    IPP_SPCHBR_16000
    IPP_SPCHBR_18250
    IPP_SPCHBR_19850
    IPP_SPCHBR_23050
    IPP_SPCHBR_23850
    IPP_SPCHBR_24000
    IPP_SPCHBR_32000
    IPP_SPCHBR_40000
    IPP_SPCHBR_DTX
} IppSpchBitRate;
```

The specifiers `IPP_SPCHBR_4750`, `IPP_SPCHBR_5150`, `IPP_SPCHBR_5900`, `IPP_SPCHBR_6700`, `IPP_SPCHBR_7400`, `IPP_SPCHBR_7950`, `IPP_SPCHBR_10200` and `IPP_SPCHBR_12200` are used in GSM-AMR functions and correspond to 4.75, 5.15, 5.9, 6.7, 7.4, 7.95, 10.2 and 12.2 Kbits/s transmitting rates, respectively. The specifiers `IPP_SPCHBR_5300` and `IPP_SPCHBR_6300` are used in G.723.1 codec and correspond to 5.3 (low) and 6.3 (high) Kbits/s transmitting rates. The specifiers `IPP_SPCHBR_9600`, `IPP_SPCHBR_12800` and `IPP_SPCHBR_16000` are used in G.728 codec and correspond to 9.6, 12.8 and 16 Kbits/s transmitting rates.

The specifiers `IPP_SPCHBR_16000`, `IPP_SPCHBR_24000`, `IPP_SPCHBR_32000` and `IPP_SPCHBR_40000` are used in G.726 codec and correspond to 16, 24, 32 and 40 Kbits/s transmitting rates, respectively.

The specifiers `IPP_SPCHBR_6600`, `IPP_SPCHBR_8850`, `IPP_SPCHBR_12650`, `IPP_SPCHBR_14250`, `IPP_SPCHBR_15850`, `IPP_SPCHBR_18250`, `IPP_SPCHBR_19850`, `IPP_SPCHBR_23050`, and `IPP_SPCHBR_23850` are used in AMR WB codec and correspond to 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 Kbits/s transmitting rates, respectively.

## Common Functions

This section describes common functions used in different speech codecs. The list of these functions is given in Table 9-1.

**Table 9-1. Intel IPP Common Speech Coding Functions**

Function Base Name	Operation
<code>ConvPartial</code>	Performs linear convolution of 1D signals.
<code>InterpolateC_NR</code>	Computes the weighted sum of two vectors.
<code>Mul_NR</code>	Multiplies the elements of two vectors.
<code>MulC_NR</code>	Multiplies each element of a vector by a constant value.
<code>MulPowerC_NR</code>	Performs power weighting for each element of a vector.
<code>AutoScale</code>	Scales by the maximal elements.
<code>DotProdAutoScale</code>	Computes the dot product of two vectors using the automatic scaling.
<code>InvSqrt</code>	Computes inverse square root of vector elements.
<code>AutoCorr</code>	Calculates autocorrelation of a vector.
<code>AutoCorrLagMax</code>	Estimates the maximum auto-correlation of a vector.
<code>AutoCorr_NormE</code>	Estimates normal auto-correlation of a vector.
<code>CrossCorr</code>	Estimates the cross-correlation of two vectors.
<code>CrossCorrLagMax</code>	Estimates the maximum cross-correlation between two vectors.
<code>SynthesisFilter</code>	Computes the speech signal by filtering the input speech through the synthesis filter $1/A(z)$ .

## ConvPartial

*Performs linear convolution of 1D signal.*

---

### Syntax

```

IppStatus ippsConvPartial_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsConvPartial_16s32s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
ipp32s* pDst, int len);

IppStatus ippsConvPartial_NR_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
Ipp16s* pDst, int len);

IppStatus ippsConvPartial_NR_Low_16s (const Ipp16s* pSrc1, const Ipp16s*
pSrc2, Ipp16s* pDst, int len);

```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the source and destination vectors.
<i>scaleFactor</i>	Scale factor used for output data scaling.

### Description

The function `ippsConvPartial` is declared in the `ippsc.h` file. The function flavor `ippsConvPartial_16s32s` computes the convolution of the vectors *pSrc1* and *pSrc2* as:

$$pDst[i] = \sum_{j=0}^i pSrc1[i] \cdot pSrc2[i-j] , \quad i = 0, \dots, len-1.$$

The function flavor `ippsConvPartial_16s_Sfs` computes the convolution of the vectors *pSrc1* and *pSrc2* and scales output data as given by:



$$pDst[i] = 2^{-scaleFactor} \cdot \sum_{j=0}^i pSrc1[i] \cdot pSrc2[i-j] , \quad i = 0, \dots, len-1$$

Computed results are clipped.

The function flavor `ippsConvPartial_NR_16s` performs the same operation as the function `ippsConvPartial_16s32s`, but uses NR rounding and the default scale factor equal to 15.

The function flavor `ippsConvPartial_NR_Low_16s` performs the same operation as the function `ippsConvPartial_NR_16s`, but it requires each value of the sum does not exceed the `Ipp32s` data type range.

Note that the output results produced by the partial convolution function are the same as the first `len` results of the general signal processing convolution function `ippsConv_16s_Sfs`, if the difference due to rounding is not taken into account.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> , <code>pSrc2</code> , or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.
<code>ippStsScaleRangeErr</code>	Indicates an error when <code>scaleFactor</code> is negative.

## InterpolateC\_NR

*Computes the weighted sum of two vectors*

### Syntax

```
IppStatus ippsInterpolateC_NR_16s(const Ipp16s* pSrc1, Ipp16s val1, int
val1ScaleFactor, const Ipp16s* pSrc2, Ipp16s val2, int val2ScaleFactor,
Ipp16s* pDst, int len);
```

### Parameters

<code>pSrc1</code>	Pointer to the first source vector
<code>val1</code>	First factor

<i>val1ScaleFactor</i>	First factor scale
<i>pSrc2</i>	Pointer to the second source vector
<i>val2</i>	Second factor
<i>val2ScaleFactor</i>	Second factor scale
<i>pDst</i>	Pointer to the destination vector
<i>len</i>	Length of the vectors

### Description

The function `ippsInterpolateC_NR` is declared in the `ipps.h` file. This function computes the weighted sum of two vectors as

$$pDst[i] = (2^{\text{val1ScaleFactor}} \cdot \text{val1} \cdot pSrc1[i] + 2^{\text{val2ScaleFactor}} \cdot \text{val2} \cdot pSrc2[i]) \cdot 2^{-16}$$

Here  $i = 0, K, \text{len}-1$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.
<code>ippStsSizeErr</code>	Indicates an error when <i>val1ScaleFactor</i> or <i>val2ScaleFactor</i> is less than zero.

## Mul\_NR

*Multiplies the elements of two vectors.*

---

### Syntax

```

IppStatus ippsMul_NR_16s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
Ipp16s* pDst, int len, int scaleFactor);

IppStatus ippsMul_NR_16s_ISfs (const Ipp16s* pSrc, Ipp16s* pSrcDst, int len,
int scaleFactor);

```

## Parameters

<i>pSrc1, pSrc2</i>	Pointers to the source vectors to be multiplied.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrc</i>	Pointer to the vector to be multiplied by the elements of <i>pSrcDst</i> (for in-place operation).
<i>pSrcDst</i>	Pointer to the source and destination vector (for in-place operation).
<i>len</i>	Number of elements in each vector.
<i>scaleFactor</i>	Scale factor for output data scaling.

## Description

The function `ippsMul_NR` is declared in the `ippsc.h` file. This function multiplies the first source vector *pSrc1* by the second source vector *pSrc2* element-wise, and stores results in *pDst*.

The in-place function flavor multiplies the vector *pSrc* by the vector *pSrcDst* element-wise, and stores results in *pSrcDst*.

Both function flavors scale the multiplication results in accordance with the *scaleFactor* value (see [Integer Scaling](#)). If output values exceed the data range, the results are saturated.

Functions `ippsMul_NR` perform “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , <i>pSrc</i> , <i>pSrcDst</i> , or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is less than 0.

## MulC\_NR

*Multiplies each element of a vector by a constant value.*

---

### Syntax

```
IppStatus ippsMulC_NR_16s_Sfs (Ipp16s* pSrc, Ipp16s val, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippsMulC_NR_16s_ISfs (Ipp16s val, Ipp16s* pSrcDst, int len, int scaleFactor);
```

### Parameters

<i>val</i>	The scalar value used to multiply to each element of the source vector.
<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor for output data scaling.

### Description

The function `ippsMulC_NR` is declared in the `ippsc.h` file. This function multiplies the source vector *pSrc* by the scalar value *val*, and stores the result in *pDst*.

The in-place function flavor multiplies the vector *pSrcDst* by *val*, and stores the result in *pSrcDst*.

Both function flavors scale the multiplication results in accordance with the *scaleFactor* value (see [Integer Scaling](#)). If output values exceed the data range, the results are saturated.

Functions `ippsMulC_NR` perform “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when the `pSrc`, `pDst`, or `pSrcDst` pointer is NULL.

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

`ippStsScaleRangeErr` Indicates an error when `scaleFactor` is less than 0.

## MulPowerC\_NR

Performs power weighting for each element of a vector.

---

### Syntax

```
IppStatus ippMulPowerC_NR_16s_Sfs (const Ipp16s* pSrc, Ipp16s val, Ipp16s*
pDst, int len, int scaleFactor);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>val</code>	The scalar value to be multiplied to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>len</code>	Number of elements in each vector.
<code>scaleFactor</code>	Scale factor for output data scaling.

### Description

The function `ippMulPowerC_NR` is declared in the `ippsc.h` file. This function multiplies each element of the source vector `pSrc` by a power of a scalar value, `val`, and stores the results in `pDst`. The calculation is performed as:

$$pDst[i] = val^i * pSrc[i], 0 \leq i < len.$$

The function scales the multiplication results in accordance with the `scaleFactor` value (see [Integer Scaling](#)). If output values exceed the data range, the results are saturated.

The function `ippMulPowerC_NR` performs “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is NULL.

`ippStsSizeErr` Indicates an error when *len* is less than or equal to 0.  
`ippStsScaleRangeErr` Indicates an error when *scaleFactor* is less than 0.

## AutoScale

*Scales by the maximal elements.*

---

### Syntax

```
IppStatus ippsAutoScale_16s (const Ipp16s* pSrc, Ipp16s* pDst, int len, int* pScale);
```

```
IppStatus ippsAutoScale_16s_I(Ipp16s* pSrcDst, int len, int* pScale);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>len</i>	Number of elements in each vector.
<i>pScale</i>	Pointer to the input/output scaling factor.

### Description

The function `ippsAutoScale` is declared in the `ippsc.h` file. This function scales the input vector as follows:

$$pDst[i] = 2^{scaleFactor} * pSrc[i], i = 0, 1, \dots, len-1,$$

where *scaleFactor* = *normMax* - *pScale*[0], and *normMax* is calculated so that the maximal absolute value of the vector can be normalized. The value *pScale*[0] is the input scaling factor. This scaling factor is replaced by *scaleFactor* after processing.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> , <i>pSrcDst</i> , <i>pScale</i> pointer is NULL
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

`ippStsScaleRangeErr` Indicates an error when the scale factor pointed by `pScale` is less than 0.

## DotProdAutoScale

*Computes the dot product of two vectors using the automatic scaling.*

---

### Syntax

```
IppStatus ippsDotProdAutoScale_16s32s_Sfs(const Ipp16s* pSrc1, const Ipp16s*
pSrc2, int len, Ipp32s* pDp, int* pSfs);
```

### Parameters

<code>pSrc1</code>	Pointer to the first source vector.
<code>pSrc2</code>	Pointer to the second source vector.
<code>len</code>	Number of elements in each vector.
<code>pDp</code>	Pointer to the output result.
<code>pSfs</code>	Pointer to the scaling factor.

### Description

The function `ippsDotProdAutoScale` is declared in the `ippsc.h` file. This function computes the dot product of two vectors and automatically scales it during calculation, adjusting the scale factor so as to eliminate possible overflow in the process:

$$pDp = 2^{\text{scaleFactor}} \cdot \sum_{i=0}^{\text{len}-1} pSrc1[i] \cdot pSrc2[i]$$

The final scaling factor is returned by `pSfs`. Vectors `pSrc1` and `pSrc2` must have the same length, `len`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc1</code> , <code>pSrc2</code> , <code>pDp</code> , or <code>pSfs</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

## InvSqrt

Computes inverse square root of vector elements.

### Syntax

```
IppStatus ippInvSqrt_32s_I (Ipp32s* pSrcDst, int len);
```

### Parameters

<code>pSrcDst</code>	Pointer to the source and destination vector.
<code>len</code>	Number of elements in the vector.

### Description

The function `ippInvSqrt` is declared in the `ippsc.h` file. This function computes the inverse square roots as:

$$pDst[i] = \frac{2^{31}}{\sqrt{pSrc[i]}} , \quad i = 0, ..len-1 ,$$

using the approximation table. The input and output vectors are scaled by 30 bits.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.



## AutoCorr

*Calculates autocorrelation of a vector.*

---

### Syntax

```
IppStatus ippsAutoCorr_16s32s (const Ipp16s* pSrc, int srcLen, Ipp32s* pDst,
int dstLen);
```

### Parameters

<i>pSrc</i>	Pointer to the first source vector [ <i>srcLen</i> ].
<i>srcLen</i>	Length of the source vector.
<i>pDst</i>	Pointer to the destination vector [ <i>dstLen</i> ].
<i>dstLen</i>	Length of the destination vector (the number of autocorrelation values to calculate).

### Description

The function `ippsAutoCorr` is declared in the `ippsc.h` file. This function calculates autocorrelation of the input vector as follows:

$$pDst[n] = \sum_{i=n}^{srcLen-1} pSrc[i-n] \cdot pSrc[i]$$

Here  $n = 0, \dots, dstLen - 1$ .

and stores the result in *pDst*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>srcLen</i> or <i>dstLen</i> is less or equal to 0.

## AutoCorrLagMax

*Estimates the maximum auto-correlation of a vector.*

---

### Syntax

```
IppStatus ippsAutoCorrLagMax_Inv_16s (const Ipp16s* pSrc, int len, int
lowerLag, int upperLag, Ipp32s* pMax, int* pMaxLag);

IppStatus ippsAutoCorrLagMax_Fwd_16s (const Ipp16s* pSrc, int len, int
lowerLag, int upperLag, Ipp32s* pMax, int* pMaxLag);

IppStatus ippsAutoCorrLagMax_32f (const Ipp32f* pSrc, int len, int lowerLag,
int upperLag, Ipp32f* pMax, int* pMaxLag);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector [-upperLag, len].
<i>len</i>	Length of autocorrelation.
<i>lowerLag</i>	Lower input lag value.
<i>upperLag</i>	Upper input lag value.
<i>pMax</i>	Pointer to the output maximum of the correlation.
<i>pMaxLag</i>	Pointer to the output lag which holds the maximum of the correlation.

### Description

These functions are declared in the `ippsc.h` file. The functions `ippsAutoCorrLagMax_Inv` and `ippsAutoCorrLagMax_32f` find the maximum autocorrelation within the given lag range as:

$$pMax = \max_n \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i-n]$$

Here  $lowerLag \leq n \leq upperLag$ .

The function `ippsAutoCorrLagMax_Fwd` finds the maximum autocorrelation within the given lag range as given by:

$$pMax = \max_n \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n]$$

Here  $lowerLag \leq n \leq upperLag$ .

If several maximums exist, functions return the first of them.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> , <code>pMax</code> , or <code>pMaxLag</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## AutoCorr\_NormE

Estimates normal auto-correlation of a vector.

### Syntax

```
IppStatus ippsAutoCorr_NormE_16s32s (const Ipp16s* pSrc, int len, Ipp32s* pDst, int lenDst, int* pNorm);
```

```
IppStatus ippsAutoCorr_NormE_NR_16s (const Ipp16s* pSrc, int len, Ipp16s* pDst, int lenDst, int* pNorm);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector in Q12.
<code>len</code>	Number of elements in the source vector.
<code>pDst</code>	Pointer to the destination vector that stores the estimated auto-correlation results of the source vector.
<code>lenDst</code>	The number of elements in the destination vector.
<code>pNorm</code>	Pointer to the output scale factor.

## Description

The function `ippAutoCorr_NormE` is declared in the `ippsc.h` file. This function calculates the autocorrelation of the source vector `pSrc`. The results are scaled according to the first autocorrelation coefficient (energy) value. Specifically, autocorrelation coefficients are multiplied by the factor  $2^{norm}$ , where  $norm \geq 0$  is calculated so as to make the first coefficient normalized:

$$pDst[n] = 2^{norm} \cdot \sum_{i=0}^{len-1} pSrc[i] \cdot pSrc[i+n], \quad 0 \leq n < lenDst, 0 \leq norm$$

where

$$pSrc[i] = \begin{cases} pSrc[i] & 0 \leq i < len \\ 0 & otherwise \end{cases}$$

The corresponding scaling factor,  $norm$ , is returned by `pNorm`.

The function that has the `NR` suffix performs the “nearest right” rounding (see [Rounding Mode](#)).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> or <code>pNorm</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> or <code>lenDst</code> is less than or equal to zero.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

## CrossCorr

Estimates the cross-correlation of two vectors.

### Syntax

```
IppStatus ippsCrossCorr_16s32s_Sfs (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
int len, Ipp32s* pDst, int scaleFactor);
```

```
IppStatus ippsCrossCorr_NormM_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
int len, Ipp16s* pDst);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>len</i>	Number of elements in the source and destination vectors.
<i>pDst</i>	Pointer to the destination vector.
<i>scaleFactor</i>	Scale factor of the destination vector.

### Description

The function `ippsCrossCorr` is declared in the `ippsc.h` file. This function estimates the cross-correlation between the vector *pSrc1* and the vector *pSrc2* as given by:

$$\text{corr}[n] = 2^{\text{scaleFactor}} \cdot \sum_{i=0}^{\text{len}-1} p\text{Src1}[i] \cdot p\text{Src2}[i+n] \quad ,$$

where

$$p\text{Src2}[i] = \begin{cases} p\text{Src2}[i] & , 0 \leq i < \text{len} \\ 0 & , \text{otherwise} \end{cases}$$

Results are stored in the vector *pDst*. Correlation sums are saturated. Scaling is performed according to the *scaleFactor* value.

The function that has the `NormMs` suffix uses the scale factor that normalizes the absolute maximum of the correlation sums.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , or <i>pDst</i> pointer is NULL.
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is negative.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

## CrossCorrLagMax

*Estimates the maximum cross-correlation between two vectors.*

---

### Syntax

```
IppStatus ippsCrossCorrLagMax_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
int len, int lag, Ipp32s* pMax, int* pMaxLag);
```

```
IppStatus ippsCrossCorrLagMax_32f64f (const Ipp32f* pSrc1, const Ipp32f*
pSrc2, int len, int lag, Ipp64f* pMax, int* pMaxLag);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector [ <i>len</i> ].
<i>pSrc2</i>	Pointer to the second source vector [ <i>len+lag+1</i> ].
<i>len</i>	Length of the cross-correlation.
<i>lag</i>	The maximal lag value.
<i>pMax</i>	Pointer to the maximum cross-correlation.
<i>pMaxLag</i>	Pointer to the lag which holds the cross-correlation maximum.

### Description

The functions `ippsCrossCorrLagMax` are declared in the `ippsc.h` file. These functions find the maximum of the cross-correlation between two input vectors by the formula:

$$pMax = \max_{0 \leq n \leq lag} \sum_{i=0}^{len-1} pSrc1[i] \cdot pSrc2[i+n]$$

The functions return the value of the cross-correlation maximum and the lag which holds it. The function `ippsCrossCorrLagMax_16s` returns *pMax* value multiplied by two.

If more than one lag which holds the cross-correlation maximum exist, the function returns the biggest lag.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , <i>pMax</i> , or <i>maxLag</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>lag</i> is less than 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less or equal to 0.

## SynthesisFilter

*Computes the speech signal by filtering the input speech through the synthesis filter 1/A(z).*

### Syntax

```
IppStatus ippsSynthesisFilter_NR_16s_Sfs(const Ipp16s* pLPC, const Ipp16s*
pSrc, Ipp16s* pDst, int len, int scaleFactor, const Ipp16s* pMem);
```

```
IppStatus ippsSynthesisFilterLow_NR_16s_ISfs(const Ipp16s* pLPC, Ipp16s*
pSrcDst, int len, int scaleFactor, const Ipp16s* pMem);
```

**THE FOLLOWING FUNCTION IS DEPRECATED.**

```
IppStatus ippsSynthesisFilter_NR_16s_ISfs(const Ipp16s* pLPC, Ipp16s* pSrcDst,
int len, int scaleFactor, const Ipp16s* pMem);
```

### Parameters

<i>pLPC</i>	Pointer to the input filter coefficients $a_0, a_1, \dots, a_{10}$ .
<i>pSrc</i>	Pointer to the input vector.

<i>pSrcDst</i>	Pointer to the history input/filtered output vector.
<i>pDst</i>	Pointer to the filtered output.
<i>len</i>	Number of elements in the vector.
<i>scaleFactor</i>	Scale factor for the destination vector.
<i>pMem</i>	Pointer to the memory supplied for filtering. Should be updated outside the function with last 10 values of the destination vector.

## Description




---

**CAUTION.** THE FUNCTION `ippsSynthesisFilter_NR_16s_ISfs` IS DEPRECATED. Please use the function `ippsSynthesisFilter_NR_16s_Sfs` instead.

---

The function `ippsSynthesisFilter` is declared in the `ippsc.h` file. This function filters the input signal through the synthesis filter as follows:

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) \quad , \quad n = 0, 1, \dots, len-1 \quad .$$

The function flavor `ippsSynthesisFilterLow_NR_16s_ISfs` performs no saturation and assumes that input data are small enough to ensure correct calculation without checking the overflow condition. This mode of operation is indicated by the suffix `Low` in the function name.

All flavors perform “nearest right” rounding needed to meet the bit-to-bit exactness requirement.

## Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pLPC</i> , <i>pSrcDst</i> , <i>pDst</i> , or <i>pMem</i> pointer is <code>NULL</code> .
<code>ippsStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippsStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is not equal to 12 or 13.
<code>ippsStsOverflow</code>	Indicates a warning that at least one result value was saturated.



## G.729 Functions

The Intel IPP functions described in this section implement building blocks that can be used to create speech codecs compliant with the ITU-T Recommendation G.729 (see [ITU729], [ITU729A], [ITU729B]).

These functions are used in the Intel® IPP *G.729 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

The list of these functions is given in Table 9-2.

**Table 9-2. Intel IPP G.729 Functions**

Function Base Name	Operation
Basic Functions	
<a href="#">DotProd_G729</a>	Computes the dot product of two vectors.
<a href="#">Interpolate_G729</a>	Computes the weighted sum of two vectors.
Linear Prediction Analysis Functions	
<a href="#">AutoCorr_G729</a>	Estimates the auto-correlation of a vector.
<a href="#">LevinsonDurbin_G729</a>	Calculates LP coefficients from the autocorrelation coefficients.
<a href="#">LPCToLSP_G729</a>	Converts LP coefficients to LSP coefficients.
<a href="#">LSFToLSP_G729</a>	Converts Line Spectral Frequencies to LSP coefficients.
<a href="#">LSFQuant_G729</a>	Performs quantization of LSF coefficients.
<a href="#">LSFDecode_G729</a>	Decodes quantized LSFs.
<a href="#">LSFDecodeErased_G729</a>	Reconstructs quantized LSFs in case when a frame is erased.
<a href="#">LSPToLPC_G729</a>	Converts LSP coefficients to LP coefficients.
<a href="#">LSPQuant_G729</a>	Quantizes the LSP coefficients.
<a href="#">LSPToLSF_G729</a>	Converts LSP coefficients to LSF coefficients.
<a href="#">LagWindow_G729</a>	Applies 60Hz bandwidth expansion.
Codebook Search Functions	
<a href="#">OpenLoopPitchSearch_G729</a>	Searches for an optimal pitch value.
<a href="#">AdaptiveCodebookSearch_G729</a>	Searches for the integer delay and the fraction delay, and computes the adaptive vector.
<a href="#">DecodeAdaptiveVector_G729</a>	Restores the adaptive codebook vector by interpolating the past excitation.
<a href="#">FixedCodebookSearch_G729</a>	Searches for the fixed codebook vector.
<a href="#">GainCodebookSearch_G729</a>	Finds pitch and fixed gains.
<a href="#">ToeplitzMatrix_G729</a>	Calculates 616 elements of the Toeplitz matrix for the fixed codebook search.

Function Base Name	Operation
<b>Codebook Gain Functions</b>	
<a href="#">DecodeGain_G729</a>	Decodes the adaptive and fixed-codebook gains.
<a href="#">GainControl_G729</a>	Calculates adaptive gain control.
<a href="#">GainQuant_G729</a>	Quantizes the codebook gain using a two-stage conjugate-structured codebook.
<a href="#">AdaptiveCodebookContribution_G729</a>	Updates target vector for codebook search by subtracting adaptive codebook contribution.
<a href="#">AdaptiveCodebookGain_G729</a>	Calculates the gain of the adaptive-codeblock vector and the filtered codebook vector.
<b>Filter Functions</b>	
<a href="#">ResidualFilter_G729</a>	Implements an inverse LP filter and obtains the residual signal.
<a href="#">SynthesisFilter_G729</a>	Reconstructs the speech signal from LP coefficients and residuals.
<a href="#">LongTermPostFilter_G729</a>	Restores the long-term information from the old speech signal.
<a href="#">ShortTermPostFilter_G729</a>	Restores speech signal from the residuals.
<a href="#">TiltCompensation_G729</a>	Compensates for the tilt in the short-term filter.
<a href="#">HarmonicFilter</a>	Calculates the harmonic filter.
<a href="#">HighPassFilterSize_G729</a>	calculates the G729 high-pass filter size.
<a href="#">HighPassFilterInit_G729</a>	Initializes the high-pass filter.
<a href="#">HighPassFilter_G729</a>	Performs G729 high-pass filtering.
<a href="#">IIR16s_G729</a>	Performs IIR filtering.
<a href="#">PhaseDispersionGetStateSize_G729D</a>	Queries the memory length of the phase dispersion filter.
<a href="#">PhaseDispersionInit_G729D</a>	Initializes the phase dispersion filter memory.
<a href="#">PhaseDispersionUpdate_G729D</a>	Updates the phase dispersion filter state.
<a href="#">PhaseDispersion_G729D</a>	Performs the phase dispersion filtering.
<a href="#">Preemphasize_G729A</a>	Computes pre-emphasis of a post filter.
<a href="#">WinHybridGetStateSize_G729E</a>	Queries the length of the hybrid windowing module memory.
<a href="#">WinHybridInit_G729E</a>	Initializes the hybrid windowing module memory.
<a href="#">WinHybrid_G729E</a>	Applies the hybrid window and computes autocorrelation coefficients.
<a href="#">RandomNoiseExcitation_G729B</a>	Initializes a random vector with a Gaussian distribution.
<a href="#">FilteredExcitation_G729</a>	Computes filtered excitation.

## Basic Functions

These functions perform basic operation for codec.

### DotProd\_G729

*Computes the dot product of two vectors.*

---

#### Syntax

```
IppStatus ippsDotProd_G729A_16s32s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
int len, Ipp32s* pDp);

IppStatus ippsDotProd_G729A_32f (const Ipp32f* pSrc1, const Ipp32f* pSrc2,
int len, Ipp32f* pDp);
```

#### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>len</i>	Number of elements in each vector.
<i>pDp</i>	Pointer to the output result.

#### Description

The functions `ippsDotProd_G729A` are declared in the `ippsc.h` file. The function `ippsDotProd_G729A_16s32s` computes the dot product of two source vectors *pSrc1* and *pSrc2* as:

$$pDp = 2 \cdot \sum_{i=0}^{len/2} pSrc1[2 \cdot i] \cdot pSrc2[2 \cdot i]$$

while for the `ippsDotProd_G729A_32f` function the result is

$$pDp = \sum_{i=0}^{len/2} pSrc1[2 \cdot i] \cdot pSrc2[2 \cdot i]$$

Vectors *pSrc1* and *pSrc2* must have the same length, *len*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc1</i> , <i>pSrc2</i> , or <i>pDp</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

## Interpolate\_G729

*Computes the weighted sum of two vectors.*

---

### Syntax

```
IppStatus ippInterpolate_G729_16s (const Ipp16s* pSrc1, const Ipp16s* pSrc2,
Ipp16s* pDst, int len);
```

```
IppStatus ippInterpolateC_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s val1,
const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippInterpolateC_NR_G729_16s_Sfs (const Ipp16s* pSrc1, Ipp16s val1,
const Ipp16s* pSrc2, Ipp16s val2, Ipp16s* pDst, int len, int scaleFactor);
```

```
IppStatus ippInterpolateC_G729_32f (const Ipp32f* pSrc1, Ipp32f val1, const
Ipp32f* pSrc2, Ipp32f val2, Ipp32f* pDst, int len);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>val1</i>	Factor to multiply to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>val2</i>	Factor to multiply to the second source vector.
<i>pDst</i>	Pointer to the destination (interpolated) vector.
<i>len</i>	Number of elements in the vectors.

*scaleFactor*

Scale factor for the destination vector.

### Description

These functions are declared in the `ippsc.h` file. The function `ippsInterpolate_G729_16s` computes the weighted sum as:

```
pDst[i] = (pSrc1[i] + sign(pSrc1[i])) >> 1 + (pSrc2[i] + sign(pSrc2[i]))>>1,
for i = 0, 1,... len - 1.
```

Functions `ippsInterpolateC_G729_16s_Sfs` and `ippsInterpolateC_NR_G729_16s_Sfs` both use the same formula to compute the weighted sum as:

```
pDst[i] = (val1 * pSrc1[i] + val2 * pSrc2[i]) >> scaleFactor for i = 0, 1,...
len - 1.
```

However, these functions differ in the rounding mode they use for output results (see [Rounding Mode](#)).

The function `ippsInterpolateC_G729_32f` computes the weighted sum as:

```
pDst[i] = (val1 * pSrc1[i] + val2 * pSrc2[i]) for i = 0, 1,... len.
```

### Return Values

`ippStsNoErr`

Indicates no error.

`ippStsNullPtrErr`Indicates an error when the *pSrc1*, *pSrc2*, or *pDst* pointer is NULL.`ippStsSizeErr`Indicates an error when *len* is less than or equal to 0.`ippStsScaleRangeErr`Indicates an error when *scaleFactor* is less than 0.

## Linear Prediction Analysis Functions

Functions described in this section implement LSP coding (quantization) and decoding, as well as transformation between LPC, LSP and LSF coefficients.

## AutoCorr\_G729

*Estimates the auto-correlation of a vector.*

---

### Syntax

```
IppStatus ippsAutoCorr_G729B(const Ipp16s* pSrcSpch, Ipp16s*
pResultAutoCorrExp, Ipp32s* pDstAutoCorr);
```

## Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal vector [240].
<i>pResultAutoCorrExp</i>	Pointer to the exponent of autocorrelation coefficients.
<i>pDstAutoCorr</i>	Pointer to the autocorrelation coefficients vector [13].

## Description

The function `ippsAutoCorr_G729` is declared in the `ippsc.h` file. This function calculates a set of 11 autocorrelation coefficients and their exponent for the input speech signal. The function is applied to the vector of 240 speech samples, which include 120 samples from past speech frames, 80 samples from the present speech frame, and 40 samples from the future frame. The functionality is as follows.

1. First, apply to speech samples the asymmetric windows given by:

$$w_{1p}(n) = \begin{cases} 0.54 - 0.46 \cos\left(\frac{2n\pi}{399}\right), & n = 0, 1, \dots, 199 \\ \cos\left(\frac{2(n-200)\pi}{159}\right), & n = 200, 201, \dots, 239 \end{cases}$$

2. Next, calculate autocorrelations of the windowed speech samples  $s(i)$ ,  $i = 0, 1, \dots, 239$ , using the formula:

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k), \quad k = 0, 1, \dots, 11$$

3. Finally, the autocorrelation coefficients are scaled according to the first autocorrelation coefficient (energy) value. Specifically, autocorrelation coefficients are multiplied by the factor  $2^{norm}$ , where  $norm \leq 0$  is calculated so as to make the first coefficient normalized. The corresponding scaling factor,  $norm$ , is returned via `pResultAutoCorrExp`.

The function `ippsAutoCorr_G729B` is actually a combination of `ippsMul_NR` and `ippsAutoCorr_NormE` functions. The following code details the correspondence:

```
{
short sig_win[240];
ippsMul_NR_16s_Sfs(pSrcSpch,window,sig_win,240,15);
ippsAutoCorr_NormE_16s32s(sig_win,240,pDstAutoCorr,11,
&pResultAutoCorrExp);
}
```

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LevinsonDurbin\_G729

*Calculates LP coefficients from the autocorrelation coefficients.*

---

### Syntax

```
IppStatus ippsLevinsonDurbin_G729_32s16s(const Ipp32s* pSrcAutoCorr, int
order, Ipp16s* pDstLPC, Ipp16s* pDstRc, Ipp16s* pResultResidualEnergy);

IppStatus ippsLevinsonDurbin_G729_32f(const Ipp32f* pSrcAutoCorr, int order,
Ipp32f* pDstLpc, Ipp32f* pDstRc, Ipp32f* pResultResidualEnergy);

IppStatus ippsLevinsonDurbin_G729B(const Ipp32s* pSrcAutoCorr, Ipp16s*
pDstLPC, Ipp16s* pDstRC, Ipp16s* pResultResidualEnergy);
```

### Parameters

<i>order</i>	The LP order.
<i>pSrcAutoCorr</i>	Pointer to the autocorrelation coefficients vector [ <i>order</i> + 1].
<i>pDstLPC</i>	Pointer to the output LP coefficients [ <i>order</i> + 1].
<i>pDstRC</i>	Pointer to the output reflection coefficients vector [ <i>order</i> ].
<i>pResultResidualEnergy</i>	Pointer to the residual energy.

## Description

The functions `ippsLevinsonDurbin_G729` and `ippsLevinsonDurbin_G729B` are declared in the `ippsc.h` file. These functions calculate Linear Prediction (LP) coefficients of the LP filter with the given order from the autocorrelation coefficients, using Levinson-Durbin algorithm. The function `ippsLevinsonDurbin_G729` uses the parameter `order`, while the function `ippsLevinsonDurbin_G729B` operates for the default `order = 10`.

To obtain LP coefficients  $a_i$ ,  $i = 1, 2, \dots, \text{order}$ , the following set of equations is to be solved:

$$\sum_{i=1}^{\text{order}} a_i \times r(|i-k|) = -r(k) \text{ , } k = 1, 2, \dots, \text{order} .$$

,  $k = 1, 2, \dots, \text{order}$  .

The functions perform the following steps:

1. Levinson-Durbin algorithm is applied to solve the above set of equations. This algorithm uses the following recursion.

$$\begin{aligned} E^{[0]} &= r(0) \\ \text{for } i &= 1 \text{ to } \text{order} \\ a_0^{[i-1]} &= 1 \\ k_i &= - \left[ \sum_{j=0}^{i-1} a_j^{[i-1]} \times r(i-j) \right] / E^{[i-1]} \\ a_i^{[i]} &= k_i \\ \text{for } j &= 1 \text{ to } i-1 \end{aligned}$$



```


$$a_j^{[i]} = a_j^{[i-1]} + k_i \times a_{i-j}^{[i-1]}$$

end

$$E^{[i]} = E^{[i-1]} - k_i^2 E^{[i-1]}$$

end

```

2. Set  $E$  as the output residual energy and  $k_i$  as reflection coefficients.

3. If the LPC filter used in this algorithm is unstable, that is some  $|k_i|$  is very close to 1.0 during recursion, the elements of input vectors *pSrcDstLPC* and *pSrcDstRC* RC coefficients are not changed and the residual energy is set to 0.

### Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when the <i>pSrcAutoCorr</i> , <i>pDstRC</i> , <i>pResultResidualEnergy</i> , or <i>pDstLPC</i> pointer is <i>NULL</i> .
<i>ippStsSizeErr</i>	Indicates an error when <i>order</i> is less than or equal to 0, or when <i>order</i> is greater than 30.

## LPCToLSP\_G729

Converts LP coefficients to LSP coefficients.

### Syntax

```

IppStatus ippsLPCToLSP_G729_16s(const Ipp16s* pSrcLPC, const Ipp16s*
pSrcPrevLSP, Ipp16s* pDstLSP);

IppStatus ippsLPCToLSP_G729A_16s(const Ipp16s* pSrcLPC, const Ipp16s*
pSrcPrevLSP, Ipp16s* pDstLSP);

IppStatus ippsLPCToLSP_G729A_32f(const Ipp32f* pSrcLPC, const Ipp32f*
pSrcPrevLSP, Ipp32f* pDstLSP);

IppStatus ippsLPCToLSP_G729_32f(const Ipp32f* pSrcLPC, const Ipp32f*
pSrcPrevLSP, Ipp32f* pDstLSP);

```

## Parameters

<i>pSrcLPC</i>	Pointer to the LP coefficients vector [11], in Q12 for 16s data.
<i>pSrcPrevLSP</i>	Pointer to the previous LSP coefficients vector [10], in Q15 for 16s data.
<i>pDstLSP</i>	Pointer to the computed LSP coefficients vector [10].

## Description

These functions are declared in the `ippsc.h` file. They convert 10th order LP coefficients to LSP coefficients. The first function `ippsLPCToLSP_G729` is designed for G.729/B codec while the second function `ippsLPCToLSP_G729A` is designed for G.729A codec. Functions perform the following steps:

1. Calculate the polynomial coefficients of  $F_1(z)$  and  $Fz_2(z)$ , using the following recursive relations:

$$\begin{aligned} f_1(i+1) &= a_{i+1} + a_{10-i} - f_1(i) \\ f_2(i+1) &= a_{i+1} - a_{10-i} + f_2(i), \quad i = 0, 1, \dots, 4, \end{aligned}$$

where  $f_1(0) = f_2(0) = 1.0$ .

2. Use Chebyshev polynomials to evaluate  $F_1(z)$  and  $Fz_2(z)$ . The Chebyshev polynomials are given by.

$$\begin{aligned} C_1(\omega) &= \cos(5\omega) + f_1(1)\cos(4\omega) + f_1(2)\cos(3\omega) + f_1(3)\cos(2\omega) + f_1(4)\cos(\omega) + f_1(5)/2 \\ C_2(\omega) &= \cos(5\omega) + f_2(1)\cos(4\omega) + f_2(2)\cos(3\omega) + f_2(3)\cos(2\omega) + f_2(4)\cos(\omega) + f_2(5)/2 \end{aligned}$$

For G.729/B, evaluate  $F_1(z)$  and  $Fz_2(z)$  at 60 points equally spaced between 0 and  $\pi$  and check for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided four times to track the root. For G.729A, evaluate  $F_1(z)$  and  $Fz_2(z)$  at 50 points equally spaced between 0 and  $\pi$  and check for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided two times to track the root.

3. If all 10 roots needed to determine LSP coefficients are not found, just use the previous set of LSP coefficients.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LSFToLSP\_G729

*Converts Line Spectral Frequencies to LSP coefficients.*

---

### Syntax

```
IppStatus ippLSFToLSP_G729_16s (const Ipp16s* pLSF, Ipp16s* pLSP);
```

### Parameters

<i>pLSF</i>	Pointer to the LSF vector [10], in Q13 in the range $[0, \pi]$ .
<i>pLSP</i>	Pointer to the LSP vector [10], in Q15 in the range $[-1; 1]$ .

### Description

The function `ippLSFToLSP_G729` is declared in the `ippsc.h` file. This function converts the Line Spectral Frequencies (LSF) to the LSP coefficients as given by:

$$pLSP[i] = \cos(pLSF[i]), i = 1, \dots, 10.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

## LSFQuant\_G729

*Performs quantization of LSF coefficients.*

---

### Syntax

```
IppStatus ippsLSFQuant_G729_16s (const Ipp16s* pLSF, Ipp16s* pQuantLSFTable,
Ipp16s* pQuantLSF, Ipp16s* quantIndex);
```

```
IppStatus ippsLSFQuant_G729B_16s (const Ipp16s* pLSF, Ipp16s* pQuantLSFTable,
Ipp16s* pQuantLSF, Ipp16s* quantIndex);
```

```
IppStatus ippsLSFQuant_G729B_32f (const Ipp32f* pLSF, Ipp32f* pQuantLSFTable,
Ipp32f* pQuantLSF, int* quantIndex);
```

### Parameters

<i>pLSF</i>	Pointer to the LSF vector [10], in Q13 in the range [0,π] for 16s data.
<i>pQuantLSFTable</i>	Pointer to a matrix of 4 rows and 10 columns, containing previously quantized LSFs.
<i>pQuantLSF</i>	Pointer to the quantized LSF vector, in Q13 in the range [0,π] for 16s data.
<i>quantIndex</i>	Pointer to the output combined codebook indices <i>L0</i> , <i>L1</i> , <i>L2</i> , and <i>L3</i> .

### Description

These functions are declared in the `ippsc.h` file. The function `ippsLSFQuant_G729_16s` quantizes the difference between the input LSF coefficients and previously quantized LSF coefficients, using the switched Moving Average (MA) predictor. Quantization is performed using a two-stage Vector Quantizer (VQ): 10-dimensional VQ using codebook *L1* (128 entries), and 10-bit split VQ (5 dimensions each) on two codebooks, *L2* and *L3* (32 entries each).

The function `ippsLSFQuant_G729B_16s` quantizes the LSF coefficients using a two-stage split VQ, (in 5 and 4 bits). The second order MA predictor used in the SID-LPC quantization procedure is calculated as a linear interpolation of the first and second MA predictors, with weight values 0.6 and 0.4, respectively. The first stage of quantization is the same as that in G.729, but only part of the quantization table (32 entries) is used. The second stage is different. No splitting is done and also only portion of the second table (16 entries) is used.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLSFLow</code>	Indicates a warning that the corresponding filter has low stability.
<code>ippStsLSFHigh</code>	Indicates a warning that the corresponding filter has high stability.
<code>ippStsLSFLowAndHigh</code>	Indicates a warning that the corresponding filter has both low and high stability.

## LSFDecode\_G729

*Decodes quantized LSFs.*

---

### Syntax

```

IppStatus ippsLSFDecode_G729_16s (const Ipp16s* quantIndex, Ipp16s*
pQuantLSFTable, Ipp16s* pQuantLSF);

IppStatus ippsLSFDecode_G729B_16s (const Ipp16s* quantIndex, Ipp16s*
pQuantLSFTable, Ipp16s* pQuantLSF);

IppStatus ippsLSFDecode_G729B_32f (const int* quantIndex, Ipp32f*
pQuantLSFTable, Ipp32f* pDstQLsp);

IppStatus ippsLSFDecode_G729_32f (const int* quantIndex, Ipp32f*
pQuantLSFTable, Ipp32f* pQuantLSF);

```

### Parameters

<code>quantIndex</code>	Pointer to the vector of indices: L0, L1, L2, L3 (see Table 8/G.729 in [ITU729] ) or L0, L1, L2 (see also [ITU729B] , section 4.3) .
<code>pQuantLSFTable</code>	Pointer to the input/output table [4][10] of 4 previously quantized LSFs vectors, in Q13 for 16s data.
<code>pQuantLSF</code>	Pointer to the quantized LSF output vector [10], in Q13 for 16s data.

## Description

These functions are declared in the `ippsc.h` file. The function `ippsLSFDecode_G729` retrieves the quantized LSF coefficients from the quantization table and indices.

The function `ippsLSFDecode_G729B` retrieves the quantized LSF coefficients for the SID frame. Only *L0*, *L1*, *L2* indices are used.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLSFLow</code>	Indicates a warning of only low stability of LSF.
<code>ippStsLSFHigh</code>	Indicates a warning of only high stability of LSF.
<code>ippStsLSFLowAndHigh</code>	Indicates a warning of both low and high stability of LSF.

## LSFDecodeErased\_G729

*Reconstructs quantized LSFs in case when a frame is erased.*

---

### Syntax

```
IppStatus ippsLSFDecodeErased_G729_16s (Ipp16s maIndex, Ipp16s*
pQuantLSFTable, Ipp16s* pQuantLSF);

IppStatus ippsLSFDecodeErased_G729_32f (int maIndex, Ipp32f* pQuantLSFTable,
const Ipp32f* pSrcPrevLSF);
```

### Parameters

<i>maIndex</i>	Switched MA predictor ( <i>L0</i> ) .
<i>pQuantLSFTable</i>	Pointer to the input/output table [4][10] of 4 previously quantized LSFs vectors, in Q12 for 16s data.
<i>pQuantLSF</i>	Pointer to the quantized LSF output vector [10], in Q12 for 16s data.

## Description

The function `ippsLSFDecodeErased_G729` is declared in the `ippsc.h` file. This function retrieves the quantized LSF coefficients through the previously decoded switched MA-predictor index ( $L0$ ).

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.  
`ippStsOutOfRangeErr` Indicates an error when  $maIndex < 0$ , or  $maIndex \geq MA\_NP$ .

## LSPToLPC\_G729

Converts LSP coefficients to LP coefficients.

## Syntax

```
IppStatus ippsLSPToLPC_G729_16s(const Ipp16s* pSrcLSP, Ipp16s* pDstLPC);
IppStatus ippsLSPToLPC_G729_32f(const Ipp32f* pSrcLsp, Ipp32f* pDstLpc);
```

## Parameters

*pSrcLSP* Pointer to the LSP coefficients vector [10], in Q15 for 16s data.  
*pDstLPC* Pointer to the LP coefficients vector [11], in Q12 for 16s data.

## Description

The function `ippsLSPToLPC_G729` is declared in the `ippsc.h` file. This function converts a set of 10th order LSP coefficients to LP coefficients. It performs the following steps:

1. Calculates the polynomial coefficients of  $F_1(z)$  and  $F_2(z)$ , using the recursive relations:

for  $i = 1$  to 5

$$f_1(i) = -2q_{2i-1} * f_1(i-1) + 2f_1(i-2)$$

for  $j = i-1$  down to 1

$$f_1^{[i]}(j) = f_1^{[i-1]}(j) - 2q_{2i-1} * f_1^{[i-1]}(j-1) + f_1^{[i-1]}(j-2)$$

end

end

Here the initial values are set to  $f_1(0) = 1$  and  $f_1(-1) = 0$ . The coefficients  $f_2(i)$  are computed similarly by replacing  $q_{2i-1}$  by  $q_{2i}$ .

2.  $F_1(z)$  and  $F_2(z)$  are then multiplied by  $1+z^{-1}$  and  $1-z^{-1}$  respectively to obtain  $F'_1(z)$  and  $F'_2(z)$  as given by:

$$f'_1(i) = f_1(i) + f_1(i-1), i = 1, 2, \dots, 5$$

$$f'_2(i) = f_2(i) + f_2(i-1), i = 1, 2, \dots, 5$$

3. Finally, the function computes the LP coefficients from  $f'_1(i)$  and  $f'_2(i)$  as:

$$a_i = \begin{cases} 0.5 \times f'_1(i) + 0.5 \times f'_2(i), & i = 1, 2, \dots, 5 \\ 0.5 \times f'_1(11-i) - 0.5 \times f'_2(11-i), & i = 6, 7, \dots, 10 \end{cases}$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcLSP</i> or <i>pDstLPC</i> pointer is NULL.

## LSPQuant\_G729

*Quantizes the LSP coefficients.*

---

### Syntax

```

IppStatus ippsLSPQuant_G729_16s(const Ipp16s* pSrcLSP, Ipp16s*
pSrcDstPrevFreq, Ipp16s* pDstQLSP, Ipp16s* pDstQLSPIndex);

IppStatus ippsLSPQuant_G729E_32f(const Ipp32f* pSrcLSP, Ipp32f*
pSrcDstPrevFreq, Ipp32f* pDstQLSP, Ipp32f* pDstQLSPIndex);

```

### Parameters

<i>pSrcLSP</i>	Pointer to the LSP coefficients vector [10], in Q15 for 16s data.
----------------	---



<i>pSrcDstPrevFreq</i>	Pointer to the four previous and updated quantized frequencies vector [40], in Q13 for 16s data. Elements 0 to 9 are for the newest frame and elements 30 to 39 are for the oldest frame.
<i>pDstQLSF</i>	Pointer to the LSF coefficients vector [10].
<i>pDstQLSP</i>	Pointer to the LSP coefficients vector [10], in Q15 for 16s data.
<i>pDstQLSPIndex</i>	Pointer to the combined codebook indices $L0$ , $L1$ , $L2$ and $L3$ , of length 2. Its first element represents indices $L0$ and $L1$ in the following combination: $L1$ occupies the first 7 bits starting from LSB, whereas $L0$ resides in the next 1 bit adjacent to $L1$ . The second element represents indices $L2$ and $L3$ in the following combination: $L3$ occupies the first 5 bits starting from LSB, and $L2$ resides in the next 5 bits adjacent to $L3$ .

## Description

The functions `ippsLSPQuant_G729` are declared in the `ippsc.h` file. These functions obtain the quantized LSP coefficients and the codebook indices. They perform the following operations:

1. Converts the LSP coefficients  $g_i$  to LSF coefficients  $\alpha_i$  in the normalized frequency domain  $[0, \pi]$  as:

$$\alpha_i = \arccos(g_i), \quad i = 1, \dots, 10$$

2. Uses a switched 4th order MA prediction to predict the LSF coefficients of the current frame. The difference between the computed and predicted coefficients is quantized using a two-stage vector quantizer. The first stage is a 10-dimensional VQ using codebook  $L1$  with 128 entries (7 bits). The second stage is a 10 bit VQ which was implemented as a split VQ using two 5-dimensional codebooks,  $L2$  and  $L3$ , containing 32 entries (5 bits) each. To explain the quantization process, it is convenient to first describe the decoding process. Each coefficient is obtained from the sum of two codebooks:

$$\hat{l} = \begin{cases} L1_i(L1) + L2_i(L2) , & i = 1, \dots, 5 \\ L1_i(L1) + L3_{i-5}(L3) , & i = 6, \dots, 10 \end{cases}$$

where  $L1$ ,  $L2$  and  $L3$  are the codebook indices.

3. The vector to be quantized for the current frame  $m$  is obtained from:

$$l_i = \left[ \omega_i^{(m)} - \sum_{k=1}^4 \hat{p}_{i,k} \hat{l}_i^{(m-k)} \right] / \left( 1 - \sum_{k=1}^4 \hat{p}_{i,k} \right), \quad i = 1, \dots, 10,$$

for  $i = 1, \dots, 10$ ,

where  $p_{i,k}$  are the coefficients of the switched MA predictor. The first codebook **L1** is searched and the entry  $L1$  that minimizes the (unweighted) mean-squared error is selected. This is followed by a search of the second codebook **L2**, which defines the lower part of the second stage. To avoid sharp resonances in the quantized LP synthesis filter, the partial vector  $\hat{l}_i$ ,  $i = 1, \dots, 5$ , is rearranged such that adjacent coefficients have a minimum distance of  $J$ . The rearrangement procedure is shown below:

for  $i = 2, \dots, 10$

```

    if  $\hat{l}_{i-1} > \hat{l}_i - J$ 
         $\hat{l}_{i-1} = (\hat{l}_i + \hat{l}_{i-1} - J) / 2$ 
         $\hat{l}_i = (\hat{l}_i + \hat{l}_{i-1} + J) / 2$ 
    end
end

```

where  $J$  is 0.0012 for the first pass. Using the selected first stage vector  $L1$  and the lower part of the second stage  $L2$ , the higher part of the second stage is searched from the codebook **L3**. Again, the rearrangement procedure is used to guarantee a minimum distance of 0.0012. The resulting vector  $\hat{l}_i$ ,  $i = 1, \dots, 10$ , is rearranged to guarantee a minimum distance of 0.0006. The quantized LSF coefficients for the current frame  $m$  are obtained from the weighted sum of previous quantizer outputs  $\hat{l}_i^{(m-k)}$ , and the current quantizer output  $\hat{l}_i^{(m)}$ :

$$\hat{\omega}_i^{(m)} = \left( 1 - \sum_{k=1}^4 \hat{p}_{i,k} \right) \hat{l}_i^{(m)} + \sum_{k=1}^4 \hat{p}_{i,k} \hat{l}_i^{(m-k)}, \quad i = 1, \dots, 10$$

There are two MA predictors. Which MA predictor to use is defined by a single bit  $L0$ . It is determined by the one that minimizes the weighted mean-squared error:

$$E_{lsf} = \sum_{i=1}^{10} w_i (\omega_i - \hat{\omega}_i)^2$$

The weights  $w_i$  are made adaptive as a function of the unquantized LSF coefficients,

$$w_1 = \begin{cases} 1.0 & , \text{ if } \omega_2 - 0.04\pi - 1 > 0 \\ 10(\omega_2 - 0.04\pi - 1)^2 + 1, & \text{ otherwise} \end{cases}$$

$$w_i = \begin{cases} 1.0 & , \text{ if } \omega_{i+1} - \omega_{i-1} - 1 > 0 \\ 10(\omega_{i+1} - \omega_{i-1} - 1)^2 + 1, & \text{ otherwise} \end{cases}, \quad 2 \leq i \leq 9$$

$$w_{10} = \begin{cases} 1.0 & , \text{ if } -\omega_9 + 0.92\pi - 1 > 0 \\ 10(-\omega_9 + 0.92\pi - 1)^2 + 1, & \text{ otherwise} \end{cases}$$

In addition, the weights  $w_5$  and  $w_6$  are multiplied by 1.2 each.

4. To avoid sharp resonance in the LP filter, the rearrangement procedure is applied twice to the resulting vector  $\hat{l}_i$  with  $\mathcal{J} = 0.0012$  and  $0.0006$  sequentially.

5. After computing quantized LSF coefficients, the corresponding filter is checked for stability. This is done as follows:

Order the coefficients in increasing value;

```

if  $\hat{\omega}_i < 0.005$  then  $\hat{\omega}_i = 0.005$ ;
if  $\hat{\omega}_{i+1} - \hat{\omega}_i < 0.0391$  then  $\hat{\omega}_{i+1} = \hat{\omega}_i + 0.0391$ ,  $i = 1, \dots, 9$ ;
if  $\hat{\omega}_{10} > 3.315$  then  $\hat{\omega}_{10} = 3.315$ .

```

6. Convert the quantized LSF vectors to LSP vectors.

The function `ippsLSPQuant` is actually a combination of `ippsLSPToLSF`, `ippsLSFQuant`, and `ippsLSFToLSP` functions. The following code details the correspondence.

```

IppStatus ippsLSPQuant_G729_16s(const Ipp16s* pSrcLSP, Ipp16s*
pSrcDstPrevFreq, Ipp16s* pDstQLSP, Ipp16s* pDstQLSPIndex) {
    __ALIGN(8) short lsf[LP_ORDER];

    __ALIGN(8) short lsp_q[LP_ORDER];

    short q_index[4];
    IppStatus sts=ippsLSPToLSF_G729_16s(pSrcLSP, lsf);
    if (sts != ippsStsNoErr) return sts;
    sts = ippsLSFQuant_G729_16s(lsf, pSrcDstPrevFreq, lsf_q, q_index );
    if(sts != ippsStsNoErr) return sts;
    pDstQLSPIndex[0] = (q_index[0]<<G729_NC0_B) | q_index[1];
    pDstQLSPIndex[1] = (q_index[2]<<G729_NC1_B) | q_index[3];
    ippsLSFToLSP_G729_16s(lsp_q, pDstQLSP)
    return ippsStsNoErr;
}

```

### Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LSPToLSF\_G729

*Converts LSP coefficients to LSF coefficients.*

---

### Syntax

```

IppStatus ippsLSPToLSF_Norm_G729_16s (const Ipp16s* pLSP, Ipp16s* pLSF);
IppStatus ippsLSPToLSF_G729_16s (const Ipp16s* pLSP, Ipp16s* pLSF);

```

## Parameters

<i>pLSP</i>	Pointer to the LSP vector, in Q15 in the range [-1:1].
<i>pLSF</i>	Pointer to the LSF vector. Values must be scaled by 13 bits in the range [0:π] (for <code>ippsLSPToLSF_G729_16s</code> function) or by 15 bits in the range [0: 0.5] (for <code>ippsLSPToLSF_Norm_G729_16s</code> function).

## Description

These functions are declared in the `ippsc.h` file. The function `ippsLSPToLSF_Norm_G729` converts the LSP coefficients to LSF coefficients as follows:

$$pLSF[i] = 2^{scaleFactor} * \arccos(pLSP[i]), 0 \leq i < 10.$$

The scale factor is chosen such that the first LSF coefficient is normalized to the interval [0:0.5] by multiplying to  $1/\pi$ . The function `ippsLSPToLSF_G729` retains the result within the interval [0:π]. Both functions use the same approximation table to calculate the inverse cosine.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pLSP</i> or <i>pLSF</i> pointer is NULL.

## LagWindow\_G729

*Applies 60Hz bandwidth expansion.*

### Syntax

```
IppStatus ippsLagWindow_G729_32s_I (Ipp32s* pSrcDst, int len);
```

### Parameters

<i>pSrcDst</i>	Pointer to the autocorrelation vector.
<i>len</i>	Number of elements in the vector.

### Description

The function `ippsLagWindow_G729` is declared in the `ippsc.h` file. This function applies a 60Hz bandwidth expansion to the autocorrelation vector as follows:

```

r[0] = 1.0001*r[0]
r[i] = wlag[i]*r[i], i = 0,.. len-1

```

where

```

wlag[i] = exp(-1/2*(2πf0 *i/fs)2),
and f0 = 60Hz, fs = 8000Hz.

```

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsRangeErr</code>	Indicates an error when <code>len &gt; 12</code> .

## Codebook Search Functions

These functions perform open loop pitch estimation using the adaptive codebook, and search in ACELP excitation (fixed) codebook for optimal signs and positions of the pulses.

## OpenLoopPitchSearch\_G729

*Searches for an optimal pitch value.*

---

### Syntax

```

IppStatus ippOpenLoopPitchSearch_G729_16s(const Ipp16s* pSrc, Ipp16s*
bestLag);

IppStatus ippOpenLoopPitchSearch_G729A_16s(const Ipp16s* pSrc, Ipp16s*
bestLag);

IppStatus ippOpenLoopPitchSearch_G729A_32f(const Ipp32f* pSrc, Ipp16s*
bestLag);

```

### Parameters

<code>pSrc</code>	Pointer to the perceptually weighted speech signal vector [170].
<code>bestLag</code>	Pointer to the open-loop pitch search result.

## Description

These functions are declared in the `ippsc.h` file. They use an open-loop method to get the pitch value. To reduce the complexity of the search for the best adaptive-codebook delay, the search range is limited around a candidate delay, which is obtained from maximizing the correlation coefficients of the perceptually weighted speech.

The function `ippsOpenLoopPitchSearch_G729A` is designed for G.729A codec [ITU729A].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## AdaptiveCodebookSearch\_G729

*Searches for the integer delay and the fraction delay, and computes the adaptive vector.*

### Syntax

```
IppStatus ippsAdaptiveCodebookSearch_G729_16s(Ipp16s valOpenDelay, const
Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s* pDstAdptVector, Ipp16s
subFrame);
```

```
IppStatus ippsAdaptiveCodebookSearch_G729A_16s(Ipp16s valOpenDelay, const
Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s* pDstAdptVector, Ipp16s
subFrame);
```

```
IppStatus ippsAdaptiveCodebookSearch_G729D_16s (Ipp16s valOpenDelay, const
Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
pSrcDstPrevExcitation, Ipp16s* pDstDelay, Ipp16s subFrame);
```

### Parameters

<code>valOpenDelay</code>	Open-loop delay, in the range [18,145].
<code>pSrcAdptTarget</code>	Pointer to the target signal for adaptive-codebook search vector [40].
<code>pSrcImpulseResponse</code>	Pointer to the impulse response of weighted synthesis filter vector [40], in Q12.

<i>pSrcDstPrevExcitation</i>	Pointer to the previous and updated excitation vector [194].
<i>pDstDelay</i>	Pointer to the integer delay and fraction delay vector [2].
<i>pDstAdptVector</i>	Pointer to the adaptive vector [40].
<i>subFrame</i>	Subframe number, 0 or 1.

## Description

These functions are declared in the `ippsc.h` file. They search for the integer delay  $T$  and the fraction delay  $t$ , using the target signal and the past filtered excitation, and compute the adaptive vector. They are applied in subframes.

The function `ippsAdaptiveCodebookSearch_G729A` is designed for G.729A codec [ITU729A].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>valOpenDelay</code> is not in the range [18, 145], or <code>subFrame</code> is not 0 or 1.

## DecodeAdaptiveVector\_G729

*Restores the adaptive codebook vector by interpolating the past excitation.*

---

### Syntax

```

IppStatus ippsDecodeAdaptiveVector_G729_16s(const Ipp16s* pSrcDelay, Ipp16s*
pSrcDstPrevExcitation, Ipp16s* pDstAdptVector);

IppStatus ippsDecodeAdaptiveVector_G729_16s_I(const Ipp16s* pSrcDelay, Ipp16s*
pSrcDstPrevExcitation);

IppStatus ippsDecodeAdaptiveVector_G729_32f_I(const Ipp32s* pSrcDelay, Ipp32f*
pSrcDstPrevExcitation);

```

### Parameters

<i>pSrcDelay</i>	Pointer to the integer and fraction delay for each subframe.
------------------	--



*pSrcDstPrevExcitation* Pointer to the past and updated excitations vector [194].

*pDstAdptVector* Pointer to the adaptive codebook vector [40].

## Description

The function `ippsDecodeAdaptiveVector_G729` is declared in the `ippsc.h` file. This function restores the adaptive codebook vector by interpolating the past excitation. It performs the following steps:

1. Decodes the integer part and fractional part of the pitch delay  $T$  and  $t$ .
2. Interpolates the past excitation to get the current adaptive excitation as:

$$v(n) = \sum_{i=0}^9 u(n-T-i) \times b_{30}(t+3i) + \sum_{i=0}^9 u(n-T+1+i) \times b_{30}(3-t+3i) ,$$

$n = 0, 1, \dots, 39$

## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

`ippStsRangeErr` Indicates an error when *pSrcDelay*[0] is not in the range [18, 145], or *pSrcDelay*[1] is not -1, 0 or 1.

## FixedCodebookSearch\_G729

*Searches for the fixed codebook vector.*

### Syntax

```
Ippl6s* ippsFixedCodebookSearch_G729_16s(const Ippl6s* pSrcFixedCorr,
Ippl6s* pSrcDstMatrix, Ippl6s* pDstFixedVector, Ippl6s* pDstFixedIndex,
Ippl6s* pSearchTimes, Ippl6s subFrame);
```

```
Ippl6s* ippsFixedCodebookSearch_G729_32s16s(const Ippl6s* pSrcFixedCorr,
Ippl32s* pSrcDstMatrix, Ippl6s* pDstFixedVector, Ippl6s* pDstFixedIndex,
Ippl6s* pSearchTimes, Ippl6s subFrame);
```

```

IppStatus ippsFixedCodebookSearch_G729A_16s(const Ipp16s* pSrcFixedCorr,
Ipp16s* pSrcDstMatrix, Ipp16s* pDstFixedVector, Ipp16s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729A_32s16s(const Ipp16s* pSrcFixedCorr,
Ipp32s* pSrcDstMatrix, Ipp16s* pDstFixedVector, Ipp16s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729E_16s(int mode, const Ipp16s*
pSrcFixedTarget, const Ipp16s* pSrcLtpResidual, const Ipp16s*
pSrcImpulseResponse, Ipp16s* pDstFixedVector, Ipp16s* pDstFltFixedVector,
Ipp16s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729D_16s(const Ipp16s* pSrcFixedCorr,
const Ipp16s* pSrcImpulseResponse, Ipp16s* pSrcDstMatrix, Ipp16s*
pDstFixedVector, Ipp16s* pDstFltFixedVector, Ipp16s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729_32f (const Ipp32f* pSrcFixedCorr,
Ipp32f* pSrcDstMatrix, Ipp32f* pDstFixedVector, Ipp32s* pDstFixedIndex,
Ipp32s* pSearchTimes, Ipp32s subFrame);

IppStatus ippsFixedCodebookSearch_G729A_32f (const Ipp32f* pSrcFixedCorr,
Ipp32f* pSrcDstMatrix, Ipp32f* pDstFixedVector, Ipp32s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729D_32f (Ipp32f* pSrcDstFixedCorr, Ipp32f*
pSrcDstMatrix, const Ipp32f* pSrcImpulseResponse, Ipp32f* pDstFixedVector,
Ipp32f* pDstFltFixedVector, Ipp32s* pDstFixedIndex);

IppStatus ippsFixedCodebookSearch_G729E_32f (int mode, Ipp32f*
pSrcDstFixedCorr, const Ipp32f* pSrcLtpResidual, const Ipp32f*
pSrcImpulseResponse, Ipp32f* pDstFixedVector, Ipp32f* pDstFltFixedVector,
Ipp32s* pDstFixedIndex);

```

## Parameters

<i>mode</i>	Indicates forward (= 0) or backwards (= 1) LP analysis mode.
<i>pSrcFixedCorr</i>	Pointer to the correlation signal of fixed-codebook search vector [40], in Q13.
<i>pSrcDstMatrix</i>	Pointer to the Toeplitz matrix of length 616 (elements of Ipp16s type in Q9 or Ipp32s type in Q25).
<i>pSrcFixedTarget</i>	Pointer to the input updated target speech vector.
<i>pSrcLtpResidual</i>	Pointer to the input residual after long term prediction vector.

<i>pSrcImpulseResponse</i>	Pointer to the LP synthesis filter [40].
<i>pDstFixedVector</i>	Pointer to the fixed-codebook vector [40], in Q13.
<i>pDstFltFixedVector</i>	Pointer to the output filtered fixed-codebook vector.
<i>pDstFixedIndex</i>	Pointer to the fixed-codebook index. The first element is the sign codeword <i>s</i> and the second element is the position codeword <i>c</i> . Index length is the following: 2 - for all G729, all G729A, and G729D_32f flavors; 4 - for all G729E flavors; 1 - for G729D_16s flavor.
<i>pSearchTimes</i>	Pointer to the maximum search time for the remaining subframes.
<i>subFrame</i>	Subframe number, 0 or 1.

## Description

These functions are declared in the `ippsc.h` file. They search for the fixed-codebook vector and corresponding vector index, respectively. They are applied in subframes.

The first function `ippsFixedCodebookSearch_G729` is designed for G.729/B codec and uses the nested-loop search approach [ITU729B]. The Toeplitz matrix is updated in this function.

The function `ippsFixedCodebookSearch_G729A` is designed for G.729A codec and uses the iterative depth-first, tree search approach [ITU729A].

Both `ippsFixedCodebookSearch_G729` and `ippsFixedCodebookSearch_G729A` functions search four signed pulses in positions given by following table:

Pulses	Positions
$i_0$	0,5,10,15,20,25,30,35
$i_1$	1,6,11,16,21,26,31,36
$i_2$	2,7,12,17,22,27,32,37
$i_3$	3,8,13,18,23,28,33,38,4,9,14,19,24,29,34,39

The function `ippsFixedCodebookSearch_G729E` is designed for G.729E codec and searches ten signed pulses in five tracks in positions shown in the following table:

Pulses	Positions
$i_0, i_1$	0,5,10,15,20,25,30,35

Pulses	Positions
$i_2, i_3$	1,6,11,16,21,26,31,36
$i_4, i_5$	2,7,12,17,22,27,32,37
$i_6, i_7$	3,8,13,18,23,28,33,38
$i_8, i_9$	4,9,14,19,24,29,34,39

The two pulses for each track may overlap resulting in a single pulse with duplicated amplitude.

The function `ippsFixedCodebookSearch_G729D` is designed for G.729D codec and searches for two signed pulses in two overlapping tracks given in following table:

Pulses	Positions
$i_0$	1, 3, 6, 8, 11, 13, 16, 18, 21, 23, 26, 28, 31, 33, 36, 38
$i_1$	0, 1, 2, 4, 5, 6, 7, 9, 10, 11, 12, 14, 15, 16, 17, 19, 20, 21, 22, 24, 25, 26, 27, 29, 30, 31, 32, 34, 35, 36, 37, 39

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>subFrame</code> is not 0 or 1 (for integer functions only).

## GainCodebookSearch\_G729

*Finds pitch and fixed gains.*

### Syntax

```

IppStatus ippsGainCodebookSearch_G729_32f( const Ipp32f* pSrcCorrFactors,
Ipp32f valPredictedCodebookGain, const int* pSrcCand, Ipp32f* pDstIdxs,
Ipp32s valTimeFlag);

```

```
IppStatus ippsGainCodebookSearch_G729D_32f (const Ipp32f* pSrcCorrFactors,
Ipp32f valPredictedCodebookGain, const int* pSrcCand, Ipp32f* pDstIdxs,
Ipp32s valTimeFlag);
```

## Parameters

<i>pSrcCorrFactors</i>	Pointer to the correlations.
<i>valPredictedCodebookGain</i>	Predicted codebook gain.
<i>pSrcCand</i>	Pointer to the pre-searched candidates.
<i>pDstIdxs</i>	Pointer to the output indexes.
<i>valTimeFlag</i>	Taming process indicator.

## Description

The functions `ippsGainCodebookSearch_G729` and `ippsGainCodebookSearch_G729D` are declared in the `ipps.h` file. As part of LSP quantization algorithm (refer to [ippsGainQuant\\_G729](#) for details), these functions operate in floating-point arithmetic and perform selection of the gain vector in the codebook *GA* and *GB* in the range limited by *pSrcCand*[0] and *pSrcCand*[1]. The search algorithm consists of minimizing the following expression:

$$E = pSrcCorrFactors[0] \times G_p^2 + pSrcCorrFactors[2] \times G_c^2 + pSrcCorrFactors[1] \times G_p + pSrcCorrFactors[3] \times G_c + pSrcCorrFactors[1] \times G_p \times G_c,$$

where  $G_p$  and  $G_c$  are optimum pitch and code gains, respectively, in selected domain.

The `ippsGainCodebookSearch_G729_32f` function uses the 7-bit codebook, whereas the function `ippsGainCodebookSearch_G729D_32f` uses the 6-bit codebook.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>valTimeFlag</i> is not 0 or 1.
<code>ippStsRangeErr</code>	For the <b>G729</b> function flavor: indicates an error when <i>pSrcCand</i> [0] is not in the range [0,4) or <i>pSrcCand</i> [1] is not in the range [0,8). For the <b>G729D</b> function flavor: indicates an error when <i>pSrcCand</i> [0] or <i>pSrcCand</i> [1] is not in the range [0,2).

## ToeplizMatrix\_G729

*Calculates elements of the Toepliz matrix for the fixed codebook search.*

---

### Syntax

```

IppStatus ippsToeplizMatrix_G729_16s(const Ipp16s* pSrcImpulseResponse,
Ipp16s* pDstMatrix);

IppStatus ippsToeplizMatrix_G729_16s32s(const Ipp16s* pSrcImpulseResponse,
Ipp32s* pDstMatrix);

IppStatus ippsToeplizMatrix_G729_32f(const Ipp32f* pSrcImpulseResponse,,
Ipp32f* pDstMatrix);

IppStatus ippsToeplizMatrix_G729D_32f(const Ipp32f* pSrcImpulseResponse,,
Ipp32f* pDstMatrix);

```

### Parameters

*pSrcImpulseResponse* Pointer to the impulse response vector [40], in Q12 for 16s data.

*pDstMatrix* Pointer to the elements of Toepliz matrix of length 616, (of `Ipp16s` type in Q9 or `Ipp32s` type in Q25).

### Description

The function `ippsToeplizMatrix_G729` is declared in the `ippsc.h` file. This function calculates 616 elements in Toepliz matrix for the fixed-codebook search. These elements can be expressed as:

$$\Phi(i, j) = \sum_{n=j}^{39} h(n-i) \times h(n-j), \quad 0 \leq i \leq 39, \quad 0 \leq j \leq 39$$

where  $h(i)$ ,  $i = 0, 1, \dots, 39$ , is the impulse response.

The function stores the calculated 616 elements in *pDstMatrix* in the following order:

1.  $\Phi(m_i, m_i)$ ,  $i = 0, 1, 2, 3$ ,  $3 \times 8 + 16 = 40$  elements; starting location: 0.

$\Phi(0, 0), \Phi(5, 5), \dots, \Phi(35, 35), \Phi(1, 1), \Phi(6, 6), \dots, \Phi(36, 36),$   
 $\Phi(2, 2), \Phi(7, 7), \dots, \Phi(37, 37), \Phi(3, 3), \Phi(8, 8), \dots, \Phi(39, 39)$  .

2.  $\Phi(m_0, m_1)$ ,  $8 \times 8 = 64$  elements; starting location: 40.

$\Phi(0, 1), \dots, \Phi(0, 36), \Phi(5, 1), \dots, \Phi(5, 36), \Phi(10, 1), \dots, \Phi(35, 36).$

3.  $\Phi(m_0, m_2)$ ,  $8 \times 8 = 64$  elements; starting location: 104.

$\Phi(0, 2), \dots, \Phi(0, 37), \Phi(5, 2), \dots, \Phi(5, 37), \Phi(10, 2), \dots, \Phi(35, 37).$

4.  $\Phi(m_0, m_3)$ ,  $8 \times 16 = 128$  elements; starting location: 168.

$\Phi(0, 3), \dots, \Phi(0, 38), \Phi(5, 3), \dots, \Phi(5, 38), \Phi(10, 3), \dots, \Phi(35, 38).$

$\Phi(0, 4), \dots, \Phi(0, 39), \Phi(5, 4), \dots, \Phi(5, 39), \Phi(10, 4), \dots, \Phi(35, 39).$

5.  $\Phi(m_1, m_3)$ ,  $8 \times 8 = 64$  elements; starting location: 296.

$\Phi(1, 2), \dots, \Phi(1, 37), \Phi(6, 2), \dots, \Phi(6, 37), \Phi(11, 2), \dots, \Phi(36, 37).$

4.  $\Phi(m_1, m_3)$ ,  $8 \times 16 = 128$  elements; starting location: 360.

$\Phi(1, 3), \dots, \Phi(1, 38), \Phi(6, 3), \dots, \Phi(6, 38), \Phi(11, 3), \dots, \Phi(36, 38).$

$\Phi(1, 4), \dots, \Phi(1, 39), \Phi(6, 4), \dots, \Phi(6, 39), \Phi(11, 4), \dots, \Phi(36, 39).$

4.  $\Phi(m_2, m_3)$ ,  $8 \times 16 = 128$  elements; starting location: 488.

$\Phi(2, 3), \dots, \Phi(2, 38), \Phi(7, 3), \dots, \Phi(7, 38), \Phi(12, 3), \dots, \Phi(37, 38).$

$\Phi(2, 4), \dots, \Phi(2, 39), \Phi(7, 4), \dots, \Phi(7, 39), \Phi(12, 4), \dots, \Phi(37, 39).$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

### Codebook Gain Functions

These functions can be used to predict, quantize, decode, and control fixed- and adaptive-codebook gains both in G.729 encoding and decoding procedures.

## DecodeGain\_G729

Decodes the adaptive and fixed-codebook gains.

### Syntax

```
IppStatus ippsDecodeGain_G729_16s (Ipp32s energy, Ipp16s* pPastEnergy, const
Ipp16s* pQuaIndex, Ipp16s* pGain);
```

```
IppStatus ippsDecodeGain_G729I_16s(Ipp32s energy, Ipp16s valGainAttenuation,
Ipp16s* pPastEnergy, const Ipp16s* pQuaIndex, Ipp16s* pGain);
```

### Parameters

<i>energy</i>	Input energy of the codeword.
<i>pPastEnergy</i>	Pointer to the input/output vector (in Q14) of the log-energies for fixed codebook contributions of the 4 previous subframes.
<i>pQuaIndex</i>	NULL for frame erasure; otherwise, pointer to the vector of codebook indices: <i>pQuaIndex</i> [0] - first stage codebook index, <i>pQuaIndex</i> [1] - second stage codebook index.
<i>valGainAttenuation</i>	Attenuation factor for the gains in case of frame erasure ( <i>pQuaIndex</i> = NULL).
<i>pGain</i>	Pointer to the input/output decoded gain: <i>pGain</i> [0] - adaptive (pitch) gain, <i>pGain</i> [1] - fixed codebook gain.

### Description

The function `ippsDecodeGain_G729` is declared in the `ippsc.h` file. This function decodes the adaptive and fixed-codebook gains. The fixed codebook gain  $g_c$  can be expressed as follows:

$$g_c = \gamma g'_c,$$

where  $g'_c$  is a predicted gain based on previous fixed codebook energies, and  $\gamma$  is a correction factor. The predicted gain is obtained by predicting the log-energy of the current fixed-codebook contribution from the log-energy of the previous fixed-codebook contribution, using the 4th order MA predictor with coefficients [0.68, 0.58, 0.34, 0.19].



The adaptive codebook gain and the factor are vector-quantized using conjugate structured codebooks. The first element in each codebook represents the quantized adaptive-codebook gain and the second element represents the quantized fixed-codebook. In case of frame erasure, the gains are the attenuated versions of the previous gains.

**ippsDecodeGain\_G729\_16s.** The two-stage conjugate structured two-dimensional codebook is used: first stage in 3 bits and second stage in 4 bits. In case of frame erasure, factors 0.9 and 0.98 in Q15 are used for attenuation of the adaptive and fixed codebook gains, respectively.

**ippsDecodeGain\_G729I\_16s.** The new 6-bit conjugate structured codebook is used. In case of frame erasure, the factor given by *valGainAttenuation* is used for attenuation of both the adaptive and fixed codebook gains.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## GainControl\_G729

*Calculates adaptive gain control.*

---

### Syntax

```
IppStatus ippsGainControl_G729_16s_I (const Ipp16s* pSrc, Ipp16s* pSrcDst,
Ipp16s* pGain);

IppStatus ippsGainControl_G729A_16s_I (const Ipp16s* pSrc, Ipp16s* pSrcDst,
Ipp16s* pGain);

IppStatus ippsGainControl_G729_32f_I (Ipp32f gainScalingFactor, Ipp32f
gainFactor, Ipp32f* pSrcDst, Ipp32f* pGain);
```

### Parameters

<i>pSrc</i>	Pointer to the source reconstructed speech vector [40].
<i>gainScalingFactor</i>	Gain scaling factor, which is the ratio of energies of the reconstructed speech signal and post-filtered output signal.
<i>gainFactor</i>	Gain adjustment factor.

<i>pSrcDst</i>	Pointer to the input post-filtered signal and output gain-compensated signal vector [40].
<i>pGain</i>	Pointer to the output adaptive gain.

### Description

These functions are declared in the `ippsc.h` file. The function `ippsGainControl_G729` compensates the gain difference between the reconstructed speech signal *sr*, given by *pSrc*, and the filtered signal *sf*, given by *pSrcDst*. First, the gain factor *G* is calculated as follows:

$$G = \frac{\sum_{i=0}^{39} |sr[i]|}{\sum_{i=0}^{39} |sf[i]|}$$

The output gain *spf* for the post-filtered signal *pSrcDst* is calculated as:

$$spf = g^{(n)} * sf(n), n = 0, \dots, 39,$$

where

$$g^{(n)} = 0.85 * g^{(n-1)} + 0.15 * G, n = 0, \dots, 39, g^{(-1)} = 1.0.$$

The function returns  $g^{(39)}$  in *pGain*.

For the function `ippsGainControl_G729A_16s`, the gain factor *G* and factors  $g^{(n)}$  are calculated differently as given by:

$$G = \sqrt{\frac{\sum_{i=0}^{39} |sr[i]|^2}{\sum_{i=0}^{39} |spf[i]|^2}}$$

$$g^{(n)} = 0.9 * g^{(n-1)} + 0.1 * G$$

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## GainQuant\_G729

*Quantizes the codebook gain using a two-stage conjugate-structured codebook.*

### Syntax

```
IppStatus ippGainQuant_G729_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
pSrcFltAdptVector, const Ipp16s* pSrcFixedVector, const Ipp16s*
pSrcFltFixedVector, Ipp16s* pSrcDstEnergyErr, Ipp16s* pDstQGain, Ipp16s*
pDstQGainIndex, Ipp16s tameProcess);
```

```
IppStatus ippGainQuant_G729D_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s*
pSrcFltAdptVector, const Ipp16s* pSrcFixedVector, const Ipp16s*
pSrcFltFixedVector, Ipp16s* pSrcDstEnergyErr, Ipp16s* pDstQGain, Ipp16s*
pDstQGainIndex, Ipp16s tameProcess);
```

### Parameters

`pSrcAdptTarget` Pointer to the target signal for adaptive codebook search vector [40], in Q11.  
`pSrcFltAdptVector` Pointer to the filtered adaptive codebook vector [40], in Q11.

<i>pSrcFixedVector</i>	Pointer to the pre-weighted fixed-codebook vector [40], in Q12.
<i>pSrcFltFixedVector</i>	Pointer to the filtered fixed codebook vector [40], in Q12.
<i>pSrcDstEnergyErr</i>	Pointer to the previous predicted energy error vector [4], in Q10.
<i>pDstQGain</i>	Pointer to the quantized gain ( $G_p$ and $G_c$ ).
<i>pDstQGainIndex</i>	Pointer to the gain codebook indices, $GA$ and $GB$ .
<i>tameProcess</i>	Taming process indicator.

## Description

The functions `ippsGainQuant_G729` and `ippsGainQuant_G729D` are declared in the `ippsc.h` file. These functions use a two-stage conjugate-structured codebook. The function `ippsGainQuant_G729_16s` quantizes the gain using the 7-bit codebook, whereas the function `ippsGainQuant_G729D_16s` uses the 6-bit codebook. The gain codebook search is done by minimizing the mean-squared weighted error between original and reconstructed speech:

$$E = \mathbf{x}^t \mathbf{x} + g_p^2 \mathbf{y}^t \mathbf{y} + g_c^2 \mathbf{z}^t \mathbf{z} - 2g_p \mathbf{x}^t \mathbf{y} - 2g_c \mathbf{x}^t \mathbf{z} + 2g_p g_c \mathbf{y}^t \mathbf{z},$$

where  $\mathbf{x}$  is the adaptive target,  $\mathbf{y}$  is the filtered adaptive vector, and  $\mathbf{z}$  is the filtered fixed vector;  $g_p$  is the pitch gain,  $g_c$  is the fixed gain.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>tameProcess</code> is not 0 or 1.

## AdaptiveCodebookContribution\_G729

*Updates target vector for codebook search by subtracting adaptive codebook contribution.*

---

## Syntax

```

IppStatus ippsAdaptiveCodebookContribution_G729_16s (Ipp16s gain, const
Ipp16s* pFltAdptVector, const Ipp16s* pSrcAdptTarget, Ipp16s* pDstAdptTarget);

IppStatus ippsAdaptiveCodebookContribution_G729_32f (Ipp32f gain, const
Ipp32f* pFltAdptVector, const Ipp32f* pSrcAdptTarget, Ipp32f* pDstAdptTarget);

```

## Parameters

<i>gain</i>	Adaptive codebook gain $g_p$ .
<i>pFltAdptVector</i>	Pointer to the input filtered adaptive codebook vector $y(n)$ .
<i>pSrcAdptTarget</i>	Pointer to the input target vector $x(n)$ .
<i>pDstAdptTarget</i>	Pointer to the output updated target vector $x'(n)$ .

## Description

The function `ippsAdaptiveCodebookContribution_G729` is declared in the `ippsc.h` file. This function subtracts the adaptive codebook contribution from the target signal and stores updated target vector as follows:

$$x'(n) = x(n) - g_p * y(n), n = 0, \dots, 39$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## AdaptiveCodebookGain\_G729

*Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.*

## Syntax

```
IppStatus ippsAdaptiveCodebookGain_G729_16s(const Ipp16s* pSrcAdptTarget,
const Ipp16s* pSrcImpulseResponse, const Ipp16s* pSrcAdptVector, Ipp16s*
pDstFltAdptVector, Ipp16s* pResultAdptGain);
```

```
IppStatus ippsAdaptiveCodebookGain_G729A_16s(const Ipp16s* pSrcAdptTarget,
const Ipp16s* pSrcLPC, const Ipp16s* pSrcAdptVector, Ipp16s*
pDstFltAdptVector, Ipp16s* pResultAdptGain);
```

## Parameters

<i>pSrcAdptTarget</i>	Pointer to the adaptive target signal vector [40].
<i>pSrcImpulseResponse</i>	Pointer to the impulse response of the perceptual weighting filter vector [40], in Q12.

<i>pSrcAdptVector</i>	Pointer to the adaptive-codebook vector [40], in Q12 (for <code>ippsAdaptiveCodebookGain_G729_16s</code> ) or in Q10 (for <code>ippsAdaptiveCodebookGain_G729A_16s</code> ).
<i>pSrcLPC</i>	Pointer to the LPC coefficients of the synthesis filter vector [11], in Q12.
<i>pDstFltAdptVector</i>	Pointer to the filtered adaptive-codebook vector [40].
<i>pResultAdptGain</i>	Pointer to the adaptive-codebook gain, in Q14.

### Description

These functions are declared in the `ippsc.h` file. They compute the gain of the adaptive-codebook vector, and calculate the filtered adaptive-codebook vector, respectively. Both functions are applied in subframes.

The function `ippsAdaptiveCodebookGain_G729` is designed for G.729/B codec [\[ITU729B\]](#).

The function `ippsAdaptiveCodebookGain_G729A` is designed for G.729A codec [\[ITU729A\]](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Filter Functions

These functions are used to perform different types of filtering, including high pass filtering on pre-processing stage of encoding and post-processing stage of decoding, as well as the long-term and short-term postfilter in decoding.

The residual (FIR), simple synthesis (IIR), harmonic, and preemphasize filter functions may be combined to perform more complex synthesis filtering and may be used in different encode and decode stages.

## ResidualFilter\_G729

*Implements an inverse LP filter and obtains the residual signal.*

---

### Syntax

```
ippStatus ippsResidualFilter_G729_16s(const Ipp16s* pSrcSpch, const Ipp16s*
pSrcLPC, Ipp16s* pDstResidual);
```

```
IppStatus ippsResidualFilter_G729E_16s(const Ipp16s* pCoeffs, int order,
const Ipp16s* pSrc, Ipp16s* pDst, int len);
```

### Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal. Elements <i>pSrcSpch</i> [0...39] are the present speech signals, whereas <i>pSrcSpch</i> [-10... -1] are the history to be used.
<i>pSrcLPC</i>	Pointer to the LP coefficients vector [11], in Q12.
<i>pDstResidual</i>	Pointer to the LP residual.
<i>pCoeffs</i>	Pointer to the vector [ <i>order</i> + 1] of filter coefficients.
<i>order</i>	The filter order.
<i>pSrc</i>	Pointer to the source vector [ <i>len</i> ]. Values <i>pSrc</i> [- <i>order</i> ,..,-1] must also be supplied.
<i>pDst</i>	Pointer to the output filtered residual vector [ <i>len</i> ].
<i>len</i>	Length of the source and destination vectors.

### Description

The functions `ippsResidualFilter_G729` and `ippsResidualFilter_G729E` are declared in the `ippsc.h` file.

**ippsResidualFilter\_G729.** This function filters the input speech signal through an inverse LP filter and gets the residual signal as:

$$r(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i \times s(n-i) ,$$

$n = 0, 1, \dots, 39$  .

**ippsResidualFilter\_G729E.** This function implements the same operation as the previous function, but accepts the variable filter order.

### Return Values

`ippStsNoErr` Indicates no error.

<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>order</code> is less than or equal to 0, or when <code>len</code> is less than or equal to 0 and <code>valLPCOrder &gt; len</code> .

## SynthesisFilter\_G729

*Reconstructs the speech signal from LP coefficients and residuals.*

---

### Syntax

```
IppStatus ippsSynthesisFilter_G729_16s(const Ipp16s* pSrcResidual, const
Ipp16s* pSrcLPC, Ipp16s* pSrcDstSpch);
```

```
IppStatus ippsSynthesisFilterZeroStateResponse_NR_16s(const Ipp16s* pSrcLPC,
Ipp16s* pDstImp, int len, int scaleFactor);
```

```
IppStatus ippsSynthesisFilter_G729E_16s(const Ipp16s* pLPC, int order, const
Ipp16s* pSrc, Ipp16s* pDst, int len, const Ipp16s* pMem);
```

```
IppStatus ippsSynthesisFilter_G729E_16s_I(const Ipp16s* pLPC, int order,
Ipp16s* pSrcDst, int len, const Ipp16s* pMem);
```

```
IppStatus ippsSynthesisFilter_G729_32f(const Ipp32f* pLpc, int order, const
Ipp32f* pSrc, Ipp32f* pDst, int len, Ipp32f* pMem);
```

### Parameters

<code>pSrcResidual</code>	Pointer to the LP residual signal vector [40].
<code>pSrcLPC</code>	Pointer to the LP coefficients vector [11], in Q12 for 16s data.
<code>pSrcDstSpch</code>	Pointer to the synthesized and updated speech. Elements <code>pSrcDstSpch[0...39]</code> are the present synthesized speech, <code>pSrcDstSpch[-10...-1]</code> are the history to be used.
<code>pLPC</code>	Pointer to the LP coefficients vector [ <code>order+1</code> ].
<code>order</code>	The filter order. ( <code>pLPC[order+1]</code> vector must be supplied on input.)
<code>pSrc</code>	Pointer to the source vector [ <code>len</code> ].
<code>pDst</code>	Pointer to the destination speech vector [ <code>len</code> ].



<i>pSrcDst</i>	Pointer to the source and destination vector [ <i>len</i> ].
<i>pDstImp</i>	Pointer to the destination zero state impulse response vectore [ <i>len</i> ].
<i>len</i>	Length of the source and destination vectors.
<i>pMem</i>	Pointer to the filter memory vector [ <i>order</i> ] supplied for filtering.
<i>scaleFactor</i>	Scale factor for the input LP coefficients and output impulse response, accordingly.

## Description

These functions are declared in the `ippsc.h` file.

The function `ippsSynthesisFilter_G729_16s` uses the default LP filter of 10-th order to reconstruct speech signal from the residual signal as:

$$\hat{s}(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i \hat{s}(n-i) , \quad n = 0, 1, \dots, 39$$

$n = 0, 1, \dots, 39$ .

The function `ippsSynthesisFilterZeroStateResponse_NR_16s` performs the same operation as `ippsSynthesisFilter_NR_16s` for the same parameters, but uses no memory and processes the impulse input vector *pSrc* which is set as

$pSrc[0] = 2^{scaleFactor}$ ,  $pSrc[i] = 0$ ,  $i = 1, \dots, len - 1$ .

The functions `ippsSynthesisFilter_G729E_16s`, `ippsSynthesisFilter_G729E_16s_I`, and `ippsSynthesisFilter_G729_32f` perform synthesis filtering of the order given by the parameter *order* and operate like `ippsSynthesisFilter_G729_16s` function for the default order 10.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>order</i> is less than or equal to 0.
<code>ippStsOverflow</code>	Indicates a warning that an overflow has occurred.

## LongTermPostFilter\_G729

*Restores the long-term information from the old speech signal.*

---

### Syntax

```
IppStatus ippsLongTermPostFilter_G729_16s (Ipp16s gammaFactor, int valDelay,
const Ipp16s* pSrcDstResidual, Ipp16s* pDstFltResidual, Ipp16s* pResultVoice);

IppStatus ippsLongTermPostFilter_G729A_16s(Ipp16s valDelay, const Ipp16s*
pSrcSpch, const Ipp16s* pSrcLPC, Ipp16s* pSrcDstResidual, Ipp16s*
pDstFltResidual);

IppStatus ippsLongTermPostFilter_G729B_16s(Ipp16s valDelay, const Ipp16s*
pSrcSpch, const Ipp16s* pSrcLPC, Ipp16s* pSrcDstResidual, Ipp16s*
pDstFltResidual, Ipp16s* pResultVoice, Ipp16s frameType);
```

### Parameters

<i>gammaFactor</i>	Post filter coefficient $\gamma_p$ in Q15.
<i>valDelay</i>	Pitch delay.
<i>pSrcSpch</i>	Pointer to the reconstructed speech $\hat{s}(n)$ , in Q15. Elements <i>pSrcSpch</i> [0...39] are the present speech signals, elements <i>pSrcSpch</i> [-10...-1] are the history that is used by the function.
<i>pSrcLPC</i>	Pointer to the LP coefficients $\hat{a}_i$ vector [11], in Q12.
<i>pSrcDstResidual</i>	Pointer to the residual after long term prediction vector [40], elements [-152...-1] are the long term prediction history that will be used.
<i>pDstFltResidual</i>	Pointer to the output filtered residual vector [40].
<i>pResultVoice</i>	Pointer to the voice information.
<i>frameType</i>	The type of the frame: 0 - stands for untransmitted frame, 1 - stands for normal speech frame, 2 - stands for SID frame.

### Description

These functions are declared in the `ipps.h` file. These FIR filters are used to restore the long-term relationship from old speech at the length of pitch. They are applied in subframes.

The first function `ippsLongTermPostFilter_G729` is designed for G.729 codec [ITU729].

The function `ippsLongTermPostFilter_G729A` is designed for G.729A codec [ITU729A].

The function `ippsLongTermPostFilter_G729B` is designed for G.729B codec [ITU729B] and actually is the combination of other functions as demonstrated by the code below:

```
{
    const short g_pst[11] = {32767,18022,9912,5451,2998,1649,907,499,274,151,83};
    /* y_n = 0.55 powered by i=0,1,...,10 in Q15 */
    short coeffs[LP_ORDER+1];          /* temporary nominator coefficients */
    short vc;

    ippsMul_NR_16s_Sfs(g_pst, pSrcLpc, coeffs, LP_11, 15);
    ippsResidualFilter_G729_16s(pSrcSpch, coeffs, pSrcDstResidual+154);
    if (1 == frameType){
        ippsLongTermPostFilter_G729_16s (16384,valDelay, pSrcDstResidual+ 154,
            pDstFltResidual, &vc); /* Harmonic filtering: g_p=0.5 in Q15*/
        *pResultVoice = (vc != 0);
    }else{
        ippsCopy_16s(pSrcDstResidual + 154,pDstFltResidual,40);
        *pResultVoice = 0;
    }
}
```

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>valDelay</code> is not in the range [18, 145] for the function <code>ippsLongTermPostFilter_G729A</code> , or <code>valDelay</code> is not in the range [0, 143] for the function <code>ippsLongTermPostFilter_G729B</code> , or <code>frameType</code> is not in the range [0, 2].

## ShortTermPostFilter\_G729

*Restores speech signal from the residuals.*

### Syntax

```
IppStatus ippsShortTermPostFilter_G729A_16s(const Ipp16s* pSrcLPC, const
Ipp16s* pSrcFltResidual, Ipp16s* pSrcDstSpch);
```

**THE FOLLOWING FUNCTION IS DEPRECATED:**

```
IppStatus ippsShortTermPostFilter_G729_16s(const Ipp16s* pSrcLPC, const
Ipp16s* pSrcFltResidual, Ipp16s* pSrcDstSpch, Ipp16s* pDstImpulseResponse);
```

**Parameters**

<i>pSrcLPC</i>	Pointer to the quantized LP coefficients vector [11], in Q12.
<i>pSrcFltResidual</i>	Pointer to the residual signal $x(n)$ vector [40], in Q15.
<i>pSrcDstSpch</i>	Pointer to the short-term filtered speech $y(n)$ , in Q15. Elements <i>pSrcDstSpch</i> [0...39] are the present short-term filtered speech signals, elements <i>pSrcDstSpch</i> [-10...-1] are the history that is used by the function.
<i>pDstImpulseResponse</i>	Pointer to the generated impulse response $h_f(n)$ vector [20], in Q12.

**Description**


---

**CAUTION.** THE FUNCTION `ippsShortTermPostFilter_G729_16s` IS DEPRECATED. Please use the function `ippsSynthesisFilter_NR_16s_Sfs` instead.

---

These functions are declared in the `ippsc.h` file. These two IIR filters restore the speech from the residual. They both are applied in subframes.

The function `ippsShortTermPostFilter_G729` is designed for G.729/B codec [ITU729, ITU729B].

The function `ippsShortTermPostFilter_G729A` is designed for G.729A codec [ITU729A].

**Return Values**

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## TiltCompensation\_G729

Compensates for the tilt in the short-term filter.

### Syntax

```
IppStatus ippsTiltCompensation_G729_16s(const Ipp16s* pSrcImpulseResponse,
Ipp16s* pSrcDstSpch);

IppStatus ippsTiltCompensation_G729A_16s(const Ipp16s* pSrcLPC, Ipp16s*
pSrcDstFltResidual);

IppStatus ippsTiltCompensation_G729E_16s(Ipp16s val, const Ipp16s* pSrc,
Ipp16s* pDst);
```

### Parameters

<i>pSrcImpulseResponse</i>	Pointer to the impulse response $h_f(n)$ vector [20], in Q12.
<i>pSrcLPC</i>	Pointer to the gamma weighted LP coefficients vector [22], in Q12. The first 11 elements refer to $\phi_h$ and the next 11 elements refer to $\phi_d$ .
<i>pSrcDstSpch</i>	Pointer to the present and tilt-compensated speech $x(n)$ , in Q15. Elements <i>pSrcDstSpch</i> [0...39] are the present speech signals, element <i>pSrcDstSpch</i> [-1] is the history that will be used.
<i>pSrcDstFltResidual</i>	Pointer to the long-term filtered LP residual, in Q15. Elements <i>pSrcDstFltResidual</i> [0...39] are the present long-term filtered LP residual signals, element <i>pSrcDstFltResidual</i> [-1] is the history that will be used.
<i>val</i>	The input tilt factor.
<i>pSrc</i>	Pointer to the source vector [41].
<i>pDst</i>	Pointer to the output filtered residual vector [40]

### Description

These functions are declared in the `ippsc.h` file. These FIR filters compensate for the tilt in the short-term filter. They are applied in subframes.

The function `ippsTiltCompensation_G729` is designed for G.729/B codec [[ITU729](#), [ITU729B](#)].

The function `ippsTiltCompensation_G729E` is similar but it does not calculate the tilt factor and performs the operation for a given input tilt factor `val` and filter memory specified by `pSrc[0]`.

The function `ippsTiltCompensation_G729A` is designed for G.729A codec [[ITU729A](#)].

### Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## HarmonicFilter

*Calculates the harmonic filter.*

---

### Syntax

```

IppStatus ippsHarmonicFilter_16s_I (Ipp16s beta, int T, Ipp16s* pSrcDst, int
len);

IppStatus ippsHarmonicFilter_NR_16s (Ipp16s beta, int T, const Ipp16s* pSrc,
Ipp16s* pDst, int len);

IppStatus ippsHarmonicFilter_32f_I(Ipp32f beta, int T, Ipp32f* pSrcDst, int
len);

```

### Parameters

<code>beta</code>	The beta factor, in Q15 for 16s data.
<code>T</code>	The integer delay.
<code>pSrc</code>	Pointer to the source vector. Elements <code>pSrc[-T,...,0,...,len-1]</code> are used as input.
<code>pDst</code>	Pointer to the destination vector.
<code>pSrcDst</code>	Pointer to the source and destination vector. Elements <code>pSrcDst[-T,...,0,...,len-1]</code> are used as input, elements <code>pSrcDst[0,...,len-1]</code> are computed by the function.
<code>len</code>	Number of elements in the source and destination vectors.

## Description

These functions are declared in the `ippsc.h` file.

The functions `ippsHarmonicFilter_16s_I` and `ippsHarmonicFilter_32f_I` implement the adaptive pre-filter:

$H(z) = 1/(1-\beta*z^{-T})$ , which can be used to enhance the harmonic component of speech.

The calculation is as follows:

$pSrcDst[n] = pSrcDst[n] + beta * pSrcDst[n-T], 0 \leq n < len.$

The function `ippsHarmonicFilter_NR_16s` implements the harmonic noise shaping filter:

$H(z) = 1+\beta*z^{-T}$

The calculation is as follows:

$pDst[n] = pSrc[n] + beta * pSrc[n-T], 0 \leq n < len.$

Both filters can be used to enhance the harmonic component of the input signal.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## HighPassFilterSize\_G729

*Calculates the G729 high-pass filter size.*

---

### Syntax

```
IppStatus ippsHighPassFilterSize_G729 (int* pSize);
```

### Parameters

*pSize*                      Pointer to the output value of the memory size.

### Description

The function `ippsHighPassFilterSize_G729` is declared in the `ippsc.h` file. This function calculates the size of memory required for the `ippsHighPassFilterInit_G729` function to operate.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is <code>NULL</code> .

## HighPassFilterInit\_G729

*Initializes the high-pass filter.*

---

### Syntax

```
IppStatus ippHighPassFilterInit_G729 (Ipp16s* pCoeff, char* pMemUpdated);
```

### Parameters

<i>pCoeff</i>	Pointer to the array of length 6 containing values $a_0$ , $a_1$ , $a_2$ , $b_0$ , $b_1$ , $b_2$ , in Q12 or Q13.
<i>pMemUpdated</i>	Pointer to the memory allocated for the filter.

### Description

The function initializes the internal data of the high-pass filter:  $a_0$  must be equal to scaled 1 (in Q12 or Q13). The filter history data  $x_{-2}$ ,  $x_{-1}$ ,  $y_{-2}$ ,  $y_{-1}$  are set to zero. The `ippHighPassFilter_G729_16s_ISfs` function uses this memory for filtering.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## HighPassFilter\_G729

*Performs G729 high-pass filtering.*

---

### Syntax

```
IppStatus ippHighPassFilter_G729_16s_ISfs (Ipp16s* pSrcDst, int len, int  
scaleFactor, char* pMemUpdated);
```



## Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector [ <i>len</i> ].
<i>len</i>	Number of elements in the source and destination vectors.
<i>scaleFactor</i>	Scale factor for output data scaling.
<i>pMemUpdated</i>	Pointer to the memory allocated for the filter.

## Description

The function `ippsHighPassFilter_G729` is declared in the `ippsc.h` file. This function performs the input signal pre-processing or the output signal post-processing using the high-pass filter:

$$H_h(z) = \frac{b_0 + b_1 \cdot z^{-1} + b_2 \cdot z^{-2}}{a_0 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2}}$$

Currently, only  $a_0=1$  (in Q12 or Q13) is supported. The function uses the scale factor value equal to 12 for input data pre-filtering, and equal to 13 for output data post-filtering.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is not equal to 12 or 13.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## IIR16s\_G729

*Performs IIR filtering.*

---

### Syntax

```

IppStatus ippsIIR16sLow_G729_16s(const Ipp16s* pCoeffs, const Ipp16s* pSrc,
Ipp16s* pDst, Ipp16s* pMem);

IppStatus ippsIIR16s_G729_16s(const Ipp16s* pCoeffs, const Ipp16s* pSrc,
Ipp16s* pDst, Ipp16s* pMem);

```

## Parameters

<i>pCoeffs</i>	Pointer to the input vector of filter coefficients vector [20]: $b_1, \dots, b_{10}, a_1, \dots, a_{10}$ .
<i>pSrc</i>	Pointer to the input signal vector [40].
<i>pDst</i>	Pointer to the output filtered vector [40].
<i>pMem</i>	Pointer to the filter memory vector [20]. This vector must be initially filled with zeroes.

## Description

The functions `ippsIIR16sLow_G729` and `ippsIIR16s_G729` are declared in the `ippsc.h` file.

These functions perform infinite impulse response (IIR) filtering using the following transfer function:

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The function `ippsIIR16sLow_G729` does not check for overflow and result saturation, whereas the function `ippsIIR16s_G729` makes these checks.

The filter memory is updated.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## PhaseDispersionGetStateSize\_G729D

*Queries the memory length of the phase dispersion filter.*

---

### Syntax

```
IppStatus ippsPhaseDispersionGetStateSize_G729D_16s (int* pSize);
```

### Parameters

*pSize*                      Pointer to the output phase dispersion filter memory length in bytes.

### Description

The function `ippsPhaseDispersionGetStateSize_G729D` reports the size of memory that the phase dispersion filter needs for proper operation.

### Return Values

`ippStsNoErr`               Indicates no error.  
`ippStsNullPtrErr`       Indicates an error when the *pSize* pointer is NULL.

## PhaseDispersionInit\_G729D

*Initializes the phase dispersion filter memory.*

---

### Syntax

```
IppStatus ippsPhaseDispersionInit_G729D_16s  
(IppsPhaseDispersion16s_State_G729D* pPhdMem);
```

### Parameters

*pPhdMem*                      Pointer to the memory of the phase dispersion filter.

### Description

The function `ippsPhaseDispersionInit_G729D` sets the contents of the given buffer as memory of the phase dispersion filter in initial state.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pPhDMem</i> pointer is <code>NULL</code> .

## PhaseDispersionUpdate\_G729D

*Updates the phase dispersion filter state.*

---

### Syntax

```
IppStatus ippPhaseDispersionUpdate_G729D_16s(Ipp16s valPitchGain, Ipp16s
valCodebookGain, IppsPhaseDispersion16s_State_G729D* pPhDMem);
```

### Parameters

<i>valPitchGain</i>	Long term pitch gain.
<i>valCodebookGain</i>	Codebook gain
<i>pPhDMem</i>	Pointer to the memory of phase dispersion filter.

### Description

The function `ippPhaseDispersionUpdate_G729D` updates the state of phase dispersion filter memory with given pitch and codebook gains.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pPhDMem</i> pointer is <code>NULL</code> .

## PhaseDispersion\_G729D

*Performs the phase dispersion filtering.*

---

### Syntax

```
IppStatus ippPhaseDispersion_G729D_16s(Ipp16s valCodebookGain, Ipp16s
valPitchGain, const Ipp16s* pSrcExcSignal, Ipp16s* pDstFltExcSignal, Ipp16s*
pSrcDstInnovation, IppsPhaseDispersion16s_State_G729D* pPhDMem);
```

## Parameters

<code>valPitchGain</code>	Input long term pitch gain.
<code>valCodebookGain</code>	Input codebook gain
<code>pSrcExcSignal</code>	Pointer to the input excitation vector [40].
<code>pDstFltExcSignal</code>	Pointer to the output filtered excitation vector [40].
<code>pSrcDstInnovation</code>	Pointer to the input/output innovative codebook vector [40].
<code>pPhDMem</code>	Pointer to the memory of phase dispersion filter.

## Description

The function `ippsPhaseDispersion_G729D` performs the phase dispersion filtering for the given pitch and codebook gains. The filter alters the innovation signal (mainly its phase), such that a new innovation signal is created with the energy being more spread over the subframe. The filtering is performed by a circular convolution using one of the three "semi-random" impulse responses according to the different amounts of spreading.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Preemphasize\_G729A

*Computes pre-emphasis of a post filter.*

### Syntax

```

IppStatus ippsPreemphasize_G729A_16s (Ipp16s gamma, const Ipp16s* pSrc,
Ipp16s* pDst, int len, Ipp16s* pMem);

IppStatus ippsPreemphasize_G729A_16s_I (Ipp16s gamma, Ipp16s* pSrcDst, int
len, Ipp16s* pMem);

IppStatus ippsPreemphasize_32f_I(Ipp32f gamma, Ipp32f* pSrcDst, int len,
Ipp32f* pMem);

```

## Parameters

<code>gamma</code>	The filter coefficient, in Q15 for 16s data.
--------------------	--

<i>pSrc</i>	Pointer to the source vector, in Q0.
<i>pDst</i>	Pointer to the destination vector, in Q0.
<i>pSrcDst</i>	Pointer to the source and destination vector, in Q0.
<i>len</i>	Number of elements in the source and destination vectors.
<i>pMem</i>	Pointer to the filter memory [1].

## Description

These functions are declared in the `ippsc.h` file and compute pre-emphasis of a post filter.

The computation is performed according to the difference signal pre-emphasis equation:

$$H(z) = 1 - \text{gamma} * z^{-1}$$

The filter memory *pMem* must be initialized to zero and then be constantly updated while filtering.

The functions `ippPreemphasize_G729A` perform no rounding (clipping).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## WinHybridGetStateSize\_G729E

*Queries the length of the hybrid windowing module memory.*

---

### Syntax

```
IppStatus ippWinHybridGetStateSize_G729E_16s (int* pSize);
IppStatus ippWinHybridGetStateSize_G729E_32f (int* pSize);
```

### Parameters

<i>pSize</i>	Pointer to the output length of hybrid windowing module memory in bytes.
--------------	--

### Description

The function `ippsWinHybridGetStateSize_G729E` reports the size of memory that the hybrid windowing module needs for proper operation.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is <code>NULL</code> .

## WinHybridInit\_G729E

Initializes the hybrid windowing module memory.

### Syntax

```
IppStatus ippsWinHybridInit_G729E_16s (IppsWinHybridState_G729E_16s* pMem);
IppStatus ippsWinHybridInit_G729E_32f(IppsWinHybridState_G729E_32f* pMem);
```

### Parameters

<i>pMem</i>	Pointer to the memory of hybrid windowing module.
-------------	---

### Description

The function `ippsWinHybridInit_G729E` sets the contents of the given buffer as the memory of the hybrid windowing module in initial state.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is <code>NULL</code> .

## WinHybrid\_G729E

Applies the hybrid window and computes autocorrelation coefficients.

### Syntax

```
IppStatus ippsWinHybrid_G729E_16s32s (const Ipp16s* pSrcSynthSpeech, Ipp32s*
pDstInvAutoCorr, IppsWinHybridState_G729E_16s* pMem);
```

```
IppStatus ippsWinHybrid_G729E_32f(const Ipp32f* pSrcSynthSpeech, Ipp32f*
pDstInvAutoCorr, IppsWinHybridState_G729E_32f* pMem);
```

### Parameters

<i>pSrcSynthSpeech</i>	Pointer to the input synthesized speech vector [144].
<i>pDstInvAutoCorr</i>	Pointer to the output autocorrelation vector [31].
<i>pMem</i>	Pointer to the memory of hybrid windowing module.

### Description

The function `ippsWinHybrid_G729E` is declared in the `ippsc.h` file. This function applies the hybrid window to input synthesized speech and computes autocorrelation coefficients as follows.

1. First, the input speech vector is multiplied by the hybrid window:

$$S_{w+L}(k) = S_w(k) * w(144 - k), \quad k = 0, \dots, 144.$$

where  $w(k) = \sin((k + 1) * 0.047783)$  for  $k = 0, \dots, 34$ , and  $w(k) = \sin(36 * 0.047783) * a^{(35 - k)}$  for  $k = 35, \dots, 144$ .

Here  $a = 0.9928337491$  so that  $a^{40} = 0.75$  and  $a^{160} = 0.75^4 = 0.31640625$ .

2. Second, the autocorrelation of the windowed speech is calculated by the formulas:

$$R_{rec}(n) = \sum_{k=0}^{k=79} s(k + 30) \cdot s(k + 30 - n), \quad n = 0, \dots, 30$$

$$R(n) = \sum_{k=0}^{k=34} s(k + 110) \cdot s(k + 110 - n), \quad n = 0, \dots, 30$$

3. Third, the output autocorrelation is computed and the memory is updated as follows:

$$mem[k] = 0.31640625 * mem[k] + R_{rec}(k),$$

$$pDstInvAutoCorr[k] = mem[k] + R(k), \quad k = 0, \dots, 30.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------



`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## RandomNoiseExcitation\_G729B

*Initializes a random vector with a Gaussian distribution.*

### Syntax

```
IppStatus ippsRandomNoiseExcitation_G729B_16s (Ipp16s* pSeed, Ipp16s* pExc,
int len);
```

```
IppStatus ippsRandomNoiseExcitation_G729B_16s32f(Ipp16s* pSeed, Ipp32f* pExc,
int len);
```

### Parameters

<code>pSeed</code>	Pointer to the input/output seed of random generator.
<code>pExc</code>	Pointer to the destination vector.
<code>len</code>	Length of the destination vector.

### Description

The function `ippsRandomNoiseExcitation_G729B` is declared in the `ippsc.h` file. This function generates random excitations as follows:

$$pExc[n] = \frac{\sum_{i=1}^{12} \xi_{(12n+i)}}{128}, \quad n = 0, \dots, len-1$$

$n = 0, \dots, len-1$

where  $\xi_{k+1} = 31821 * \xi_k + 13839$ ,  $\xi_0 = pSeed$ .

The seed of the random generator  $\xi$  is updated.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

## FilteredExcitation\_G729

*Computes filtered excitation.*

---

### Syntax

```
IppStatus ippsFilteredExcitation_G729_32f(const Ipp32f* pSrcImpulseResponse,
Ipp32f* pSrcDstPastExc, Ipp32f valExc, Ipp32s len);
```

### Parameters

`pSrcImpulseResponse` Impulse response [`len`].

`pSrcDstPastExc` Pointer to the previous and updated excitation vector [`len`].

`valExc` Excitation

`len` Length.

### Description

The function `ippsFilteredExcitation_G729` is declared in the `ippsc.h` file. The function computes filtered excitation using following formula:

$$pSrcDstPastExc[i] = SrcDstPastExc[i-1] + valExc * pSrcImpulseResponse[i],$$

$$pSrcDstPastExc[0] = valExc,$$

where  $i = len - 1, \dots, 1$ .

### Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

`ippStsSizeErr` Indicates an error when `len` is less than or equal to zero.

## G.729.1 Functions

This section describes the Intel IPP functions that can be used in implementing speech codecs following ITU-T recommendations G.729.1 (ex.G729EV) [ITU7291]. The primitives are primarily designed to implement the well-defined, computationally expensive core operations that comprise

the codec portion of the G.729.1 system. The G.729.1 codec comprises an algorithm extending ITU-T G.729 [ITU729] for the scalable coding of narrowband and wideband speech and audio signals at 8-32 kbit/s. The output of the G.729.1 codec has a bandwidth of 50-4000 Hz at 8 and 12 kbit/s and 50-7000 Hz from 14 to 32 kbit/s. At 8 kbit/s, G.729.1 is fully interoperable with G.729 [ITU729], G.729 Annex A [ITU729A] and G.729 Annex B [ITU729B].

The list of these functions is given in Table 9-3.

**Table 9-3 Intel IPP G.729 Functions**

Function Base Name	Operation
<code>FilterHighpassGetStateSize_G7291</code>	Returns the size of the high-pass filter state memory.
<code>FilterHighpassInit_G7291</code>	Returns the size of the high-pass filter state memory.
<code>FilterHighpass_G7291</code>	Performs high-pass filtering.
<code>FilterLowpass_G7291</code>	Performs high-pass filtering.
<code>QMFGetStateSize_G7291</code>	Returns the size of the QMF encode state memory.
<code>QMFInit_G7291</code>	Initializes the state memory of QMF filter.
<code>QMFEncode_G7291</code>	Performs QMF analysis filtering.
<code>QMFDecode_G7291</code>	Performs QMF synthesis filtering.
<code>LSFDecode_G7291</code>	Decodes the quantized LSFs.
<code>AdaptiveCodebookSearch_G7291</code>	Searches for the integer delay and the fraction delay, and computes the adaptive vector.
<code>AdaptiveCodebookGain_G7291</code>	Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.
<code>AlgebraicCodebookSearchL1_G7291, AlgebraicCodebookSearchL2_G7291</code>	Performs the fixed (algebraic) codebook search.
<code>GainQuant_G7291</code>	Quantizes the codebook gain using a two-stage conjugate-structured codebook.
<code>EnvelopTime_G7291</code>	Computes the time envelope parameters.
<code>EnvelopFrequency_G7291</code>	Computes the frequency envelope parameters.
<code>GenerateExcitationGetStateSize_G7291</code>	Calculates the size of the state memory for generation of the excitation signal.
<code>GenerateExcitationInit_G7291</code>	Initializes the state memory for generation of the excitation signal.
<code>GenerateExcitation_G7291</code>	Performs generation of the excitation signal.
<code>ShapeEnvelopTime_G7291</code>	Performs the shaping of the time envelope of the excitation signal.
<code>ShapeEnvelopFrequency_G7291</code>	Performs the shaping of the frequency envelope of the excitation signal.
<code>CompressEnvelopTime_G7291</code>	Performs an adaptive amplitude compression.
<code>MDCTFwd_G7291</code>	Computes forward MDCT.

Function Base Name	Operation
<a href="#">MDCTInv_G7291</a>	Computes inverse MDCT
<a href="#">MDCTQuantFwd_G7291</a>	Performs split spherical vector quantization.
<a href="#">MDCTQuantInv_G7291</a>	Performs inverse split spherical vector quantization.
<a href="#">MDCTPostProcess_G7291</a>	Performs post-processing oh higher-band MDCT coefficients.
<a href="#">GainControl_G7291</a>	Compensates for the tilt in the short-term filter.
<a href="#">TiltCompensation_G7291</a>	Compensates for the tilt in the short-term filter.
<a href="#">QuantParam_G7291</a>	Quantizes of the TDBWE parameter set by "mean removed VQ".

## FilterHighpassGetStateSize\_G7291

*Returns the size of the high-pass filter state memory.*

### Syntax

```
IppStatus ippFilterHighpassGetStateSize_G7291_16s(int* pSize);
```

### Parameters

*pSize*                                      Pointer to the computed value of the memory size.

### Description

The function `ippFilterHighpassGetStateSize_G7291` is declared in the `ippsc.h` file. This function computes the size of the high-pass filter state structure and stores the result in *pSize*.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error when the *pSize* pointer is NULL.

## FilterHighpassInit\_G7291

*Initializes the state memory of the high-pass filter.*

### Syntax

```
IppStatus ippFilterHighpassInit_G7291_16s (IppsFilterHighpassState_G7291_16s*  
pState, const Ipp16s pCoeffs[6]);
```

## Parameters

<i>pState</i>	Pointer to memory supplied for filtering.
<i>pCoeffs</i>	Pointer to the source vector of the filter coefficients.

## Description

The function `ippsFilterHighpassInit_G7291` is declared in the `ippsc.h` file. This function initializes the state structure *pState* for the high-pass filter. The size of this structure must be computed previously by calling the functions `ippsFilterHighpassGetStateSize_G7291`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pCoeffs</i> or <i>pState</i> pointer is NULL.

# FilterHighpass\_G7291

*Performs high-pass filtering.*

## Syntax

```
IppStatus ippsFilterHighpass_G7291_16s_ISfs(Ipp16s* pSrcDst, int len,
IppsFilterHighpassState_G7291_16s* pState, int scaleFactor);
```

## Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>len</i>	Length of the <i>pSrcDst</i> .
<i>pState</i>	Pointer to the previously allocated state structure.
<i>scaleFactor</i>	Scaling factor to apply to the result.

## Description

The function `ippsFilterHighpass_G7291` is declared in the `ippsc.h` file. The lower-band signal is filtered by an elliptic high-pass filter  $H_{h1}(z)$  of order 2 to remove the frequency components below 50 Hz. The filter is defined as follows:

$$H_{h1}(z) = \frac{b_{j0} + b_{j1}z^{-1} + b_{j2}z^{-2}}{1 + a_{j1}z^{-1} + a_{j2}z^{-2}}$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> or <i>pState</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## FilterLowpass\_G7291

*Performs low-pass filtering.*

---

### Syntax

```
IppStatus ippsFilterLowpass_G7291_16s_I(Ipp16s* pSrcDst, int len, Ipp16s
pMem[12], IppHintAlgorithm hint);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector.
<i>len</i>	Length of the <i>pSrcDst</i> .
<i>pMem</i>	Pointer to the memory supplied for filtering.
<i>hint</i>	Hint for using specific code.

### Description

The function `ippsFilterLowPass_G7291` is declared in the `ippsc.h` file. The higher-band signal is filtered by an elliptic low-pass filter  $H_{h2}(z)$  of order 4 to remove the frequency components in the 3-4 kHz. The filter is defined as follows:

$$H_{h2}(z) = \frac{b_{j0} + b_{j1}z^{-1} + b_{j2}z^{-2} + b_{j3}z^{-3} + b_{j4}z^{-4}}{1 + a_{j1}z^{-1} + a_{j2}z^{-2} + a_{j3}z^{-3} + a_{j4}z^{-4}}$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> or <i>pState</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## QMFGgetStateSize\_G7291

Returns the size of the QMF encode state memory.

### Syntax

```
IppStatus ippMQMFGgetStateSize_G7291_16s (int* pSize);
```

### Parameters

*pSize*                                      Pointer to the computed value of the memory size.

### Description

The function `ippMQMFGgetStateSize_G7291` is declared in the `ippsc.h` file. This function compute the size of the external buffer for the QMF encode state structure and store the result in *pSize*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is <code>NULL</code> .

## QMFINit\_G7291

Initializes the state memory of QMF filter.

### Syntax

```
IppStatus ippMQMFINit_G7291_16s(IppsQMFGState_G7291_16s* pState);
```

### Parameters

*pState*                                      Pointer to the memory supplied for the QMF encode.

## Description

The function `ippsQMFFinit_G7291` is declared in the `ippsc.h` file. This function initializes the state structure `pState` of the QMF filter. The size of this structure must be computed previously by calling the functions `ippsQMFFGetStateSize_G7291`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is <code>NULL</code> .

## QMFEncode\_G7291

*Performs QMF analysis filtering.*

---

## Syntax

```
IppStatus ippsQMFEncode_G7291_16s(const Ipp16s* pSrc, int len, Ipp16s*
pDstLowBand, Ipp16s* pDstHighBand, IppsQMFFState_G7291_16s* pState);
```

## Parameters

<code>pSrc</code>	Pointer to the input vector [ <code>len</code> ].
<code>len</code>	Number of samples.
<code>pDstLowBand</code>	Pointer to the output lower-band vector [ <code>len/2</code> ].
<code>pDstHighBand</code>	Pointer to the output higher-band vector [ <code>len/2</code> ].
<code>pState</code>	Pointer to the memory supplied for QMF encode.

## Description

The function `ippsQMFEncode_G7291` is declared in the `ippsc.h` file. This function is used to split the input signal sampled at 16000 Hz into two signals sampled at 8000 Hz: a lower band signal and a higher band signal using Quadrature Mirror Filter (QMF) analysis filterbank. The description of the QMF analysis algorithm can be found in [ITU7291] section 6.1.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .



`ippStsSizeErr` Indicates an error when *len* is less than or equal to 0, or greater than 320.

## QMFDecode\_G7291

*Performs QMF synthesis filtering.*

### Syntax

```
IppStatus ippSQMFDecode_G7291_16s (const Ipp16s* pSrcLowBand, const Ipp16s*
pSrcHighBand, Ipp16s valGainSwitching, int len, Ipp16s* pDst,
IppsQMFState_G7291_16s* pState);
```

### Parameters

<i>pSrcLowBand</i>	Pointer to the input lower-band vector [ <i>len</i> ].
<i>pSrcHighBand</i>	Pointer to the input higher-band vector [ <i>len</i> ].
<i>valGainSwitching</i>	Gain attenuation factor.
<i>len</i>	Number of samples.
<i>pDst</i>	Pointer to the output vector [ <i>len</i> *2].
<i>pState</i>	Pointer to the memory supplied for QMF decode.

### Description

The function `ippSQMFDecode_G7291` is declared in the `ippsc.h` file. This function is used to obtain the output signal sampled at 16000 Hz from two input signals sampled at 8000 Hz: a lower band signal and a higher band signal. Note that the QMF synthesis filterbank performs an additional signal upscaling (by 2) to compensate the signal downscaling (by 2) done through the encoding. The description of the QMF synthesis algorithm can be found in [\[ITU7291\]](#) section 7.5.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or greater than 320.

## LSFDecode\_G7291

*Decodes the quantized LSFs.*

---

### Syntax

```
IppStatus ippLSFDecode_G7291_16s (const Ipp16s pQuantIndex[4], Ipp16s
pSrcDstPrevLSF[4][10], Ipp16s pSrcDstLSF[10]);
```

### Parameters

<i>pQuantIndex</i>	Pointer to the input vector of codebook indexes.
<i>pQuantIndex</i>	Pointer to the input/output table of 4 previously quantized LSFs vectors.
<i>pSrcDstLSF</i>	Pointer to the quantized LSF input/output vector.

### Description

The function `ippLSFDecode_G7291` is declared in the `ippsc.h` file. This function retrieves the quantized LSF coefficients from the quantization table and indexes.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLSFLow</code>	Indicates a warning of low stability of LSF.
<code>ippStsLSFHigh</code>	Indicates a warning of high stability of LSF.

## AdaptiveCodebookSearch\_G7291

*Searches for the integer delay and the fraction delay, and computes the adaptive vector.*

---

### Syntax

```
IppStatus ippAdaptiveCodebookSearch_G7291_16s(const Ipp16s
pSrcAdptTarget[40], const Ipp16s pSrcImpulseResponse[40], const Ipp16s
pSrcPrevExcitation[154], Ipp16s pDstDelay[2], Ipp16s lagMin, Ipp16s lagMax,
int subFrame);
```

## Parameters

<i>pSrcAdptTarget</i>	Pointer to the target signal for the adaptive-codebook search vector.
<i>pSrcImpulseResponse</i>	Pointer to the impulse response of the weighted synthesis filter vector.
<i>pSrcPrevExcitation</i>	Pointer to the previous and updated excitation vector.
<i>pDstDelay</i>	Pointer to the output integer and fraction delays vector.
<i>lagMin</i>	Minimum value of the searching range.
<i>lagMax</i>	Maximum value of the searching range.
<i>subFrame</i>	Subframe number, either 0 or 1.

## Description

The function `ippsAdaptiveCodebookSearch_G7291` are declared in the `ippsc.h` file. It searches for the integer delay  $T$  and the fraction delay  $t$ , using the target signal and the past filtered excitation, and computes the adaptive vector. The search is performed in the subframes *subFrame*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>subFrame</i> is not equal to 0 or 1.
<code>ippStsRangeErr</code>	Indicates an error when <i>lagMin</i> is less than 20, or greater than 143 or <i>lagMax</i> .
<code>ippStsRangeErr</code>	Indicates an error when <i>lagMax</i> is less than 20, or greater than 143 or <i>lagMin</i> .

## AdaptiveCodebookGain\_G7291

*Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.*

---

### Syntax

```
IppStatus ippAdaptiveCodebookGain_G7291_16s(const Ipp16s pSrcAdptTarget[40],
const Ipp16s pSrcImpulseResponse[40], const Ipp16s pSrcAdptVector[40], Ipp16s
pDstFltAdptVector[40], Ipp16s* pResultAdptGain);
```

### Parameters

<i>pSrcAdptTarget</i>	Pointer to the adaptive target signal vector.
<i>pSrcImpulseResponse</i>	Pointer to the impulse response of the perceptual weighting filter vector.
<i>pSrcAdptVector</i>	Pointer to the adaptive-codebook vector.
<i>pDstFltAdptVector</i>	Pointer to the output filtered adaptive-codebook vector.
<i>pResultAdptGain</i>	Pointer to the output adaptive-codebook gain value.

### Description

The function `ippAdaptiveCodebookGain_G7291` is declared in the `ippsc.h` file. It computes the gain of the adaptive-codebook vector, calculates the filtered adaptive-codebook vector and coefficients needed for the gain quantization, respectively. This function are applied in subframes.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## AlgebraicCodebookSearchL1\_G7291, AlgebraicCodebookSearchL2\_G7291

Performs the fixed (algebraic) codebook search.

### Syntax

```
IppStatus ippsAlgebraicCodebookSearchL1_G7291_16s(const Ipp16s
pSrcFixedTarget[40], const Ipp16s pSrcLtpResidual[40], Ipp16s
pSrcDstFltAdptVector[40], Ipp16s pSrcDstImpulseResponse[40], Ipp16s pitchLag,
Ipp16s pitchGain, Ipp16s pDstFixedVector[40], Ipp16s pDstFltFixedVector[40],
Ipp16s pDstCodebookIdx[2]);
```

```
IppStatus ippsAlgebraicCodebookSearchL2_G7291_16s(const Ipp16s
pSrcFixedTarget[40], const Ipp16s pSrcLtpResidual[40], Ipp16s
pSrcDstImpulseResponse[40], Ipp16s voiceFactor, Ipp16s pDstFixedVector[40],
Ipp16s pDstFltFixedVector[40], Ipp16s pDstCodebookIdx[2]);
```

### Parameters

<i>pSrcFixedTarget</i>	Pointer to the input updated target speech vector.
<i>pSrcLtpResidual</i>	Pointer to the input residual vector after long term prediction.
<i>pSrcDstFltAdptFixedVector</i>	Pointer to the output filtered adaptive fixed-codebook vector.
<i>pSrcDstImpulseResponse</i>	Pointer to the output LP synthesis filter.
<i>pitchLag</i>	Value of the pitch lag.
<i>pitchGain</i>	Gain value of the last quantized pitch.
<i>pDstFixedVector</i>	Pointer to the fixed-codebook vector.
<i>pDstFltFixedVector</i>	Pointer to the output filtered fixed-codebook vector.
<i>pDstCodebookIndex</i>	Pointer to the fixed-codebook index.
<i>voiceFactor</i>	Voicing factor value.

### Description

The functions `ippsAlgebraicCodebookSearchL1_G7291` and `ippsAlgebraicCodebookSearchL2_G7291` are declared in the `ippsc.h` file. The `ippsAlgebraicCodebookSearchL1_G7291` function searches the fixed-codebook vector and index at 8 kbps rate similar to the

G.729 fixed-codebook search. The difference is that for some parts this function uses the orthogonal search procedure and global pulse replacement method described in [ITU7291] section 6.3.7.

The `ippsAlgebraicCodebookSearchL2_G7291` function searches the fixed-codebook vector and index at 12 kbps. The 12 kbit/s enhancement layer relies on the extra fixed codebook; that, when combined with the 8 kbit/s core layer excitation, offers a richer excitation signal. In particular this extra fixed codebook better represents high frequencies of the LP excitation. This benefit is increased by the use of a modified perceptual filter to stress high frequencies. See more details in [ITU7291] section 6.3.9.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>pitchLag</i> is less than 19 or greater than 144.

## GainQuant\_G7291

*Quantizes the codebook gain using a two-stage conjugate-structured codebook.*

---

### Syntax

```
IppStatus ippsGainQuant_G7291_16s(const Ipp16s pSrcFixedTarget[40], const
Ipp16s pSrcFltFixedVector[40], Ipp16s valSrcQGain, Ipp16s* pDstQGainIndex,
Ipp16s* pDstQGain, int subFrame);
```

### Parameters

*pSrcFixedTarget* Pointer to the target signal for the adaptive codebook search vector.

*pSrcFltFixedVector* Pointer to the filtered fixed codebook vector.

*valSrcQGain* Quantized gain of the first stage.

*pDstQGainIndex* Pointer to the output gain codebook index.

*pDstQGain* Pointer to the output quantized gain.

*subFrame* Subframe number in range [0, 3]

### Description

The function `ippsGainQuant_G7291` are declared in the `ippsc.h` file. This function quantizes the gain using 7-bit codebook. It is applied in subframes. Procedure of the quantization of the 8 kbits/s gains is similar to the corresponding procedure in the G.729 codec.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>subFrame</i> is less than 0 or greater than 3.

## EnvelopTime\_G7291

*Computes the time envelope parameters.*

---

### Syntax

```
IppStatus ippsEnvelopTime_G7291_16s(const Ipp16s* pSrc, Ipp16s nls, Ipp16s* pDstEnvelopTime, int subFrame);
```

### Parameters

<i>pSrc</i>	Pointer to the original highband input signal [10* <i>subFrame</i> ].
<i>nls</i>	Normalization of the input signal.
<i>pDstEnvelopTime</i>	Pointer to the output time envelope parameters [ <i>subFrame</i> ].
<i>subFrame</i>	Number of consecutive subframes in <i>pSrc</i> .

### Description

The function `ippsEnvelopTime_G7291` is declared in the `ippsc.h` file. This function calculates the time envelope parameters as logarithmic subframe energies. See [ITU7291] section 6.5.1.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

`ippStsRangeErr` Indicates an error when `subFrame` is less than 0.

## EnvelopFrequency\_G7291

*Computes the frequency envelope parameters.*

---

### Syntax

```
IppStatus ippEnvelopFrequency_G7291_16s(const Ipp16s pSrc[64], Ipp16s
pDstEnvelopFreq[12]);
```

### Parameters

`pSrc` Pointer to the original highband input signal.  
`pDstEnvelopFreq` Pointer to the output frequency envelope parameters.

### Description

The function `ippEnvelopFrequency_G7291` is declared in the `ippsc.h` file. This function calculates the frequency envelope parameters. To calculate 12 frequency envelope parameters, the `pSrc` signal is windowed and transformed by FFT of length 64. Finally, the frequency envelope parameters are calculated as logarithmic weighted subband energies for 12 evenly spaced and wide overlapping subbands in the FFT domain. See [ITU7291] section 6.5.2.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## GenerateExcitationGetStateSize\_G7291

*Calculates the size of the state memory for generation of the excitation signal.*

---

### Syntax

```
IppStatus ippGenerateExcitationGetStateSize_G7291_16s(int* pSize);
```

### Parameters

`pSize` Pointer to the output value of the memory size.



### Description

The function `ippsGenerateExcitationGetStateSize_G7291` are declared in the `ippsc.h` file. This function computes the size of the external buffer for state of generation of the excitation signal and store the result in `pSize`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSize</code> pointer is <code>NULL</code> .

## GenerateExcitationInit\_G7291

*Initializes the state memory for generation of the excitation signal.*

---

### Syntax

```
IppStatus
ippsGenerateExcitationInit_G7291_16s(IppsGenerateExcitationState_G7291_16s*
pState);
```

### Parameters

<code>pState</code>	Pointer to the memory supplied for generation of the excitation signal.
---------------------	---

### Description

The function `ippsGenerateExcitationInit_G7291` are declared in the `ippsc.h` file. This function initializes the state structure `pState` for generation of the excitation signal. The size of this structure must be computed previously by calling the function `ippsGenerateExcitationGetStateSize_G7291`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is <code>NULL</code> .

## GenerateExcitation\_G7291

*Performs generation of the excitation signal.*

---

### Syntax

```
IppStatus ippsGenerateExcitation_G7291_16s(const Ipp16s pIntPitchLag[4],
const Ipp16s pFracPitchLag[4], const Ipp32s pLtpPower[4], const Ipp32s
pFixPower[4], Ipp16s pDst[80], IppsGenerateExcitationState_G7291_16s* pState);
```

### Parameters

<i>pIntPitchLag</i>	Pointer to the integer part of pitch lag.
<i>pFracPitchLag</i>	Pointer to the fractional part of pitch lag.
<i>pLtpPower</i>	Pointer to the ltp power ratio.
<i>pFixPower</i>	Pointer to the fix power ratio.
<i>pDst</i>	Pointer to the output vector.
<i>pState</i>	Pointer to the memory supplied for generation of the excitation signal.

### Description

The function `ippsGenerateExcitation_G7291` are declared in the `ippsc.h` file. This function generates TDBWE excitation according to [\[ITU7291\]](#) section 7.2.2.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## ShapeEnvelopTime\_G7291

*Performs the shaping of the time envelope of the excitation signal.*

---

### Syntax

```
IppStatus ippsShapeEnvelopTime_G7291_16s(const Ipp16s pSrc[80], const Ipp16s
pSrcEnvelopTime[8], Ipp16s* pSrcDstGain, Ipp16s* pSrcDstNorm, Ipp16s
pDst[208]);
```

## Parameters

<i>pSrc</i>	Pointer to the excitation input signal.
<i>pSrcEnvelopeTime</i>	Pointer to the desired time envelope.
<i>pSrcDstGain</i>	Pointer to the input/output gain of the time envelope shaping.
<i>pSrcDstNorm</i>	Pointer to the input/output norm of the time envelope shaping.
<i>pDst</i>	Pointer to the shaped output signal.

## Description

The function `ippsShapeEnvelopeTime_G7291` is declared in the `ippsc.h` file. The shaping of the time envelope of the excitation signal utilizes the decode time envelope parameters to obtain a signal with a time envelope which is near-identical to the time envelope of the encoder side higher-band signal. See [ITU7291] section 7.2.3.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## ShapeEnvelopeFrequency\_G7291

*Performs the shaping of the frequency envelope of the excitation signal.*

---

## Syntax

```
IpplStatus ippsShapeEnvelopeFrequency_G7291_16s(const Ippl6s pSrc[208], const
Ippl6s pSrcEnvelopeFreq[12], Ippl6s pDst[80], Ippl6s pDstFilterCoeffs[33],
Ippl6s pMem[32]);
```

## Parameters

<i>pSrc</i>	Pointer to the excitation input signal.
<i>pSrcEnvelopeFreq</i>	Pointer to the input desired frequency envelope.
<i>pDst</i>	Pointer to the shaped output signal.
<i>pDstFilterCoeffs</i>	Pointer to the filtering coefficients of the frequency envelope shaping.

*pMem* Pointer to the memory supplied for frequency envelope parameters.

### Description

The function `ippsShapeEnvelopFrequency_G7291` is declared in the `ippsc.h` file. The shaping of the frequency envelope of the excitation signal utilizes the decode frequency envelope parameters to obtain a signal with a frequency envelope which is near-identical to the frequency envelope of the encoder side higher-band signal. See [ITU7291] section 7.2.4.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## CompressEnvelopTime\_G7291

*Performs an adaptive amplitude compression.*

---

### Syntax

```
IppStatus ippsCompressEnvelopTime_G7291_16s(const Ipp16s pSrcEnvelopTime[28],
Ipp16s pSrcDst[160], Ipp16s pMem[2]);
```

### Parameters

*pSrcEnvelopTime* Pointer to the desired time envelope .  
*pSrcDst* Pointer to the shaped input/output signal.  
*pMem* Pointer to the input/output memory for 2 past time envelope parameters.

### Description

The function `ippsCompressEnvelopTime_G7291` is declared in the `ippsc.h` file. The function shapes the excitation signal in time domain and performs adaptive amplitude compression in order to attenuate large deviations from this envelope according to [ITU7291] section 7.2.5.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## MDCTFwd\_G7291

*Computes forward MDCT.*

---

### Syntax

```
IppStatus ippsMDCTFwd_G7291_16s(const Ipp16s pSrc[160], Ipp16s pSrcDst[160],
Ipp16s* pSrcDstNorm, Ipp16s pDstMDCTCoeffs[160], IppHintAlgorithm hint);
```

### Parameters

<i>pSrc</i>	Pointer to the input signal.
<i>pSrcDst</i>	Pointer to the old input/output signal.
<i>pDstNorm</i>	Pointer to the output normalization factor.
<i>pDstMDCTCoeffs</i>	Pointer to the output MDCT coefficients.
<i>hint</i>	Suggests using specific code

### Description

The function `ippsMDCTFwd_G7291` is declared in the `ippsc.h` file. The function computes forward MDCT and returns used normalization factor. See [ITU7291] section 6.6.3.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## MDCTInv\_G7291

*Computes inverse MDCT.*

---

### Syntax

```
IppStatus ippsMDCTInv_G7291_16s(const Ipp16s pMDCTCoeffs[160], Ipp16s
pSrcDstMDCTPrevCoeffs[160], Ipp16s pDst[160], Ipp16s scaleFactor);
```

### Parameters

<i>pMDCTCoeffs</i>	Pointer to the MDCT coefficients.
<i>pSrcDstMDCTPrevCoeffs</i>	Pointer to the old input/output MDCT memory.

<i>pDst</i>	Pointer to the output signal.
<i>scaleFactor</i>	Scale factor for the normalization MDCT coefficients.

### Description

The function `ippMDCTInv_G7291` are declared in the `ippsc.h` file. The function computes inverse MDCT using given normalization factor. See [ITU7291] section 7.3.8.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the input or output pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>scaleFactor</i> is less than 0.

## MDCTQuantFwd\_G7291

*Performs split spherical vector quantization.*

---

### Syntax

```
IppStatus ippMDCTQuantFwd_G7291_16s32u(const Ipp16s pMDCTCoeffs[320], const
Ipp16s pBitsNumber[18], Ipp32u pDst[18]);
```

### Parameters

<i>pMDCTCoeffs</i>	Pointer to the MDCT coefficients.
<i>pBitsNumber</i>	Pointer to the input number of bits allocated per subbands.
<i>pDst</i>	Pointer to the output vector quantization.

### Description

The function `ippMDCTQuantFwd_G7291` is declared in the `ippsc.h` file. This function perform spherical vector quantization. This operation id divided into two steps: search for the best codevector and indexing of the selected codevector. See [ITU7291] section 6.6.9.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is NULL.

## MDCTQuantInv\_G7291

*Performs inverse split spherical vector quantization.*

### Syntax

```
IppStatus ippsMDCTQuantInv_G7291_32u16s(const Ipp32u pSrc[18], const Ipp16s
pBitsNumber[18], const Ipp16s pQuantSpecEnv[18], Ipp16s pDstMDCTCoeffs[320]);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector quantization indices.
<i>pBitsNumber</i>	Pointer to the input number of bits allocated per subbands.
<i>pQuantSpecEnv</i>	Pointer to the input quantized spectrum envelope.
<i>pDstMDCTCoeffs</i>	Pointer to the output MDCT coefficients.

### Description

The function `ippsMDCTQuantInv_G7291` is declared in the `ippsc.h` file. This function decoding of MDCT vector quantization indices. See [ITU7291] section 7.3.5.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## MDCTPostProcess\_G7291

*Performs post-processing oh higher-band MDCT coefficients.*

### Syntax

```
IppStatus ippsMDCTPostProcess_G7291_16s(Ipp16s pSrcDstMDCTCoeffs[160], int
numBits);
```

### Parameters

<i>pSrcDstMDCTCoeffs</i>	Pointer to the input/output MDCT coefficients.
<i>numBits</i>	Number of bits to decode.

## Description

The function `ippsMDCTPostProcess_G7291` are declared in the `ippsc.h` file. This function performs MDCT short-term and long-term post-processing in high band. See [ITU7291] section 7.3.7.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrcDstMDCTCoeffs</code> pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>numBits</code> is less than 160 or greater than 640.

## GainControl\_G7291

*Calculates adaptive gain control.*

---

### Syntax

```
IppStatus ippsGainControl_G7291_16s_I(const Ipp16s pSrc[40], Ipp16s
pSrcDst[40], Ipp16s* pSrcDstGain, Ipp16s gain, Ipp16s valGainSwitching,
Ipp32s* pSrcDstSmoothLevel);
```

### Parameters

<code>pSrc</code>	Pointer to the source reconstructed speech vector.
<code>pSrcDst</code>	Pointer to the input post-filtered signal and output gain-compensated signal vector.
<code>pSrcDstGain</code>	Pointer to the input/output adaptive gain.
<code>gain</code>	First reflection coefficient.
<code>valGainSwitching</code>	Gain attenuation factor.
<code>pSrcDstSmoothLevel</code>	Pointer to the input/output smooth level.

### Description

The function `ippsGainControl_G7291` is declared in the `ippsc.h` file. This function performs gain control directly derived from the G.729 with some modifications described in [ITU7291] section 7.4.1.



### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## TiltCompensation\_G7291

*Compensates for the tilt in the short-term filter.*

### Syntax

```
IppStatus ippSTiltCompensation_G7291_16s(const Ipp16s pSrc[40], Ipp16s  
pDst[40], Ipp16s gain);
```

### Parameters

<code>pSrc</code>	Pointer to the gamma weighted LP coefficients input vector.
<code>pDst</code>	Pointer to the present and tilt-compensated output speech.
<code>gain</code>	Gain coefficient.

### Description

The function `ippSTiltCompensation_G7291` is declared in the `ippsc.h` file. This function performs FIR filtering to compensate the tilt in the short-term filter. See [ITU7291] section 7.3.10.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## QuantParam\_G7291

Quantizes of the TDBWE parameter set by "mean removed VQ".

### Syntax

```
IppStatus ippsQuantParam_G7291_16s(Ipp16s pSrcDst[28], Ipp16s
pCdbkIndices[4]);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source/destination unquantized parameter set.
<i>pCdbkIndices</i>	Pointer to the codebook indices.

### Description

The function `ippsQuantParam_G7291` is declared in the `ippsc.h` file. This function quantizes of the TDBWE parameter set by "mean removed VQ". For more details see [\[ITU7291\]](#) section 6.5.3.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## G.723.1 Functions

The Intel IPP functions described in this section implement building blocks that can be used to create speech codecs compliant with the ITU-T Recommendation G.723.1 (see [\[ITU723\]](#), [\[ITU723A\]](#)). The list of these functions is given in Table 9-4.

**Table 9-4. Intel IPP G.723.1 Functions**

Function Base Name	Operation
Linear Prediction Analysis Functions	
<a href="#">AutoCorr_G723</a>	Estimates the auto-correlation of a vector.
<a href="#">AutoCorr_NormE_G723</a>	Estimates normal auto-correlation of a vector.
<a href="#">LevinsonDurbin_G723</a>	Calculates the LP coefficients from the autocorrelation coefficients.
<a href="#">LPCToLSF_G723</a>	Converts LP coefficients to LSF coefficients.

Function Base Name	Operation
<a href="#">LSFToLPC_G723</a>	Converts LSF coefficients to the LP coefficients.
<a href="#">LSFDecode_G723</a>	Performs inverse quantization of LSFs.
<a href="#">LSFQuant_G723</a>	Quantizes LSF coefficients.
Codebook Search Functions	
<a href="#">OpenLoopPitchSearch_G723</a>	Searches for an optimal pitch value.
<a href="#">ACELPFixedCodebookSearch_G723</a>	Searches the ACELP fixed codebook for the excitation.
<a href="#">AdaptiveCodebookSearch_G723</a>	Searches for the close loop pitch and the adaptive gain index.
<a href="#">MPMLQFixedCodebookSearch_G723</a>	Searches the MP-MLQ fixed codebook for the excitation.
<a href="#">ToeplizMatrix_G723</a>	Calculates 416 elements of the Toepliz matrix for fixed codebook search.
Gain Quantization Functions	
<a href="#">GainQuant_G723</a>	Implements MP-MLQ gain estimation and quantization.
<a href="#">GainControl_G723</a>	Extracts delayed pitch contribution.
Filter Functions	
<a href="#">HighPassFilter_G723</a>	Performs high-pass filtering of the input signal.
<a href="#">IIR16s_G723</a>	Performs IIR filtering.
<a href="#">SynthesisFilter_G723</a>	Computes the speech signal by filtering the input speech through the synthesis filter $1/A(z)$ .
<a href="#">TiltCompensation_G723</a>	Computes tilt compensation filter.
<a href="#">HarmonicSearch_G723</a>	Searches for the harmonic delay and gain for the harmonic noise shaping filter.
<a href="#">HarmonicNoiseSubtract_G723</a>	Performs harmonic noise shaping.
<a href="#">DecodeAdaptiveVector_G723</a>	Restores the adaptive codebook vector from excitation, pitch, and adaptive gain.
<a href="#">PitchPostFilter_G723</a>	Calculates coefficients of the pitch post filter.

These functions are used in the Intel IPP *G.723.1 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## Linear Prediction Analysis Functions

Functions described in this section implement LSP coding (quantization) and decoding, as well as transformation between LPC, LSP and LSF coefficients.

## AutoCorr\_G723

*Estimates the auto-correlation of a vector.*

---

### Syntax

```
IppStatus ippsAutoCorr_G723_16s(const Ipp16s* pSrcSpch, Ipp16s*
pResultAutoCorrExp, Ipp16s* pDstAutoCorr);
```

### Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal vector [180].
<i>pResultAutoCorrExp</i>	Pointer to the exponent for autocorrelation coefficients.
<i>pDstAutoCorr</i>	Pointer to the autocorrelation coefficients vector [11].

### Description

The function `ippsAutoCorr_G723` is declared in the `ippsc.h` file. This function calculates the first 11 autocorrelation coefficients of the input speech signal. This function is applied to the 180 speech samples centered on the current subframe. The functionality is as follows.

1. First, apply to speech samples the Hamming windows, given by the following formula:

$$w(i) = 0.54 - 0.46 \cos \left[ \frac{(2i-1)}{399} \pi \right] ,$$

$$i = 0, 1, \dots, 179$$

2. Next, calculate the autocorrelations as follows:

$$r(k) = \sum_{i=k}^{179} s(i) \times s(i-k) ,$$

$$k = 0, 1, \dots, 10$$

3. Finally, add the autocorrelation with binomial window, given by:

$$b(0) = 1025/1024,$$

$$b(i) = \exp\left[-0.5\left(\frac{2\pi f_0 i}{f_s}\right)^2\right],$$

$i = 1, \dots, 10$ .



**NOTE.** The function `ippsAutoCorr_G723` is actually a combination of `ippsAutoScale`, `ippsMul_NR` and `ippsAutoCorr_NormE_G723` functions. The following code details the correspondence.

```
{ int autoScale=3, corrScale; short vect[180];
ippsAutoScale_16s(pSrcSpch, Vect,180, &autoScale );
/* Apply the Hamming window */
ippsMul_NR_16s_ISfs(HammWindow,Vect,180,15);
/* Compute the autocorrelation coefficients */
ippsAutoCorr_NormE_G723_16s(Vect,pDstAutoCorr,&corrScale);
*pResultAutoCorrExp = corrScale+(autoScale<<1);
}
```

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`               Indicates an error when one of the specified pointers is `NULL`.

## AutoCorr\_NormE\_G723

*Estimates normal auto-correlation of a vector.*

### Syntax

```
IppStatus ippsAutoCorr_NormE_G723_16s(const Ipp16s* pSrc, Ipp16s* pDst, int*
pNorm);
```

### Parameters

`pSrc`                                Pointer to the source vector [180].  
`pDst`                                Pointer to the destination vector, which stores the  
estimated auto-correlation results of the source vector.  
`pNorm`                               Pointer to the output normalization scale factor.

## Description

The function `ippsAutoCorr_NormE_G723` is declared in the `ippsc.h` file. This function computes 11 auto-correlation coefficients from the input signal. The first correlation coefficient (energy) is multiplied by a white noise correction factor  $b(0) = 1025/1024$ , and the remaining 10 correlation coefficients are multiplied by the binomial window coefficients  $b_n$ ,  $n = 1, \dots, 10$  defined in the reference C-code (see [ITU723]).

The auto-correlation coefficients are additionally multiplied by the factor  $2^{norm0}$ , where  $norm0 \geq 0$  is calculated so as to make the first coefficient (energy) normalized. Thus, the resulting vector  $pDst$  is computed as follows:

$$pDst[n] = b_n \cdot 2^{norm0} \cdot \sum_{i=0}^{179-n} pSrc[i] \cdot pSrc[i+n] , \quad n = 0, \dots, 10$$

$0 \leq n \leq 10$ .

The function `ippsAutoCorr_NormE` stores the normalization scale factor  $norm0$  in  $pNorm$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LevinsonDurbin\_G723

*Calculates the LP coefficients from the autocorrelation coefficients.*

---

### Syntax

```
IpplStatus ippsLevinsonDurbin_G723_16s(const Ippl16s* pSrcAutoCorr, Ippl16s*
pValResultSineDtct, Ippl16s* pResultResidualEnergy, Ippl16s* pDstLPC);
```

### Parameters

<i>pSrcAutoCorr</i>	Pointer to the autocorrelation coefficients vector [11].
<i>pValResultSineDtct</i>	Pointer to the sine detector input/output parameter.
<i>pResultResidualEnergy</i>	Pointer to the output residual energy, in Q15.

*pDstLPC*

Pointer to the output LP coefficients vector [10], in Q13.

### Description

The function `ippsLevinsonDurbin_G723` is declared in the `ippsc.h` file. This function calculates the 10th order LP coefficients from the autocorrelation coefficients using Levinson-Durbin algorithm. This function also performs sine detection.

To obtain LP coefficients  $a_i$ ,  $i = 1, 2, \dots, 10$ , the following set of equations is to be solved:

$$\sum_{i=1}^{10} a_i \times r(|i-k|) = -r(k) ,$$

$k = 1, 2, \dots, 10$ .

The function `ippsLevinsonDurbin_G723` performs the following steps:

1. Levinson-Durbin algorithm is applied to solve the above set of equations. This algorithm uses the recursion detailed in `ippsLevinsonDurbin_G729` function.
2. If the LPC filter used in this algorithm is unstable, that is  $|k_i|$  is very close to 1.0 during recursion, just set the remaining LP coefficients to zero.
3. The sine detector parameter is updated when  $i = 1$  in the recursion, as follows:

$SineDtct = SineDtct \ll 1$ .

If  $k > 0.95$ , then  $SineDtct = SineDtct + 1$ .

### Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## LPCToLSF\_G723

*Converts LP coefficients to LSF coefficients.*

---

### Syntax

```
IppStatus ippsLPCToLSF_G723_16s (const Ipp16s* pSrcLPC, const Ipp16s*  
pSrcPrevLSF, Ipp16s* pDstLSF);
```

### Parameters

<i>pSrcLPC</i>	Pointer to the LPC input vector [10]: $a_1, \dots, a_{10}$ in Q13.
<i>pSrcPrevLSF</i>	Pointer to the previous normalized LSF coefficients vector [10], in Q15.
<i>pDstLSF</i>	Pointer to the normalized LSF coefficients vector [10], in Q15.

### Description

The function `ippsLPCToLSF_G723` is declared in the `ipps.h` file. This function converts a set of 10th order LP coefficients to LSF coefficients by implementing the following operations:

1. Apply the 7.5Hz bandwidth expansion.
2. Calculate the polynomial coefficients of  $F_1(z)$  and  $F_2(z)$ , using the following recursion:

$$f_1(i+1) = a_{i+1} + a_{10-i} - f_1(i+1),$$

$$f_2(i+1) = a_{i+1} + a_{10-i} - f_2(i+1)$$

$$i = 0 \dots 4,$$

$$\text{where } f_1(0) = f_2(0) = 1.0$$

3. Use Chebyshev polynomials to find the roots of  $F_1(z)$  and  $F_2(z)$ , and obtain the LSF coefficients:

$$c_1(\omega) = \cos(5\omega) + f_1(1) \cos(4\omega) + f_1(2) \cos(3\omega) + f_1(3) \cos(2\omega) + f_1(4) \cos(\omega) + f_1(5) / 2$$

$$c_2(\omega) = \cos(5\omega) + f_2(1) \cos(4\omega) + f_2(2) \cos(3\omega) + f_2(3) \cos(2\omega) + f_2(4) \cos(\omega) + f_2(5) / 2$$

4. If all 10 roots needed to determine LSF coefficients are not found, use the set of previous LSF coefficients.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LSFToLPC\_G723

*Converts LSF coefficients to the LP coefficients.*

### Syntax

```
IppStatus ippLSFToLPC_G723_16s(const Ipp16s* pSrcLSF, Ipp16s* pDstLPC);
IppStatus ippLSFToLPC_G723_16s_I (Ipp16s* pSrcLSFDstLPC);
```

### Parameters

<code>pSrcLSF</code>	Pointer to the input LSF coefficients vector [10], in Q15.
<code>pDstLPC</code>	Pointer to the output LP coefficients vector [10], in Q13.
<code>pSrcLSFDstLPC</code>	Pointer to the input LSF and output LP coefficients vector [10], in Q15 and Q13, respectively.

### Description

The function `ippLSFToLPC_G723` is declared in the `ippsc.h` file. This function converts a set of 10th order LSF coefficients to the LP coefficients as follows:

1. Convert LSF coefficients to the LSP coefficients as:  $q_i = \cos(\omega_i)$ ,  $i = 0 \dots 9$
2. Calculate the LP coefficients using the same algorithm as in the function [ippLSPToLPC\\_G729](#) with the only difference in rounding mode required for G.723.1 bit-exact compliance.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LSFDecode\_G723

*Performs inverse quantization of LSFs.*

---

### Syntax

```
IppStatus ippsLSFDecode_G723_16s (const Ipp16s* pQuantIndex, const Ipp16s*
pPrevLSF, int erase, Ipp16s* pQuantLSF);
```

### Parameters

<i>pQuantIndex</i>	Pointer to the input LSP VQ indices vector [3].
<i>pPrevLSF</i>	Pointer to the input quantized LSF for previous subframes.
<i>erase</i>	Indicates frame erasure.
<i>pQuantLSF</i>	Pointer to the output quantized LSF for previous subframes.

### Description

The function `ippsLSFDecode_G723` is declared in the `ippsc.h` file. This function performs inverse quantization of LSFs (in other words, decodes LSP VQ indices). First three VQ table entries corresponding to the transmitted index are found. The predicted vector is added to the found vector and DC vector to form the decoded vectors in three bands separately. A stability check is performed to ensure the difference between them not is less than 31.25 Hz.

For erased frame, zero predicted vectors are chosen and stability check for the difference in 62.5 Hz is applied.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLSFLow</code>	Indicates a warning when a stability condition is not met.

## LSFQuant\_G723

*Quantizes LSF coefficients.*

---

### Syntax

```
IppStatus ippsLSFQuant_G723_16s32s(const Ipp16s* pSrcLSF, const Ipp16s*
pSrcPrevLSF, Ipp32s* pResultQLSFIndex);
```

### Parameters

<i>pSrcLSF</i>	Pointer to the LSF coefficients vector [10], in Q15.
<i>pSrcPrevLSF</i>	Pointer to the previous LSF coefficients vector [10], in Q15.
<i>pResultQLSFIndex</i>	Pointer to the combined index of quantized LSF coefficients. The combination is built by left-shifting the first codebook index by 16 bits and left-shifting the second codebook index by 8 bits, and then adding together the two shifted indices and the third codebook index.

### Description

The function `ippsLSFQuant_G723` is declared in the `ippsc.h` file. This function quantizes the LSF coefficients to obtain the codebook indices using PSVQ. It implements the following operations:

1. Calculate the diagonal weighting matrix  $w$ , determined from the unquantized LSF coefficients, as:

$$w_{1,1} = 1/(\omega_2 - \omega_1),$$

$$w_{10,10} = 1/(\omega_{10} - \omega_9),$$

$$w_{j,j} = 1/\min(\omega_j - \omega_{j-1}, \omega_{j+1} - \omega_j), j = 2 \dots 9$$

2. Calculate the prediction LSF coefficients as given by:

$$\omega_p = (\omega - \omega_{DC}) - 0.375(\omega_{-1} - \omega_{DC})$$

where  $\omega$  are the current LSF coefficients,  $\omega_{-1}$  are the previous LSF coefficients, and  $\omega_{DC}$  are the DC LSF coefficients.

3. Search the codebook vector to minimize the following error:

$$E_l = (\omega_p - \omega_l)^T W (\omega_p - \omega_l) ,$$

where  $\omega_l$  is the  $l$ -th code vector in the codebook. The quantization uses 3-3-4 split.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Codebook Search Functions

These functions perform open loop pitch estimation using the adaptive codebook, and search in ACELP excitation (fixed) codebook for optimal signs and positions of the pulses.

### OpenLoopPitchSearch\_G723

*Searches for an optimal pitch value.*

---

#### Syntax

```
IppStatus ippOpenLoopPitchSearch_G723_16s(const Ipp16s* pSrcWgtSpch, Ipp16s*
pResultOpenDelay);
```

#### Parameters

<i>pSrcWgtSpch</i>	Pointer to the perceptually weighted speech signal. The signal length is 265, and the pointer <i>pSrcWgtSpch</i> points to its 146th element, in Q12.
<i>pResultOpenDelay</i>	Pointer to the open-loop pitch search result.

#### Description

The function `ippOpenLoopPitchSearch_G723` is declared in the `ippsc.h` file. This function extracts the open loop pitch from the weighted speech signal. It is applied in half-frames as follows:

1. The open loop pitch is chosen such that the cross-correlation is maximized:

$$c_{o1}(j) = \frac{\left[ \sum_{i=0}^{119} s(i)s(i-j) \right]^2}{\sum_{i=0}^{119} s(i-j)s(i-j)}$$

$j = 18, \dots, 142$

2. During the search, preference is given to smaller pitch periods to avoid choosing pitch multiples. Every found pitch value is compared with the previous best. If the difference is less than 18 and  $c_{o1}(j) > c_{o1}(j')$ , the process is over. Otherwise, the pitch is chosen only if  $c_{o1}(j)$  is greater than  $c_{o1}(j')$  by 1.25db.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`               Indicates an error when one of the specified pointers is `NULL`.

## ACELPFixedCodebookSearch\_G723

*Searches the ACELP fixed codebook for the excitation.*

### Syntax

```
IppStatus ippACELPFixedCodebookSearch_G723_16s(const Ipp16s* pSrcFixedCorr,
const Ipp16s* pSrcMatrix, Ipp16s* pDstFixedSign, Ipp16s* pDstFixedPosition,
Ipp16s* pResultGrid, Ipp16s* pDstFixedVector, Ipp16s* pSearchTimes);

IppStatus ippACELPFixedCodebookSearch_G723_32s16s(const Ipp16s*
pSrcFixedCorr, Ipp32s* pSrcDstMatrix, Ipp16s* pDstFixedSign, Ipp16s*
pDstFixedPosition, Ipp16s* pResultGrid, Ipp16s* pDstFixedVector, Ipp16s*
pSearchTimes);
```

### Parameters

`pSrcFixedCorr`                      Pointer to the correlation between residual and impulse response vector [60]

<i>pSrcMatrix</i>	Pointer to the elements of Toepliz matrix [416].
<i>pSrcDstMatrix</i>	Pointer to the input and output elements of Toepliz matrix [416].
<i>pDstFixedSign</i>	Pointer to the signs of the fixed vector [4].
<i>pDstFixedPosition</i>	Pointer to the positions of the fixed vector [4].
<i>pResultGrid</i>	Pointer to the beginning grid location.
<i>pDstFixedVector</i>	Pointer to the fixed vector [60].
<i>pSearchTimes</i>	Pointer to the input and output maximum search time

## Description

The function `ippsACELPFixedCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches the ACELP fixed codebook for the excitation in the 5.3Kbps encoder. It is applied in subframes as follows:

1. There are four non-zero pulses in the fixed vector as given by:

$$c(n) = \sum_{k=0}^3 \alpha_k \delta(n - m_k) \quad , \quad -$$

$$n = 0, \dots, 59$$

where  $\psi_k$  and  $m_k$ ,  $k = 0..3$  are the signs and positions of the fixed vector respectively.

2. Search for the parameters  $\psi_k$  and  $m_k$ ,  $k = 0..3$ , that minimize the following error:

$$\begin{aligned} \delta = & \Phi(m_0, m_0) + \Phi(m_1, m_1) + 2\alpha_0\alpha_1\Phi(m_0, m_1) + \\ & \Phi(m_2, m_2) + 2[\alpha_0\alpha_2\Phi(m_0, m_2) + \alpha_1\alpha_2\Phi(m_1, m_2)] + \\ & \Phi(m_3, m_3) + 2[\alpha_0\alpha_3\Phi(m_0, m_3) + \alpha_1\alpha_3\Phi(m_1, m_3) + \alpha_2\alpha_3\Phi(m_2, m_3)] \end{aligned}$$

where  $\Phi(i, j)$  is the Toepliz matrix.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## AdaptiveCodebookSearch\_G723

*Searches for the close loop pitch and the adaptive gain index.*

---

### Syntax

```
IppStatus ippAdaptiveCodebookSearch_G723(Ipp16s valBaseDelay, const Ipp16s*
pSrcAdptTarget, const Ipp16s* pSrcImpulseResponse, Ipp16s*
pSrcPrevExcitation, const Ipp32s* pSrcPrevError, Ipp16s* pResultCloseLag,
Ipp16s* pResultAdptGainIndex, Ipp16s subFrame, Ipp16s sineDtct,
IppSpchBitRate bitRate);
```

### Parameters

<code>valBaseDelay</code>	Base delay, in the range [18,145].
<code>pSrcAdptTarget</code>	Pointer to the adaptive target signal vector [60].
<code>pSrcImpulseResponse</code>	Pointer to the impulse response vector [60].
<code>pSrcPrevExcitation</code>	Pointer to the previous excitation vector [145].
<code>pSrcPrevError</code>	Pointer to the previous error vector [5].
<code>subFrame</code>	Subframe number, from 0 to 3.
<code>bitRate</code>	Transmit bit rate, equal to either <code>IPP_SPCHBR_6300</code> or <code>IPP_SPCHBR_5300</code> .
<code>sineDtct</code>	Sine detector parameter.
<code>pResultCloseLag</code>	Pointer to the lag of close pitch.
<code>pResultAdptGainIndex</code>	Pointer to the index of the adaptive gain.

### Description

The function `ippAdaptiveCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches for the close loop pitch and the adaptive gain index. It is applied in subframes as follows:

1. For subframe 0 and 2, the close loop pitch lag is chosen in the range of [-1,1] around the appropriate open loop pitch. For subframe 1 and 3, the range is [-1, 3] around the open loop pitch.
2. Denote the close loop pitch as  $L_i$ ,  $i = 0...3$ . The pitch predictor gains are vector quantized. For 6.3Kbps bit rate, two codebooks are used with 85 entries and 170 entries, respectively. If  $L_0$  is less than 58 for subframe 0 and 1, or if  $L_2$  is less than 58 for subframe 2 and 3, then the 85-entry codebook is used for pitch gain quantization. Otherwise, the 170-entry codebook is used. For 5.3Kbps bit rate, the quantization is against a 170-entry codebook.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>valBaseDelay</code> is not in the range [18, 145], or <code>subFrame</code> is not in the range [0, 3], or <code>bitRate</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## MPMLQFixedCodebookSearch\_G723

*Searches the MP-MLQ fixed codebook for the excitation.*

---

### Syntax

```
ppStatus ippMPMLQFixedCodebookSearch_G723(Ipp16s valBaseDelay, const Ipp16s*
pSrcImpulseResponse, const Ipp16s* pSrcResidualTarget, Ipp16s*
pDstFixedVector, Ipp16s* pResultGrid, Ipp16s* pResultTrainDirac, Ipp16s*
pResultAmpIndex, Ipp16s* pResultAmplitude, Ipp32s* pResultPosition, Ipp16s
subFrame);
```

### Parameters

<code>valBaseDelay</code>	Base delay, in the range [18,145].
<code>pSrcImpulseResponse</code>	Pointer to the impulse response vector [60].
<code>pSrcResidualTarget</code>	Pointer to the residual target signal vector [60].
<code>pDstFixedVector</code>	Pointer to the fixed codebook vector [60].
<code>pResultGrid</code>	Pointer to the beginning grid location, 0 or 1.
<code>pResultTrainDirac</code>	Pointer to the flag to show if train Dirac function is used: 0-unused, 1-used.



<i>pResultAmpIndex</i>	Pointer to the index of the quantized amplitude.
<i>pResultAmplitude</i>	Pointer to the amplitude of the fixed codebook vector.
<i>pResultPosition</i>	Pointer to the position of the fixed codebook vector, whose amplitude has non-zero value.
<i>subFrame</i>	Subframe number, from 0 to 3.

## Description

The function `ippsMPMLQFixedCodebookSearch_G723` is declared in the `ippsc.h` file. This function searches the MP-MLQ fixed codebook for the excitation in the 6.3Kbps encoder. It is applied in subframes as follows: 1. The error is obtained by the following calculation:

$$err(n) = res(n) - G \sum_{k=0}^{M-1} \alpha_k h(n - m_k)$$

where  $G$  is the gain factor,  $\psi_k$  and  $m_k$  are the signs and positions of the fixed vector, respectively.

2. Search for the parameters  $G$ ,  $\psi_k$  and  $m_k$  that minimize the mean square of the error signal  $err(n)$ .

3. The fixed codebook gain is obtained using the following formulas:

$$d(j) = \sum_{n=j}^{59} c(n) \cdot h(n-j)$$

$$Gm = \frac{\max_{j=0..59} \|d(j)\|}{\sum_{n=0}^{59} h(n) \cdot h(n)} ,$$

where  $c(n)$  is the fixed vector.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>subFrame</code> is not one of 0, 1, 2, or 3; or when <code>valBaseDelay</code> is not in the range [18,145].

## ToeplizMatrix\_G723

*Calculates 416 elements of the Toepliz matrix for fixed codebook search.*

---

### Syntax

```

IppStatus ippToeplizMatrix_G723_16s(const Ipp16s* pSrcImpulseResponse,
Ipp16s* pDstMatrix);

IppStatus ippToeplizMatrix_G723_16s32s(const Ipp16s* pSrcImpulseResponse,
Ipp32s* pDstMatrix);

```

### Parameters

*pSrcImpulseResponse* Pointer to the impulse response vector [60].  
*pDstMatrix* Pointer to the elements of Toepliz matrix vector [416].

### Description

The function `ippToeplizMatrix_G723` is declared in the `ippsc.h` file. This function calculates the 416 elements in Toepliz matrix for the fixed-codebook search. Elements of the Toepliz matrix can be expressed as follows:

$$\Phi(i, j) = \sum_{n=j}^{59} h(n-i) \times h(n-j) ,$$

$i \leq j, 0 \leq i \leq 59,$

where  $h(i), i = 0, 1, \dots, 59$ , is the impulse response.

The function stores the calculated 416 elements in *pDstMatrix* in the following order:

1.  $\Phi(m_i, m_i)$ , ( $i = 0, 1, 2, 3$ ),  $4 \times 8 = 32$  elements; starting location: 0.
2.  $\Phi(m_0, m_1)$ ,  $8 \times 8 = 64$  elements; starting location: 32.
3.  $\Phi(m_0, m_2)$ ,  $8 \times 8 = 64$  elements; starting location: 96.
4.  $\Phi(m_0, m_3)$ ,  $8 \times 8 = 64$  elements; starting location: 160.
5.  $\Phi(m_1, m_2)$ ,  $8 \times 8 = 64$  elements; starting location: 224.
6.  $\Phi(m_1, m_3)$ ,  $8 \times 8 = 64$  elements; starting location: 288.
7.  $\Phi(m_2, m_3)$ ,  $8 \times 8 = 64$  elements; starting location: 352.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Gain Quantization

These functions perform fixed codebook gain estimation and gain adjustment of the postfiltered signal.

## GainQuant\_G723

*Implements MP-MLQ gain estimation and quantization.*

### Syntax

```
ippStatus ippGainQuant_G723_16s (const Ipp16s* pImp, const Ipp16s* pSrc,
Ipp16s* pDstLoc, Ipp16s* pDstAmp, Ipp32s* pMaxErr, Ipp16s* pGrid, Ipp16s*
pAmp, int Np, int* isBest);
```

### Parameters

<code>pImp</code>	Pointer to the input impulse response $h$ vector [60].
<code>pSrc</code>	Pointer to the input target signal $r$ vector [60].
<code>pDstLoc</code>	Pointer to the output pulse locations.
<code>pDstAmp</code>	Pointer to the output pulse amplitudes.

<i>pMaxErr</i>	Pointer to the input/output quantization error.
<i>pGrid</i>	Pointer to the output grid: 0 - if pulses are in even positions, 1 - if pulses are in odd positions.
<i>pAmp</i>	Pointer to the output index of maximum amplitude.
<i>Np</i>	Number of pulses: equal to 6 for even subframe, equal to 5 for odd subframe.
<i>pIsBest</i>	POinter to the quantization result. Set to 0, if no pulses are found with quantization error better (less) than the input error.

## Description

The function `ippsGainQuant_G723` is declared in the `ippsc.h` file. This function estimates the unknown parameters,  $G$ ,  $\{\psi_k\}$ ,  $\{m_k\}$ ,  $k = 0, \dots, Np-1$ , that minimize the mean square of the error signal:

$$err[n] = r[n] - G \cdot \sum_{k=0}^{Np-1} a_k \cdot h[n-m_k]$$

The estimation is done as follows.

First, cross correlation of the impulse response and the target vector is computed:

$$d[j] = \sum_{n=j}^{59} r[n] \cdot h[n-j] ,$$

$$0 \leq n \leq 59$$

The estimated maximum gain is computed as:

$$G_{\max} = \frac{\max_{j=0..59} \{ |d[j]| \}}{\sum_{n=0}^{59} h[n] \cdot h[n]}$$

and then is quantized by the logarithmic quantizer. The gain values selected around this quantized gain are then used to optimize the signs and locations of the pulses for both even and odd grids that yield the minimum mean square of the error signal.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## GainControl\_G723

*Extracts delayed pitch contribution.*

---

### Syntax

```
IppStatus ippGainControl_G723_16s_I (Ipp32s energy, Ipp16s* pSrcDst, Ipp16s* pGain);
```

### Parameters

*energy* The input energy coefficient.  
*pSrcDst* Pointer to the input/output post-filtered signal.  
*pGain* Pointer to the input/output gain.

### Description

The function `ippGainControl_G723` is declared in the `ippsc.h` file. This function first computes the amplitude ratio  $g_s$  as:

$$g_s = \sqrt{\frac{\text{energy}}{\sum_{n=0}^{59} pSrcDst[n] \cdot pSrcDst[n]}}$$

If denominator in this expression is equal to 0,  $g_s$  is set to 1.

Then the input postfiltered signal is scaled as follows:

$$pSrcDst[n] = pSrcDst[n] * g_n(1+\alpha), n = 0, \dots, 59,$$

where  $g_n$  is updated using the following expression, respectively:

$$g_n = (1-\alpha)g_{n-1} + \alpha g_s, n = 0, \dots, 59, g_{-1} = 0, \text{ and } \alpha = 1/16.$$

The output gain is equal to  $g_{59}$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Filtering Functions

These functions implement different types of filtering, including high pass filtering on pre-processing stage of encoding and post-processing stage of decoding, as well as the postfilter in final decode stage.

Different kinds of synthesis (IIR), harmonic, and preemphasize filter functions can be combined to perform more complex filtering and may be used in different encode and decode stages.

## HighPassFilter\_G723

Performs high-pass filtering of the input signal.

### Syntax

```
ippStatus ippHighPassFilter_G723_16s (const Ipp16s* pSrc, Ipp16s* pDst,
int* pMem);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector [240].
<i>pDst</i>	Pointer to the destination vector [240].
<i>pMem</i>	Pointer to the filter memory vector [2]. This vector must be initially filled with zeroes.

## Description

The function `ippsHighPassFilter_G723` is declared in the `ippsc.h` file. This function pre-processes the input signal using the following filter transfer function:

$$H_h(z) = \frac{1 - z^{-1}}{1 - \frac{127}{128}z^{-1}}$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> , <i>pDst</i> or <i>pMem</i> pointer is NULL.

## IIR16s\_G723

*Performs IIR filtering.*

---

### Syntax

```

IppStatus ippsIIR16s_G723_16s32s (const Ipp16s* pCoeffs, const Ipp16s* pSrc,
Ipp32s* pDst, Ipp16s* pMem);

IppStatus ippsIIR16s_G723_16s_I (const Ipp16s* pCoeffs, Ipp16s* pSrcDst,
Ipp16s* pMem);

IppStatus ippsIIR16s_G723_32s16s_Sfs (const Ipp16s* pCoeffs, const Ipp32s*
pSrc, int scaleFactor, Ipp16s* pDst, Ipp16s* pMem);

```

## Parameters

<i>pCoeffs</i>	Pointer to the input vector of filter coefficients vector [20]: $b_1, \dots, b_{10}, a_1, \dots, a_{10}$ .
<i>pSrc</i>	Pointer to the input signal vector [60].
<i>pSrcDst</i>	Pointer to the input/output signal vector [60].
<i>scaleFactor</i>	Scale factor.
<i>pDst</i>	Pointer to the output filtered vector [60].
<i>pMem</i>	Pointer to the filter memory vector [20]. This vector must be initially filled with zeroes.

## Description

These functions are declared in the `ippsc.h` file.

The functions `ippsIIR16s_G723_16s32s` and `ippsIIR16s_G723_16s_I` perform infinite impulse response (IIR) filtering using the following transfer function:

$$H(z) = \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The function `ippsIIR16s_G723_32s16s_Sfs` performs IIR filtering using the transfer function:

$$H(z) = 2^{sFs} \cdot \frac{1 - \sum_{j=1}^{10} b_j z^{-j}}{1 - \sum_{j=1}^{10} a_j z^{-j}}$$

The filter memory is updated.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## SynthesisFilter\_G723

*Computes the speech signal by filtering the input speech through the synthesis filter  $1/A(z)$ .*

### Syntax

```

IppStatus ippsSynthesisFilter_G723_16s32s (const Ipp16s* pLPC, const Ipp16s*
pSrc, Ipp32s* pDst, Ipp16s* pMem);

IppStatus ippsSynthesisFilter_G723_16s (const Ipp16s* pLPC, const Ipp16s*
pSrc, Ipp16s* pMem, Ipp16s* pDst);

```

### Parameters

<code>pLPC</code>	Pointer to the input LP coefficients $a_0, a_1, \dots, a_{10}$ , in Q11.
<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the filtered output.
<code>pMem</code>	Pointer to the memory supplied for filtering: short integer vector [10], initially set to zero.

### Description

The function `ippsSynthesisFilter_G723` is declared in the `ippsc.h` file. This function computes the filter given by:

$$H(z) = \hat{A}^{-1}(z) = \frac{1}{a_0 + \sum_{i=1}^{10} a_i \cdot \hat{z}^{-i}}$$

This function is applied after the residual filter in computing the perceptually weighted speech signal:

$$pDst[n] = pSrc[n] \cdot pLPC[0] + \sum_{i=1}^{10} pLPC[i] \cdot pMem[n-i+1] \quad ,$$

$n = 0, \dots, 59$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsOverflow</code>	Indicates a warning that at least one result value was saturated.

## TiltCompensation\_G723

*Computes tilt compensation filter.*

---

### Syntax

```
IppStatus ippSTiltCompensation_G723_32s16s (Ipp16s val, const Ipp32s* pSrc,
Ipp16s* pDst);
```

### Parameters

<code>val</code>	The first-order partial correlation coefficient, in Q15.
<code>pSrc</code>	Pointer to the source vector [61].
<code>pDst</code>	Pointer to the output filtered vector [60].

### Description

The function `ippSTiltCompensation_G723` is declared in the `ippsc.h` file. This function computes the tilt compensation filter as:

$$H_t(z) = (1 - val \cdot z^{-1}),$$

$$pDst[i] = pSrc[i+1] + pSrc[i] \cdot val, \quad i = 0, \dots, 59$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## HarmonicSearch\_G723

*Searches for the harmonic delay and gain for the harmonic noise shaping filter.*

---

### Syntax

```
IpplStatus ippsHarmonicSearch_G723_16s(Ippl16s valOpenDelay, const Ippl16s*
pSrcWgtSpch, Ippl16s* pResultHarmonicDelay, Ippl16s* pResultHarmonicGain);
```

### Parameters

<i>valOpenDelay</i>	Open loop pitch, in the range [18,145].
<i>pSrcWgtSpch</i>	Pointer to the weighted speech vector [205]. The pointer points to the 146th element, in Q12.
<i>pResultHarmonicDelay</i>	Pointer to the output harmonic delay.
<i>pResultHarmonicGain</i>	Pointer to the output harmonic gain, in Q15.

### Description

The function `ippsHarmonicSearch_G723` is declared in the `ippsc.h` file. This function searches for the harmonic delay and harmonic gain for the harmonic noise shaping filter, using the weighted speech and open loop pitch as input. This function is applied in subframes.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>valOpenDelay</i> is not in the range [18, 145].

## HarmonicNoiseSubtract\_G723

*Performs harmonic noise shaping.*

---

### Syntax

```
IpplStatus ippsHarmonicNoiseSubtract_G723_16s_I(Ippl16s val, int T, const
Ippl16s* pSrc, Ippl16s* pSrcDst);
```

### Parameters

<i>val</i>	The input harmonic filter coefficient, in Q15.
<i>T</i>	The input harmonic filter lag.
<i>pSrc</i>	Pointer to the input zero impulse response vector [60] of the combined filter.
<i>pSrcDst</i>	Pointer to the input/output harmonic noise weighted speech vector [60].

### Description

The function `ippsHarmonicNoiseSubtract_G723` is declared in the `ipps.h` file. This function subtracts the harmonic shaped vector *pSrc* from vector *pSrcDst*, as follows:

$$pSrcDst[n] = pSrcDst[n] - (pSrc[n] + val * pSrc[n - T]), n = 0, \dots, 59$$

This operation is used for ringing subtraction, which is performed by subtracting the zero impulse response from the harmonic weighted speech vector to obtain the target vector:

$$t[n] = w[n] - z[n]$$

### Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## DecodeAdaptiveVector\_G723

*Restores the adaptive codebook vector from excitation, pitch, and adaptive gain.*

---

### Syntax

```
IppStatus ippsDecodeAdaptiveVector_G723_16s(Ipp16s valBaseDelay, Ipp16s
valCloseLag, Ipp16s valAdptGainIndex, const Ipp16s* pSrcPrevExcitation,
Ipp16s* pDstAdptVector, IppSpchBitRate bitRate);
```

### Parameters

<i>valBaseDelay</i>	Base delay, in the range [18,145].
<i>valCloseLag</i>	Lag of close pitch.
<i>valAdptGainIndex</i>	Index of adaptive gain.

<i>pSrcPrevExcitation</i>	Pointer to the past excitations vector [145].
<i>bitRate</i>	Transmit bit rate, equal to either <code>IPP_SPCHBR_6300</code> or <code>IPP_SPCHBR_5300</code> .
<i>pDstAdptVector</i>	Pointer to the adaptive codebook vector [60].

### Description

The function `ippsDecodeAdaptiveVector_G723` is declared in the `ippsc.h` file. This function decodes the adaptive vector from excitation, close loop pitch, and adaptive gain index. It is applied in subframes.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>valBaseDelay</i> is not in the range [18, 145]; or <i>valCloseLag</i> is not in the range [0, 2]; or <i>valAdptGainIndex</i> is not in the range [0, 169], or <i>bitRate</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## PitchPostFilter\_G723

Calculates coefficients of the pitch post filter.

### Syntax

```
IppStatus ippsPitchPostFilter_G723_16s(Ipp16s valBaseDelay, const Ipp16s*
pSrcResidual, Ipp16s* pResultDelay, Ipp16s* pResultPitchGain, Ipp16s*
pResultScalingGain, Ipp16s subFrame, IppSpchBitRate bitRate);
```

### Parameters

<i>valBaseDelay</i>	Base delay, in the range [18,145].
<i>pSrcResidual</i>	Pointer to the residual signal vector [365]. This pointer points to the 146th element.
<i>pResultDelay</i>	Pointer to the delay of the pitch post filter.
<i>pResultPitchGain</i>	Pointer to the gain of the pitch post filter, in Q15.
<i>pResultScalingGain</i>	Pointer to the scaling gain of the pitch post filter, in Q15.

*subFrame* Subframe number, from 0 to 3.

*bitRate* Transmit bit rate, equal to either `IPP_SPCHBR_6300` or `IPP_SPCHBR_5300`.

### Description

The function `ippsPitchPostFilter_G723` is declared in the `ippsc.h` file. This function calculates the coefficients of the pitch post filter. It is applied in subframes.

### Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

`ippStsRangeErr` Indicates an error when *valBaseDelay* is not in the range [18, 145], or *subFrame* is not 0 or 3, or *bitRate* is not a valid element of the enumerated type `IppSpchBitRate`.

## GSM-AMR Functions

This part of the chapter describes the Intel IPP functions that can be combined to construct a bit-exact implementation of the European Telecommunications Standards Institute (ETSI) Global System for Mobile Communications (GSM) Adaptive Multi-Rate (AMR) ETSI EN 301 704 GSM 06.90 version 7.5.0 Release 2001 speech codec, more commonly known as the “GSM-AMR 06.90” codec. The primitives are primarily concerned with the well-defined, computationally expensive core operations that comprise the codec portion of the GSM-AMR system.

The GSM 06.90 AMR speech codec comprises an adaptive multi-rate algorithm that represents efficiently telephone-bandwidth digital speech using compressed data rates of 4.75, 5.15, 5.90, 6.70, 7.40, 7.95, 10.2, and 12.2 kilobits per second (kbps). The GSM-AMR system adaptively controls speech codec bit rates such that output quality is maximized for a given set of channel conditions.

These functions are used in the Intel IPP *GSM/AMR Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

The list of these functions is given in Table 9-5.

**Table 9-5. Intel IPP G.729 Functions**

Function Base Name	Operation
Basic Functions	
<a href="#">Interpolate_GSMAMR</a>	Computes the weighted sum of two vectors.

Function Base Name	Operation
<a href="#">FFTFwd_RToPerm_GSMAMR</a>	Computes the forward fast Fourier transform (FFT) of a real signal.
Linear Prediction Analysis Functions	
<a href="#">AutoCorr_GSMAMR</a>	Estimates the autocorrelation sequence for a block of samples.
<a href="#">LevinsonDurbin_GSMAMR</a>	Calculates the LP coefficients using the Levinson-Durbin algorithm.
<a href="#">LPCToLSP_GSMAMR</a>	Converts LP coefficients to line spectrum pairs.
<a href="#">LSFToLSP_GSMAMR</a>	Converts Line Spectral Frequencies to LSP coefficients.
<a href="#">LSPToLPC_GSMAMR</a>	Converts LSPs to LP coefficients.
<a href="#">QuantLSPDEcode_GSMAMR</a>	Quantizes the LSP coefficient vector.
Adaptive Codebook Search Functions	
<a href="#">OpenLoopPitchSearchNonDTX_GSMAMR</a>	Computes the open-loop pitch lag.
<a href="#">OpenLoopPitchSearchDTXVAD1_GSMAMR</a>	Extracts an open-loop pitch lag estimate (VAD 1 scheme is enabled).
<a href="#">OpenLoopPitchSearchDTXVAD2_GSMAMR</a>	Extracts an open-loop pitch lag estimate (VAD 2 scheme is enabled).
<a href="#">AdaptiveCodebookSearch_GSMAMR</a>	Performs the adaptive codebook search.
<a href="#">AdaptiveCodebookDecode_GSMAMR</a>	Decodes the adaptive codebook parameters.
<a href="#">ImpulseResponseTarget_GSMAMR</a>	Computes the impulse response and target signal required for the adaptive codebook search.
<a href="#">AdaptiveCodebookGain_GSMAMR</a>	Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.
Fixed Codebook Search Functions	
<a href="#">AlgebraicCodebookSearch_GSMAMR</a>	Searches the algebraic codebook.

Function Base Name	Operation
<a href="#">FixedCodebookDecode_GSMAMR</a>	Decodes the fixed codebook vector.
Discontinuous Transmission functions	
<a href="#">Preemphasize_GSMAMR</a>	Computes pre-emphasis of an input signal in VAD option 2.
<a href="#">VAD1_GSMAMR</a>	Implements the VAD functionality corresponding to VAD option 1.
<a href="#">VAD2_GSMAMR</a>	Implements the VAD functionality corresponding to VAD option 2.
<a href="#">EncDTXSID_GSMAMR</a>	Extracts parameters for the SID frame.
<a href="#">EncDTXHandler_GSMAMR</a>	Determines the SID flag of current frame.
<a href="#">EncDTXBuffer_GSMAMR</a> , <a href="#">DecDTXBuffer_GSMAMR</a>	Buffer the LSP (or LSF) coefficients and previous log energy coefficients.
Post Processing Functions	
<a href="#">PostFilter_GSMAMR</a>	Filters the synthesized speech.

Basic Functions

These functions perform basic operation for codec.

Interpolate\_GSMAMR

*Computes the weighted sum of two vectors.*

Syntax

```
IppStatus ippsInterpolate_GSMAMR_16s (const Ipp16s* pSrc1, const Ipp16s*
pSrc2, Ipp16s* pDst, int len);
```

Parameters

<i>pSrc1</i>	Pointer to the first input vector[ <i>len</i> ].
<i>pSrc2</i>	Pointer to the second input vector[ <i>len</i> ].
<i>pDst</i>	Pointer to the output vector[ <i>len</i> ].



*len* Length of the input and output vectors.

### Description

The function `ippsInterpolate_GSMAMR` is declared in `ippsc.h` file. This function computes the weighted sum of two vectors as:

$$pDst[i] = (pSrc1[i] \gg 2) + (pSrc2[i] - (pSrc2[i] \gg 2)); 0 \leq i < len$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

## FFTFwd\_RToPerm\_GSMAMR

*Computes the forward fast Fourier transform (FFT) of a real signal.*

---

### Syntax

```
IppStatus ippsFFTFwd_RToPerm_GSMAMR_16s_I (Ipp16s* pSrcDst);
```

### Parameters

*pSrcDst* Pointer to the input and output vector[128].

### Description

The function `ippsFFTFwd_RToPerm_GSMAMR` is declared in `ippsc.h` file. This function computes the forward FFT of a real signal. This transform is the same as done by the general signal processing FFT function `ippsFFTFwd_RToPerm_16s_Sfs` with the following settings: *order* = 7, *flag* = `IPP_FFT_DIV_FWD_BY_N`, *scaleFactor* = -1. The result of function operation may be different because the rounding mode used in `ippsFFTFwd_RToPerm_16s_Sfs` does not guarantee a bit-exact operation of GSM AMR codec (see [Fourier Transform Functions](#)).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> pointer is <code>NULL</code> .

## LP Analysis and Quantization Functions

This section describes the GSM-AMR primitives that are concerned with LP analysis, quantization, encoding, and decoding. It also provides details on primitives that accomplish the following tasks:

- Autocorrelation analysis
- Levinson-Durbin algorithm
- LPC to LSP conversion
- LSP to LPC conversion
- LSP quantization and inverse quantization
- Quantized LSP encoding and decoding.

### AutoCorr\_GSMAMR

*Estimates the autocorrelation sequence for a block of samples.*

---

#### Syntax

```
IppStatus ippsAutoCorr_GSMAMR_16s32s(const Ipp16s* pSrcSpch, Ipp32s*
pDstAutoCorr, IppSpchBitRate mode);
```

#### Parameters

<i>pSrcSpch</i>	Pointer to the input speech vector (240 samples), represented using Q15.0. This should be aligned on an 8-byte boundary.
<i>pDstAutoCorr</i>	Pointer to the autocorrelation coefficients, of length 22. For 12.2 kbps mode, elements 0 ~ 10 contain the first set of autocorrelation lags, and elements 11 ~ 21 contain the second set of autocorrelation lags. For all other modes there is only one set of autocorrelation lags contained in vector elements 0 ~ 10.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The function `ippsAutoCorr_GSMAMR` is declared in `ippsc.h` file. This function estimates the autocorrelation sequence for a block of 240 samples (30 ms). For the 12.2 kbps mode, the 160 samples associated with the current 20 ms frame are combined with the last 80 samples from the previous frame, and two autocorrelation sequences are estimated. For all other bit rates, 160 samples from the current frame are combined with the last 40 samples from the previous frame as well as the first 40 samples from the next frame, and only one autocorrelation sequence is estimated. In particular, the estimates are formed as follows:

1. Tapered windowing – asymmetric tapered windows are applied to the input speech. Different windows are selected depending on the bit rate. For 12.2 kbps frames, unique tapered windows are applied for each of the two autocorrelation estimates, that is

$$w_1(n) = \begin{cases} 0.54 - 0.46 \cos \frac{\pi n}{159}, & n = 0, 1, \dots, 159 \\ 0.54 - 0.46 \cos \frac{(n-160)\pi}{79}, & n = 160, 161, \dots, 239 \end{cases}$$

and

$$w_2(n) = \begin{cases} 0.54 - 0.46 \cos \frac{2\pi n}{463}, & n = 0, 1, \dots, 231 \\ \cos \frac{2\pi(n-232)}{31}, & n = 232, 233, \dots, 239 \end{cases}$$

Neither  $w_1$  nor  $w_2$  has any look ahead. For all bit rates other than 12.2 kbps, a window centered on the current frame is applied, that is

$$w_3(n) = \begin{cases} 0.54 - 0.46 \cos \frac{2\pi n}{399}, & n = 0, 1, \dots, 199 \\ \cos \frac{2\pi(n-232)}{31}, & n = 200, 201, \dots, 239 \end{cases}$$

2. Estimation of autocorrelation lags – autocorrelation lags are estimated from the windowed speech samples  $s(i)$ ,  $i = 0, 1, \dots, 239$ , as follows

$$r(k) = \sum_{i=k}^{239} s(i) \times s(i-k) \quad , \quad k = 0, 1, \dots, 10$$

3. Bandwidth expansion – the following binomial lag window is applied to the autocorrelation sequence(s) obtained in step 2:

$$b_i(i) = \exp \left[ -0.5 \times \left( \frac{2\pi f_0 i}{f_S} \right)^2 \right], \quad i = 0, 1, \dots, 10$$

where  $f_0 = 60$  Hz and  $f_S = 8000$  Hz.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcSpch</code> or <code>pDstAutoCorr</code> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <code>mode</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## LevinsonDurbin\_GSMAMR

*Calculates the LP coefficients using the Levinson-Durbin algorithm.*

---

### Syntax

```
IppStatus ippLevinsonDurbin_GSMAMR_32s16s (const Ipp32s* pSrcAutoCorr,
Ipp16s* pSrcDstLpc);
```

## Parameters

<i>pSrcAutoCorr</i>	Pointer to the autocorrelation coefficients, a vector of length 11.
<i>pSrcDstLpc</i>	Pointer to the LP coefficients associated with the previous frame, a vector of length 11, represented using Q3.12. On output, updated to point to the LP coefficients associated with the current frame, a vector of length 11, represented using Q3.12.

## Description

The function `ippsLevinsonDurbin_GSMAMR` is declared in `ippsc.h` file. This function calculates the 10th-order LP coefficients from the autocorrelation lags using the Levinson-Durbin algorithm.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LPCToLSP\_GSMAMR

Converts LP coefficients to line spectrum pairs.

### Syntax

```
IppStatus ippsLPCToLSP_GSMAMR_16s(const Ipp16s* pSrcLpc, const Ipp16s*
pSrcPrevLsp, Ipp16s* pDstLsp);
```

### Parameters

<i>pSrcLpc</i>	Pointer to the eleven-element LP coefficient vector, represented using Q3.12.
<i>pSrcPrevLsp</i>	Pointer to the ten-element LSP coefficient vector associated with the previous frame, represented using Q0.15.
<i>pDstLsp</i>	Pointer to the ten-element LSP coefficient vector, represented using Q0.15.

## Description

The function `ippsLPCToLSP_GSMAMR` is declared in `ipps.h` file. This function converts a set of 10th-order LP coefficients to an equivalent set of line spectrum pairs (LSPs). The functionality is as follows:

Calculate the polynomial coefficients of  $F_1(z)$  and  $F_2(z)$  using the recursive relations

$$f_1(i+1) = a_{i+1} + a_{i-1} - f_1(i),$$

$$f_2(i+1) = a_{i+1} + a_{i-1} + f_1(i)$$

$$i = 0 \dots 4,$$

$$\text{where } f_1(0) = f_2(0) = 1.0$$

Use Chebyshev polynomials to evaluate  $F_1(z)$  and  $F_2(z)$ . The Chebyshev polynomials are given by the following formulas:

$$c_1(\omega) = \cos(5\omega) + f_1(1) \cos(4\omega) + f_1(2) \cos(3\omega) + f_1(3) \cos(2\omega) + f_1(4) \cos(\omega) + f_1(5) / 2$$

$$c_2(\omega) = \cos(5\omega) + f_2(1) \cos(4\omega) + f_2(2) \cos(3\omega) + f_2(3) \cos(2\omega) + f_2(4) \cos(\omega) + f_2(5) / 2$$

Evaluate  $F_1(z)$  and  $F_2(z)$  on 60 points equally spaced between 0 and  $\pi$ , checking for sign changes. A sign change indicates the existence of a root and the sign change interval is then divided 4 times to track the root.

If 10 roots for LSP coefficients are not found during the search, the previous set of LSPs is used.

## Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>IppsStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LSPToLPC\_GSMAMR

*Converts LSPs to LP coefficients.*

---

### Syntax

```
IppStatus ippsLSPToLPC_GSMAMR_16s(const Ipp16s* pSrcLsp, Ipp16s* pDstLpc);
```

## Parameters

<i>pSrcLsp</i>	Pointer to the ten-element LSP coefficient vector, represented using Q0.15.
<i>pDstLpc</i>	Pointer to the eleven-element LP coefficient vector, represented using Q3.12.

## Description

The function `ippSLSPToLPC_GSMAMR` is declared in `ippsc.h` file. This function converts a set of 10th-order LSPs to LP coefficients. The functionality is as follows:

**1.** Calculate the polynomial coefficients of  $F_1(z)$  and  $F_2(z)$ , using the recursive relations

for  $i = 1 \dots 5$

$$f_1(i) = 2q_{2i-1} * f_1(i-1) + 2f_1(i-2)$$

for  $j = i-1 \dots 1$

$$f_1(j) = f_1(j) - 2q_{2i-1} * f_1(j-1) + 2f_1(j-2)$$

where initial values  $f_1(0) = 1$  and  $f_1(-1) = 0$ . The coefficients  $f_2(i)$  are computed similarly by replacing  $q_{2i-1}$  by  $q_{2i}$ .

**2.** Multiply  $F_1(z)$  and  $F_2(z)$  by  $1+z^{-1}$  and  $1-z^{-1}$ , respectively, to obtain  $F'_1(z)$  and  $F'_2(z)$ , using the following relations:

$$f'_1(i) = f_1(i) + f_1(i-1),$$

$$f'_2(i) = f_2(i) + f_2(i-1), \text{ for } i = 1, 2, \dots, 5.$$

**3.** Compute the LP coefficients from the polynomials  $F'_1(z)$  and  $F'_2(z)$  as follows:

$$a_i = 0.5f'_1(i) + 0.5f'_2(i) \text{ for } i = 1, 2, \dots, 5, \text{ and}$$

$$a_i = 0.5f'_1(11-i) + 0.5f'_2(11-i) \text{ for } i = 6, 7, \dots, 11.$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LSFToLSP\_GSMAMR

*Converts Line Spectral Frequencies to LSP coefficients.*

---

### Syntax

```
IppStatus ippsLSFToLSP_GSMAMR_16s (const Ipp16s* pLSF, Ipp16s* pLSP);
```

### Parameters

<i>pLSF</i>	Pointer to the vector [10] of LSF normalized by factor 1/√ in Q14, so given in the range [0,1].
<i>pLSP</i>	Pointer to the LSP vector [10], in Q15 in the range [-1,1].

### Description

The function `ippsLSFToLSP_GSMAMR` is declared in `ipps.h` file. This function converts the Line Spectral Frequencies (LSF) to the LSP coefficients as given by:

$$lsp_n = \cos(lsf_n), 1 \leq n \leq 10.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## LSPQuant\_GSMAMR

*Quantizes the LSP coefficient vector.*

---

### Syntax

```
IppStatus ippsLSPQuant_GSMAMR_16s(const Ipp16s* pSrcLsp, Ipp16s* pSrcDstPrevQLsfResidual, Ipp16s* pDstQLsp, Ipp16s* pDstQLspIndex, IppSpchBitRate mode);
```



## Parameters

<i>pSrcLsp</i>	Pointer to the unquantized 20-element LSP vector, represented using Q0.15. For 12.2 kbps frames, the first LSP set is contained in vector elements 0 ~ 9, and the second LSP set is contained in vector elements 10 ~ 19. For all other bit rates, only elements 0 to 9 are valid and used for the quantization.
<i>pSrcDstPrevQLsfResidual</i>	Pointer to the ten-element quantized LSF residual from the previous frame, represented using Q0.15. On output, points to the ten-element quantized LSF residual for the current frame, represented using Q0.15.
<i>pDstQLsp</i>	Pointer to the 20-element quantized LSP vector, represented using Q0.15. For 12.2 kbps frames, elements 0 to 9 contain the first quantized LSP set, and elements 10 to 19 contain the second quantized LSP set. For all other bit rates there is only one LSP set contained in elements 0 to 9.
<i>pDstQLspIndex</i>	Pointer to the five-element vector of quantized LSP indices. For 12.2Kbps frames, all five elements contain valid data; for all other bit rates, only the first three elements contain valid indices (see the following discussion of SMQ and SVQ for the various modes).
<i>mode</i>	Bit rate specifier. The enumerated values of IPP_SPCHBR_4750 to IPP_SPCHBR_12200 are valid.

## Description

The function `ippsLSPQuant_GSMAMR` is declared in `ippsc.h` file. This function quantizes the LSP coefficient vector, then obtains quantized LSP codebook indices.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## QuantLSPDecode\_GSMAMR

*Decodes quantized LSPs.*

---

### Syntax

```
IppStatus ippsQuantLSPDecode_GSMAMR_16s(const Ipp16s* pSrcQLspIndex, Ipp16s*
pSrcDstPrevQLsfResidual, Ipp16s* pSrcDstPrevQLsf, Ipp16s* pSrcDstPrevQLsp,
Ipp16s* pDstQLsp, Ipp16s bfi, IppSpchBitRate mode);
```

### Parameters

<i>pSrcQLspIndex</i>	Pointer to the five-element vector containing codebook indices of the quantized LSPs. For 12.2 kbps frames, all five elements contain valid indices; for all other bit rates, only the first three elements contain valid indices.
<i>pSrcDstPrevQLsfResidual</i>	Pointer to the ten-element quantized LSF residual from the previous frame, represented using Q0.15. On output, points to the ten-element quantized LSF residual for the current frame, represented using Q0.15.
<i>pSrcDstPrevQLsf</i>	Pointer to the ten-element quantized LSF vector from the previous frame, represented using Q0.15. On output, points to the ten-element updated quantized LSF vector, represented using Q0.15.
<i>pSrcDstPrevQLsp</i>	Pointer to the ten-element quantized LSP vector from the previous frame, represented using Q0.15. On output, points to the ten-element updated quantized LSP vector, represented using Q0.15.
<i>pDstQLsp</i>	Pointer to a 40-element vector containing four subframe LSP sets. Two sets are generated by interpolation for 12.2 kbps frames; for all other bit rates three sets are generated by interpolation. All elements are represented in using Q0.15.
<i>bfi</i>	Bad frame indicator; "0" means a good frame; all other values mean a bad frame.
<i>mode</i>	Bit rate specifier. The enumerated values of IPP_SPCHBR_4750 to IPP_SPCHBR_12200 are valid.

## Description

The function `ippsQuantLSPDecode_GSMAMR` is declared in `ipps.h` file. This function decodes quantized LSPs from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized LSPs from previous quantized LSPs using linear interpolation. The functionality can be summarized as follows:

1. If no errors are detected on the current frame, obtain the quantized LSPs from the codebook indices and the previous quantized residual using inverse LSP quantization.
2. If errors are detected on the current frame, quantized LSFs are obtained using the following rate-dependent interpolation scheme:

$lsf\_q1(i) = lsf\_q2(i) = \alpha * past\_lsf\_q(i) + (1 - \alpha) * mean\_lsf\_q(i)$ , 12.2 kbit/s mode;

$lsf\_q(i) = \alpha * past\_lsf\_q(i) + (1 - \alpha) * mean\_lsf\_q(i)$ , otherwise

Here  $i = 0, 1, \dots, 9$ ,  $\psi = 0.95$ ,  $lsf\_q1$  and  $lsf\_q2$  (for 12.2 kbps) are two sets of quantized LSF vectors for current frame,  $past\_lsf\_q$  is  $lsf\_q2$  of the previous frame, and  $mean\_lsf$  is the average LSF vector. Note that there is only one set of quantized LSF coefficients for rates other than 12.2 kbps. The corresponding quantized LSPs are obtained by LSF-to-LSP conversion.

Linear interpolation is applied to generate four sets of quantized LSPs from the decoded set(s) of LSPs and the quantized LSPs from the previous frame.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcQLspIndex</code> , <code>pSrcDstPrevQLsfResidual</code> , <code>pSrcDstPrevQLsf</code> , <code>pSrcDstPrevQLsp</code> or <code>pDstQLsp</code> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <code>mode</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## Adaptive Codebook Functions

This section describes primitives that are concerned with various aspects of the adaptive codebook, including primitives that perform the following functions:

- Open-loop pitch search, including Non-DTX, VAD1, and VAD2
- Impulse response and target signal computation

- Adaptive codebook search
- Decoding of the adaptive codebook vector.

### Open-Loop Pitch Search (OLP)

Three functions are provided for OLP search. Use of these functions is mutually exclusive, that is, only one is appropriate for an application during any given frame type. The appropriate choice of an OLP search function depends upon the state of the DTX and VAD modules. The OLP search functions should be applied as follows:

Encoder Mode	Appropriate Function
DTX disabled	<code>ippsOpenLoopPitchSearchNonDTX_GSMAMR</code>
DTX, VAD 1 enabled	<code>ippsOpenLoopPitchSearchDTXVAD1_GSMAMR</code>
DTX, VAD 2 enabled	<code>ippsOpenLoopPitchSearchDTXVAD2_GSMAMR</code>

The OLP search functions extract a pitch estimate from a weighted version of the input speech. For 5.15 and 4.75 kbps frames, the search is performed once per frame. For all other modes, the search is performed twice per frame. A unique search is employed for 10.2 kbps frames. The OLP search details are as follows:

If the transmission bit rate is 10.2 kbps,

- Compute a windowed autocorrelation of weighted speech, that is,

$$R(k) = w(k) \times \sum_{n=0}^{79} sw(n) \times sw(n-k), \quad 20 \leq k \leq 143$$

where the sequence  $sw(n)$  contains weighted speech, and  $w$  is the weighting function given by

$$w(k) = wl(k) \times wn(k)$$

In this weighting function,  $wl$  emphasizes low pitch, and is defined in terms of the table  $cw$ , while  $wn$  is a sequence of neighboring emphasis lag coefficients associated with the previous frame, that is

$$wn(k) = \begin{cases} cw(|T_{old} - k| + 20) & weighflag = 1 \\ 1 & \text{otherwise} \end{cases}$$

and the parameter  $T_{old}$  is the median filtered pitch lag of 5 previous voiced speech half frames. The estimated open-loop pitch lag is the value  $k$  that maximizes  $R(k)$ . It is denoted by  $T_{op}$ .

b. Compute the optimal open-loop gain using the relation

$$g = \frac{\sum_{n=0}^{79} sw(n) \times sw(n - T_{op})}{\sum_{n=0}^{79} sw(n) \times sw(n)} - 0.4$$

In addition,

$$v = \begin{cases} 1 & g > 0 \\ 0.9v & \text{otherwise} \end{cases}$$

and

$$weighflag = \begin{cases} 1 & v > 0 \\ 0 & \text{otherwise} \end{cases}$$

If  $g > 0$ , the previous pitch lag buffer and  $v$  median pitch lag of the previous pitch lags are updated.

2. For rates other than 10.2 kbps, the following OLP search procedure is employed:

a. Three maximums are found in three different ranges for the correlations given by

$$R(k) = \sum_{n=0}^{length-1} sw(n) \times sw(n-k)$$

For 5.15 and 4.75 kbps frames,  $length = 160$ .

For all other bit rates (except 10.2 kbps),  $length = 80$ .

b. Normalize the three maximum correlations according to

$$M_i = \frac{M_i}{\sqrt{\sum_{n=0}^{length-1} sw^2(n - T_i)}} \quad i = 1, 2, 3$$

c. Determine the best open-loop lag using the following rule:

$T_{op} = T_1, M(T_{op}) = M_1$ , if  $M_2 > 0.85 M(T_{op})$ ;

$M(T_{op}) = M_2, T_{op} = T_2$ , if  $M_3 > 0.85 M(T_{op})$ ;

$T_{op} = T_3$ .

Each of the OLP search functions implements the rate-dependent search algorithms described above. Next, details are given for non-DTX, VAD1, and VAD2 OLP search functions.

## OpenLoopPitchSearchNonDTX\_GSMAMR

*Computes the open-loop pitch lag.*

---

### Syntax

```
IppStatus ippsOpenLoopPitchSearchNonDTX_GSMAMR_16s(const Ipp16s* pSrcWgtLpc1,
const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcSpch, Ipp16s*
pValResultPrevMidPitchLag, Ipp16s* pValResultVvalue, Ipp16s*
pSrcDstPrevPitchLag, Ipp16s* pSrcDstPrevWgtSpch, Ipp16s* pDstOpenLoopLag,
Ipp16s* pDstOpenLoopGain, IppSpchBitRate mode);
```

## Parameters

<i>pSrcWgtLpc1</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the numerator of the perceptual weighting filter.
<i>pSrcWgtLpc2</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the denominator of the perceptual weighting filter.
<i>pSrcSpch</i>	Pointer to the 170-element input speech vector, represented using Q15.0.
<i>pValResultPrevMidPitchLag</i>	Pointer to a vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>pValResultVvalue</i>	Pointer to the adaptive parameter $v$ , represented using Q0.15. On output, points to the updated adaptive parameter $v$ , represented using Q0.15. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevPitchLag</i>	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevWgtSpch</i>	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output, points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.

<i>pDstOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
<i>pDstOpenLoopGain</i>	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The function `ippsOpenLoopPitchSearchNonDTX_GSMAMR` is declared in `ippsc.h` file. This function computes the open-loop pitch lag (as well as optimal pitch gain for 10.2 kbps frames only) when both DTX and VAD are disabled. The search algorithm is described in [Open-Loop Pitch Search \(OLP\)](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>mode</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## OpenLoopPitchSearchDTXVAD1\_GSMAMR

*Extracts an open-loop pitch lag estimate (VAD 1 scheme is enabled).*

---

## Syntax

```
IppStatus ippsOpenLoopPitchSearchDTXVAD1_GSMAMR_16s(const Ipp16s* pSrcWgtLpc1,
const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcSpch, Ipp16s* pValResultToneFlag,
Ipp16s* pValResultPrevMidPitchLag, Ipp16s* pValResultVvalue, Ipp16s*
pSrcDstPrevPitchLag, Ipp16s* pSrcDstPrevWgtSpch, Ipp16s* pResultMaxHpCorr,
Ipp16s* pDstOpenLoopLag, Ipp16s* pDstOpenLoopGain, IppSpchBitRate mode);
```



## Parameters

<i>pSrcWgtLpc1</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the numerator of the perceptual weighting filter.
<i>pSrcWgtLpc2</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the denominator of the perceptual weighting filter.
<i>pSrcSpch</i>	Pointer to the 170-element input speech vector, represented using Q15.0.
<i>pValResultToneFlag</i>	Pointer to the tone flag for the VAD module. On output, points to the updated tone flag for the VAD module.
<i>pValResultPrevMidPitchLag</i>	Pointer to a vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>pValResultVvalue</i>	Pointer to the adaptive parameter $v$ , represented using Q0.15. On output, points to the updated adaptive parameter $v$ , represented using Q0.15. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevPitchLag</i>	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevWgtSpch</i>	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output, points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.

<i>pResultMaxHpCorr</i>	Pointer to the correlation maximum.
<i>pDstOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
<i>pDstOpenLoopGain</i>	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The function `ippsOpenLoopPitchSearchDTXVAD1_GSMAMR` is declared in `ippsc.h` file. This function extracts an open-loop pitch lag estimate from the weighted input speech when the VAD 1 scheme is enabled using a version of the pitch search algorithm described in [Open-Loop Pitch Search](#) (OLP) that is modified as follows:

1. For 10.2 kbps frames, the following modification is applied after the best open-loop pitch is found:

Update the tone flag (when initialized or reset, it is set to 0) in the following way

*toneflag* >>=1

$$\begin{aligned}
 DelayEnergy &= \sum_{n=0}^{79} sw^2(n - T_{op}) \\
 MaxCorr &= \sum_{n=0}^{79} sw(n - T_{op}) \times sw(n)
 \end{aligned}$$

if  $0.325 * DelayEnergy < MaxCorr$ ,  $toneflag = toneflag | 0x4000$ .

On the second OLP search for the current frame, find the maximum of the high-pass filtered autocorrelations, that is

$maxhpcorr = \max(2R(k) - R(k-1) - R(k+1) | k = 142, \dots, 21)$

Then, *maxhpcorr* is normalized by  $NormFactor = frameEnergy - frameCorr$ :

$$frameEnergy = \sum_{n=0}^{79} sw^2(n) , \quad frameCorr = \sum_{n=0}^{79} sw(n) \times sw(n-1)$$

2. For all other bit rates, the following modifications are applied:

a. Before the open-loop pitch search, update the tone flag as follows:

```
toneflag = toneflag >> 1
```

If the bit rate is either 5.15 or 4.75 kbps, update the tone flag as follows:

```
toneflag = toneflag >> 1, toneflag = toneflag | 0x2000
```

b. After finding three open-loop pitch candidates, update the tone flag as follows:

```
if (DelayEnergy × 0.65 < MaxCorr) toneflag = toneflag | 0x4000
```

This update is repeated three times with *DelayEnergy* and *MaxCorr* corresponding to the three pitch candidates. Note that the computation length of *DelayEnergy* and *MaxCorr* for 4.75 and 5.15 kbps frames is 160 samples. For all other bit rates, the length is 80 samples.

c. On the second OLP search for each frame, find the maximum of the high passed autocorrelations. The implementation is identical the 10.2 kbps correlation search, but the search range is rate-dependent.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## OpenLoopPitchSearchDTXVAD2\_GSMAMR

*Extracts an open-loop pitch lag estimate (VAD 2 scheme is enabled).*

### Syntax

```
IppStatus ippOpenLoopPitchSearchDTXVAD2_GSMAMR_16s32s(const Ipp16s*
pSrcWgtLpc1, const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcSpch, Ipp16s*
pValResultPrevMidPitchLag, Ipp16s* pValResultVvalue, Ipp16s*
```

```
pSrcDstPrevPitchLag, Ipp16s* pSrcDstPrevWgtSpch, Ipp32s* pResultMaxCorr,  
Ipp32s pResultWgtEnergy, Ipp16s* pDstOpenLoopLag, Ipp16s* pDstOpenLoopGain,  
IppSpchBitRate mode);
```

## Parameters

<i>pSrcWgtLpc1</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the numerator of the perceptual weighting filter.
<i>pSrcWgtLpc2</i>	Pointer to the 44-element vector of weighted LP coefficients, represented using Q3.12. These LP coefficients comprise the denominator of the perceptual weighting filter.
<i>pSrcSpch</i>	Pointer to the 170-element input speech vector, represented using Q15.0.
<i>pValResultToneFlag</i>	Pointer to the tone flag for the VAD module.
<i>pValResultPrevMidPitchLag</i>	Pointer to a vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. On output, points to the updated vector of median filtered pitch lags from the five previous voiced speech half-frames, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>pValResultVvalue</i>	Pointer to the adaptive parameter $v$ , represented using Q0.15. On output, points to the updated adaptive parameter $v$ , represented using Q0.15. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevPitchLag</i>	Pointer to a five-element vector of pitch lags associated with the five most recent voiced speech half-frames. On output, points to the updated five-element vector of pitch lags associated with the five most recent voiced speech half-frames. This argument is valid only for 10.2 kbps frames.
<i>pSrcDstPrevWgtSpch</i>	Pointer to a 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0. On output,

---

	points to the updated 143-element vector containing the perceptually weighted speech associated with the previous frame, represented using Q15.0.
<i>pResultMaxCorr</i>	Pointer to the correlation maximum.
<i>pResultWgtEnergy</i>	Pointer to the pitch delayed energy of the weighted speech signal, as described above. The output may be scaled, and the Q representation is not given.
<i>pDstOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
<i>pDstOpenLoopGain</i>	Pointer to a two-element vector containing optimal open-loop pitch gains, represented using Q15.0. This argument is valid only for 10.2 kbps frames.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The function `ippsOpenLoopPitchSearchDTXVAD2_GSMAMR` is declared in `ippsc.h` file. This function extracts an open-loop pitch lag estimate from the weighted input speech when the VAD 2 scheme is enabled, using a version of the pitch search algorithm described in [Open-Loop Pitch Search](#) (OLP) that is modified in the following way:

After finding the best open-loop pitch, extract from the weighted speech the maximum correlation *MaxCorr* and the delayed energy *DelayEnergy*. For both 4.75 and 5.15 kbps frames, the computation is carried for 160 samples. For all other bit rates, the computation is carried for only 80 samples using the relations

$$MaxCorr = \sum_{n=0}^{length-1} sw(n) \times sw(n - T_{op})$$

$$DelayEnergy = \sum_{n=0}^{length-1} s w^2(n - T_{op})$$

The *MaxCorr* and *DelayEnergy* values corresponding to the two half-frame open-loop pitch lag estimates are combined to obtain whole-frame estimates of *MaxCorr* and *DelayEnergy* (except during 4.75 and 5.15 kbps frames, when the OLP search is performed only once per frame).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## ImpulseResponseTarget\_GSMAMR

*Computes the impulse response and target signal required for the adaptive codebook search.*

---

### Syntax

```
IppStatus ippImpulseResponseTarget_GSMAMR_16s(const Ipp16s* pSrcSpch, const
Ipp16s* pSrcWgtLpc1, const Ipp16s* pSrcWgtLpc2, const Ipp16s* pSrcQLpc, const
Ipp16s* pSrcSynFltState, const Ipp16s* pSrcWgtFltState, Ipp16s*
pDstImpulseResponse, Ipp16s* pDstLpResidual, Ipp16s* pDstAdptTarget);
```

### Parameters

<i>pSrcSpch</i>	Pointer to the 50-element input speech vector, where elements 0 - 9 are from the previous subframe, and elements 10 - 49 are from the current subframe.
<i>pSrcWgtLpc1</i>	Pointer to an eleven-element vector of weighted LP coefficients associated with $A(z/\gamma_1)$ on the current subframe, represented using Q3.12.

<i>pSrcWgtLpc2</i>	Pointer to an eleven-element vector of weighted LP coefficients associated with $A(z/\gamma_2)$ on the current subframe, represented using Q3.12.
<i>pSrcQLpc</i>	Pointer to an eleven-element vector of quantized LP coefficients for the current subframe, represented using Q3.12.
<i>pSrcSynFltState</i>	Pointer to the ten-element vector that contains the state of the synthesis filter, represented using Q15.0.
<i>pSrcWgtFltState</i>	Pointer to the ten-element vector that contains the state of the weighting filter, represented using Q15.0..
<i>pDstImpulseResponse</i>	Pointer to the 40-element vector that contains the impulse response, represented using Q3.12.
<i>pDstLpResidual</i>	Pointer to the 40-element vector that contains the LP residual, represented using Q15.0.
<i>pDstAdptTarget</i>	Pointer to the 40-element vector that contains the LP residual, represented using Q15.0.

## Description

The function `ippsImpulseResponseTarget_GSMAMR` is declared in `ippsc.h` file. This function computes the impulse response and target signal required for the adaptive codebook search. This function is performed on a subframe basis using the following approach:

1. The impulse response  $h(n)$  of the weighted synthesis filter,  $H(z)W(z) = A(z/\gamma_1) / [\hat{A}(z)A(z/\gamma_2)]$ , is computed by applying the filters  $1/\hat{A}(z)$  and  $1/A(z/\gamma_2)$  to the zero-padded impulse response of the filter  $A(z/\gamma_1)$ .
2. The target signal is then obtained by applying to the LP residual  $res_{LP}(n)$  the cascaded synthesis and weighting filters  $1/\hat{A}(z)$  and  $A(z/\gamma_1)/A(z/\gamma_2)$ , respectively. The adaptive codebook search also uses the residual signal  $res_{LP}(n)$  to update the history of past excitations. The LP residual is obtained by inverse filtering the input speech, that is

$$res_{LP}(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i s(n-i)$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## AdaptiveCodebookSearch\_GSMAMR

*Performs the adaptive codebook search.*

---

### Syntax

```
IppStatus ippsAdaptiveCodebookSearch_GSMAMR_16s(const Ipp16s* pSrcTarget,
const Ipp16s* pSrcImpulseResponse, Ipp16s* pSrcOpenLoopLag, Ipp16s*
pValResultPrevIntPitchLag, Ipp16s* pSrcDstExcitation, Ipp16s*
pResultFracPitchLag, Ipp16s* pResultAdptIndex, Ipp16s* pDstAdptVector, Ipp16s
subFrame, IppSpchBitRate mode);
```

### Parameters

<i>pSrcTarget</i>	Pointer to the 40-element adaptive target signal vector, represented using Q15.0. This should be aligned on an 8-byte boundary.
<i>pSrcImpulseResponse</i>	Pointer to the 40-element impulse response of the weighted synthesis filter, represented using Q3.12. This should be aligned on an 8-byte boundary.
<i>pSrcOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags. For 5.15 and 4.75 kbps frames, only the first vector element contains a valid lag value, since only one lag is estimated. For all other bit rates, both vector elements contain valid pitch lag values.
<i>pValResultPrevIntPitchLag</i>	Pointer to the previous integral pitch lag.
<i>pSrcDstExcitation</i>	Pointer to the 194-element excitation vector. Elements 0 ~ 153 contain the past excitation, represented using Q15.0. Elements 154 ~ 193 are used as a buffer whenever the subframe length exceeds the pitch lag. On output, elements 154 - 193 are updated to contain the adaptive codebook vector.
<i>pResultFracPitchLag</i>	Pointer to the fractional pitch lag obtained during the adaptive codebook search.



<i>pResultAdptIndex</i>	Pointer to the coded closed-loop pitch index.
<i>pDstAdptVector</i>	Pointer to the 40-sample adaptive codebook vector, represented using Q15.0.
<i>subFrame</i>	Subframe index.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The function `ippsAdaptiveCodebookSearch_GSMAMR` is declared in `ippsc.h` file. This function performs the adaptive codebook search. The adaptive codebook search consists of a closed-loop pitch search followed by computation of an adaptive excitation vector. The adaptive excitation vector is obtained by interpolating the past excitation at the fractional pitch lag obtained during the closed-loop pitch search. The adaptive codebook is searched on every subframe.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> is not in the range <code>[0, 3]</code> .

## AdaptiveCodebookDecode\_GSMAMR

Decodes the adaptive codebook parameters.

### Syntax

```
IppStatus ippsAdaptiveCodebookDecode_GSMAMR_16s(Ipp16s valAdptIndex, Ipp16s*
pValResultPrevIntPitchLag, Ipp16s* pValResultLtpLag, Ipp16s*
pSrcDstExcitation, Ipp16s* pResultIntPitchLag, Ipp16s* pDstAdptVector, Ipp16s
subFrame, Ipp16s bfi, Ipp16s inBackgroundNoise, Ipp16s voicedHangover,
IppSpchBitRate mode);
```

### Parameters

<i>valAdptIndex</i>	Adaptive codebook index.
---------------------	--------------------------

<i>pValResultPrevIntPitchLag</i>	Pointer to the previous integer pitch lag. Used as an output argument also.
<i>pValResultLtpLag</i>	Pointer to the LTP-Lag value. Used as an output argument also.
<i>pSrcDstExcitation</i>	Pointer to the 194-element excitation vector. Elements 0 ~ 153 contain the past excitation, represented using Q15.0. Elements 154 ~ 193 are used as a buffer whenever the subframe length exceeds the pitch lag. On output, elements 154 - 193 are updated to contain the adaptive codebook vector.
<i>pResultIntPitchLag</i>	Pointer to the integer pitch.
<i>pDstAdptVector</i>	Pointer to the 40-sample adaptive codebook vector, represented using Q15.0.
<i>subFrame</i>	Subframe index.
<i>bfi</i>	Bad frame indicator. "0" signifies a good frame; any other value signifies a bad frame.
<i>inBackgroundNoise</i>	Flag set when the previous frame is considered to contain background noise and only shows minor energy level changes.
<i>voicedHangover</i>	Counter used to monitor the time since a frame was presumably voiced.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The function `ippsAdaptiveCodebookDecode_GSMAMR` is declared in `ippsc.h` file. This function decodes the adaptive codebook parameters transmitted by the encoder, and then applies them to interpolate an adaptive codebook vector. If errors are detected on the received frame, previously received parameters are used to approximate the parameters of the current frame and the adaptive codebook vector interpolation procedure is carried with the approximated parameter set. Adaptive codebook vectors are decoded for every subframe as follows:

1. If no errors are detected on the current frame, integer and fractional pitch lags are extracted from the adaptive codebook indices.

2. If errors are detected, the integer pitch is recovered either from the previous integer pitch or the LTP-Lag, and the fractional pitch is set to zero. The LTP-Lag value is replaced by the integer pitch of the 4<sup>th</sup> subframe of the previous frame (12.2 Kbps mode) or slightly modified values based on the last correctly received value (all other modes).
3. The same adaptive codebook interpolation procedure described in section 13.4.3 is applied to obtain the adaptive codebook vector.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> is not in the range [0, 3].

## AdaptiveCodebookGain\_GSMAMR

*Calculates the gain of the adaptive-codebook vector and the filtered codebook vector.*

---

### Syntax

```
IppStatus ippsAdaptiveCodebookGain_GSMAMR_16s (const Ipp16s* pSrcAdptTarget,
const Ipp16s* pSrcFltAdptVector, Ipp16s* pResultAdptGain);

IppStatus ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s (const Ipp16s*
pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, Ipp16s* pResultAdptGain,
Ipp16s* pResultAdptGainCoeffs);
```

### Parameters

<i>pSrcAdptTarget</i>	Pointer to the adaptive target signal vector [40].
<i>pSrcFltAdptVector</i>	Pointer to the filtered adaptive-codebook vector [40], in Q12.
<i>pResultAdptGain</i>	Pointer to the output adaptive-codebook gain $g_p$ , in Q14.
<i>pResultAdptGainCoeffs</i>	Pointer to the output gain coefficients vector [4] , in Q15.

## Description

The functions `ippsAdaptiveCodebookGain_GSMAMR` and `ippsAdaptiveCodebookGainCoeffs_GSMAMR` are declared in `ippsc.h` file.

**`ippsAdaptiveCodebookGain_GSMAMR_16s`.** This function calculates the adaptive-codebook gain  $g_p$  as given by:

$$g_p = \frac{\sum_{n=0}^{39} x(n)y(n)}{\sum_{n=0}^{39} y(n)y(n)}$$

bounded by  $0 \leq g_p \leq 1.2$ , in Q14,

where  $x$  is the adaptive target signal vector, and  $y$  is the filtered adaptive-codebook vector.

See also the [ippsAdaptiveCodebookGain\\_G729](#) function.

**`ippsAdaptiveCodebookGainCoeffs_GSMAMR_16s`.** This function calculates the adaptive-codebook gain  $g_p$  in the same way as the `ippsAdaptiveCodebookGain_GSMAMR_16s` function does, and additionally returns the gain in a different representation given by the following formula:

$$g_p = \frac{c_{xy} 2^{\exp_{xy}}}{c_{yy} 2^{\exp_{yy}}}, 1/2 \leq |c_{xy}|, |c_{yy}| < 1, Q15$$

The mantissas  $c_{xy}$ ,  $c_{yy}$  and exponents  $\exp_{xy}$ ,  $\exp_{yy}$  for both the normalized denominator and normalized numerator are returned in the `pResultAdptGainCoeffs` vector:

`pResultAdptGainCoeffs[0] =  $c_{yy}$ ,`

`pResultAdptGainCoeffs[1] =  $\exp_{yy}$ ,`

`pResultAdptGainCoeffs[2] =  $c_{xy}$ ,`

`pResultAdptGainCoeffs[3] =  $\exp_{xy}$ .`

If  $c_{xy} < 4$ , then zero gain is returned.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Fixed Codebook Search

This section describes primitives that are concerned with the fixed codebook, including primitives that perform the following functions:

- Fixed (algebraic) codebook search
- Fixed codebook vector decode

## AlgebraicCodebookSearch\_GSMAMR

*Searches the algebraic codebook.*

---

### Syntax

```
IppStatus ippAlgebraicCodebookSearch_GSMAMR_16s(Ipp16s valIntPitchLag,
Ipp16s valBoundQAdptGain, const Ipp16s* pSrcFixedTarget, const Ipp16s*
pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse, Ipp16s* pDstFixedVector,
Ipp16s* pDstFltFixedVector, Ipp16s* pDstEncPosSign, Ipp16s subFrame,
IppSpchBitRate mode);
```

```
IppStatus ippAlgebraicCodebookSearchEX_GSMAMR_16s(Ipp16s valIntPitchLag,
Ipp16s valBoundQAdptGain, const Ipp16s* pSrcFixedTarget, const Ipp16s*
pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse, Ipp16s* pDstFixedVector,
Ipp16s* pDstFltFixedVector, Ipp16s* pDstEncPosSign, Ipp16s subFrame,
IppSpchBitRate mode, Ipp32s* pBuffer);
```

### Parameters

<code>valIntPitchLag</code>	The nearest integer pitch lag $T$ to the closed-loop fractional pitch lag of this subframe, which is computed by closed-loop pitch search routine.
<code>valBoundQAdptGain</code>	Bounded quantized adaptive codebook gain. For MR122 mode, this value is the bounded quantized pitch gain of current subframe. While for other modes,

	it is the bounded quantized pitch gain of previous subframe. This value is represented using Q1.14 format.
<i>pSrcFixedTarget</i>	Pointer to the 40-element fixed target signal vector $x_2(n)$ , which is used to search the fixed codebook vector, represented using Q15.0. This should be aligned on an 8-byte boundary.
<i>pSrcLtpResidual</i>	Pointer to the 40-element long-term prediction residual signal vector $res_{LTP}(n)$ , represented using Q15.0.
<i>pSrcDstImpulseResponse</i>	Pointer to the 40-element weighted synthesis filter impulse response vector, represented using Q3.12. This should be aligned on an 8-byte boundary. On output, points to the updated 40-element impulse response vector, which is obtained by filtering original impulse response $h(n)$ through the pre-filter $F_E(z)$ . It is represented using Q3.12.
<i>pDstFixedVector</i>	Pointer to the 40-element fixed codebook vector $c(n)$ , represented using Q2.13.
<i>pDstFltFixedVector</i>	Pointer to the 40-element filtered fixed codebook vector $z(n)$ , which is obtained by convolving the impulse response with the fixed codebook vector, represented using Q2.13.
<i>pDstEncPosSign</i>	Pointer to the ten-element buffer that contains the encoded positions and signs of optimal pulses. For 12.2 kbps mode, 10 short words are used to store the result of this encoding. For the 10.2 kbps mode, 7 short words are used. For all other modes, only 2 short words are used.
<i>subFrame</i>	Subframe index, which ranges from 0 to 3.
<i>pBuffer</i>	Pointer to internal working buffer, of length 1K.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The functions `ippsAlgebraicCodebookSearch_GSMAMR` and `ippsAlgebraicCodebookSearchEX_GSMAMR` are declared in `ippsc.h` file. These functions search the algebraic codebook by minimizing the mean square error between the weighted input speech and the weighted synthesized speech. After the fixed codebook vector has been obtained, it is filtered through the weighted synthesis filter to obtain a fixed codebook vector. The positions and signs of the optimal pulses are encoded respectively according to the GSM06.90 specification. Algebraic codebook search is applied on each subframe.

These two functions work identically with the following exception: `ippsAlgebraicCodebookSearchEX_GSMAMR` uses an internal working buffer pointed by `pBuffer` allocated by user, but `ippsAlgebraicCodebookSearch_GSMAMR` allocates this internal working buffer in stack.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>mode</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## FixedCodebookDecode\_GSMAMR

*Decodes the fixed codebook vector.*

### Syntax

```
IppStatus ippsFixedCodebookDecode_GSMAMR_16s(const Ipp16s* pSrcFixedIndex,
Ipp16s* pDstFixedVector, Ipp16s subFrame, IppSpchBitRate mode);
```

### Parameters

<code>pSrcFixedIndex</code>	Pointer to the fixed codebook index vector. If the mode is 12.2 kbps, the vector length is 10; if the mode is 10.2 kbps, the vector length is 7; otherwise the vector length is 2.
<code>pDstFixedVector</code>	Pointer to the 40-element fixed codebook vector.
<code>subFrame</code>	Subframe index.
<code>mode</code>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The function `ippsFixedCodebookDecode_GSMAMR` is declared in `ippsc.h` file. This function decodes the fixed codebook vector from the received fixed codebook index.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> is not in the range <code>[0, 3]</code> .

## Discontinuous Transmission (DTX)

This section describes primitives that are concerned with discontinuous transmission (DTX), including primitives that perform the following functions:

- Signal pre-emphasis prior to VAD option 2
- VAD Decision Function for Option 1
- VAD Decision Function for Option 2
- Parameter Extraction for the SID frame
- DTX Handler
- DTX Buffering

Each of these primitives is described next.

## Preemphasize\_GSMAMR

*Computes pre-emphasis of an input signal in VAD option 2.*

---

### Syntax

```
IppStatus ippsPreemphasize_GSMAMR_16s (Ipp16s gamma, const Ipp16s* pSrc,
Ipp16s* pDst, int len, Ipp16s* pMem);
```

### Parameters

*gamma*                                      The filter coefficient, in Q15.



<i>pSrc</i>	Pointer to the source vector, in Q0.
<i>pDst</i>	Pointer to the destination vector, in Q0.
<i>len</i>	Number of elements in the source and destination vectors.
<i>pMem</i>	Pointer to the filter memory [1].

### Description

The function `ippsPreemphasize_GSMAMR` is declared in the `ippsc.h` file. This function computes pre-emphasis of the input signal prior to frequency domain conversion in Voice Activity Detector option 2. The function `ippsPreemphasize_GSMAMR` performs the same operation as the [ippsPreemphasize\\_G729A](#) function does, but has a slightly different order to deliver accuracy required by the GSM-AMR transcoding standard.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## VAD1\_GSMAMR

*Implements the VAD functionality corresponding to VAD option 1.*

### Syntax

```
IppStatus ippsVAD1_GSMAMR_16s(const Ipp16s pSrcSpch, IppGSMAMRVad1State*
pValResultVad1State, Ipp16s* pResultVadFlag, Ipp16s maxHpCorr, Ipp16s
toneFlag);
```

### Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal, of length 160, in Q0.
<i>pValResultVad1State</i>	On input, pointer to the VAD Option 1 history variables. On output, points to the updated VAD Option 1 history variables. The structure <code>IppGSMAMRVad1State</code> is defined below.

<i>pResultVadFlag</i>	Pointer to the VAD flag of this frame. If it is set to "1", it indicates the presence of signals that should be transmitted. If set to "0", there is no signals in this frame needed to be transmitted.
<i>maxHpCorr</i>	<i>best_corr_hp</i> value of previous frame, which is the maximum normalized value of the high pass filtered correlation. This value is the output of the open-loop pitch search function..
<i>toneFlag</i>	Tone flag, which indicates the presence of information tones or signals containing very strong periodic component. This value is the output of the open-loop pitch search function.

## Description

The function `ippsVAD1_GSMAMR` is declared in `ippsc.h` file. This function implements the VAD functionality corresponding to VAD option 1 of *ETSI GSM 06.94*. It is used to indicate whether each 20ms frame contains signals that should be transmitted - for example, speech, music or information tones. The structure `IppGSMAMRVad1State` contains the history variables of VAD Option 1. These variables are initialized before the beginning of the encoder, and can be only updated by this function. Refer to *ETSI GSM 06.94* VAD Option 1 specification for details of the implementation.

<b>typedef struct{</b>	<b>Description</b>
<code>Ipp16s pPrevSignalLevel[9];</code>	Signal level vector of <i>level[n]</i> previous frame.
<code>Ipp16s pPrevSignalSublevel[9];</code>	Intermediate signal sublevel vector of previous frame.
<code>Ipp16s pPrevAverageLevel[9];</code>	Average signal level vector <i>ave_level[ n]</i> of previous frame.
<code>Ipp16s pBkgNoiseEstimate[9];</code>	Background noise estimate vector <i>back_est[ n]</i> of previous frame.
<code>Ipp16s pFifthFltState[6];</code>	The history state of the three 5th order filters of filter bank.
<code>Ipp16s pThirdFltState[5];</code>	The history state of the five 3rd order filters of filter bank.
<code>Ipp16s burstCount;</code>	Burst counter <i>burst_count</i> that counts length of a speech burst, used by VAD hangover addition.
<code>Ipp16s hangCount;</code>	Hang counter <i>hang_counter</i> that is used by VAD hangover addition.

<b>typedef struct{</b>	<b>Description</b>
<code>Ipp16s statCount;</code>	Stationary counter variable <i>stat_count</i> that is used in background noise estimation.
<code>Ipp16s vadReg;</code>	Value that indicates intermediate VAD decision.
<code>Ipp16s complexHigh;</code>	<i>complex_high</i> value that is used as intermediate complex signal decision.
<code>Ipp16s complexLow;</code>	<i>complex_low</i> value that is used as intermediate complex signal decision.
<code>Ipp16s complexHangTimer;</code>	<i>complex_hang_timer</i> that is used as hangover initiator by Complex Activity Estimation.
<code>Ipp16s complexHangCount;</code>	<i>complex_hang_count</i> that is used as hangover counter by VAD hangover addition.
<code>Ipp16s complexWarning;</code>	<i>complex_warning</i> flag.
<code>Ipp16s corrHp;</code>	The high-pass filtered value of <i>best_corr_hp</i> .
<code>Ipp16s pitchFlag;</code>	Pitch flag that indicates the presence of vowel sounds and other periodic signals.
<code>}IppGSMAMRVad1State.</code>	



VAD option 1 history variables

initializationpPrevSignalLevelpPrevSignalSublevelpPrevAverageLevcorrHp

For the detail usage of these history variables, please refer to *ETSI GSM 06.94*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## VAD2\_GSMAMR

*Implements the VAD functionality corresponding to VAD option 2.*

### Syntax

```
IppStatus ippsVAD2_GSMAMR_16s(const Ipp16s* pSrcSpch, IppGSMAMRVad2State*
pValResultVad2State, Ipp16s* pResultVadFlag, Ipp16s ltpFlag);
```

## Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal, of length 160, in Q0.
<i>pValResultVad2State</i>	On input, pointer to the VAD Option 2 history variables. On output, points to the updated VAD Option 2 history variables. The structure <code>IppGSMAMRVad2State</code> is defined below.
<i>pResultVadFlag</i>	Pointer to the Boolean flag <i>VAD_flag</i> . If it is set to "1", it indicates the presence of signals that should be transmitted. If set to "0", there is no signals in this frame needed to be transmitted.
<i>ltpFlag</i>	<i>LTP_flag</i> value in <i>GSM 06.94</i> equation (4.24), which is generated by the comparison of the long-term prediction to a constant threshold <i>LTP_THLD</i> .

## Description

The function `ippsVAD2_GSMAMR` is declared in `ipps.h` file. This function implements the VAD functionality corresponding to VAD option 2 of *ETSI GSM 06.94*. It is used to indicate whether each 20ms frame contains signals that must be transmitted. For example, speech, music, or information tones. The structure `IppGSMAMRVad2State` contains the history variables of VAD Option 2:

<b>typedef struct{</b>	<b>Description</b>
<code>Ipp32s pEngyEstimate[16];</code>	Channel energy estimates vector $\mathbf{E}_{Ch}$ of current half-frame, which is calculated during the previous half-frame according to the equation <i>ETSI GSM 06.94</i> (4.4).
<code>Ipp32s pNoiseEstimate[16];</code>	Channel noise estimate vector $\mathbf{E}_n$ of current half-frame, which is calculated during the previous half-frame according to the equation <i>ETSI GSM 06.94</i> (4.26).
<code>Ipp16s pLongTermEngyDb[16];</code>	Channel average long-term spectral estimate vector $\mathbf{dB}$ , which is calculated during the previous half-frame according to the equation <i>ETSI GSM 06.94</i> (4.20).

<b>typedef struct{</b>	<b>Description</b>
<code>Ipp16s preEmphasisFactor;</code>	Pre-emphasis factor $\geq_p$ , which is used to pre-emphasize the input speech signal according to the equation <i>ETSI GSM 06.94 (4.1)</i> .
<code>Ipp16s updateCount;</code>	<i>update_cnt</i> value used in background noise update decision logic.
<code>Ipp16s lastUpdateCount;</code>	<i>last_update_cnt</i> value used in background noise update decision logic.
<code>Ipp16s hysteresisCount;</code>	<i>hyster_cnt</i> value used in background noise update decision logic.
<code>Ipp16s prevNormShift;</code>	Shifted bits of previous half-frame input speech when normalized to obtain high precision and avoid overflow when doing FFT transformation.
<code>Ipp16s shiftState;</code>	Shift state flag which indicates whether previous half-frame has been shifted or not.
<code>Ipp16s forcedUpdateFlag;</code>	<i>fupdate_flag</i> value which is the result of forced update logic of background noise update decision.
<code>Ipp16s ltpSnr;</code>	Long-term peak signal-noise ratio $\text{SNR}_p$ of previous half-frame, which is used to calibrate the responsiveness of VAD decision.
<code>Ipp16s variabFactor;</code>	Variability factor $\Phi$ of previous half-frame, which indicates the variability of the background noise estimate and is updated according to the equation <i>ETSI GSM 06.94 (4.13)</i> .
<code>Ipp16s negSnrBias;</code>	Negative SNR sensitivity bias factor $\gamma$ of previous half-frame.
<code>Ipp16s burstCount;</code>	Burst counter $b(m)$ used in the 10 ms half-frames's VAD Decision.
<code>Ipp16s hangOverCount;</code>	Hangover counter $h(m)$ used in the 10ms half-frame's VAD Decision.
<code>Ipp32s frameCount;</code>	Half-frame counter.
<b><code>}IppGmrVad2State;</code></b>	



**zero**

Please refer to *ETSI GSM 06.94 VAD Option 2* specification for details.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## EncDTXSID\_GSMAMR

*Extracts parameters for the SID frame.*

---

### Syntax

```
IppStatus ippEncDTXSID_GSMAMR_16s(const Ipp16s* pSrcLspBuffer, const Ipp16s*
pSrcLogEnergyBuffer, Ipp16s* pValResultLogEnergyIndex, Ipp16s*
pValResultDtxLsfRefIndex, Ipp16s* pSrcDstQLsfIndex, Ipp16s* pSrcDstPredQErr,
Ipp16s* pSrcDstPredQErrMR122, Ipp16s sidFlag);
```

### Parameters

<i>pSrcLspBuffer</i>	Pointer to the LSP coefficients of eight consecutive frames marked with VAD = 0, in the length of 80, in Q0.15.
<i>pSrcLogEnergyBuffer</i>	Pointer to the log energy coefficients of eight consecutive frames marked with unvoiced, in the length of 8, in Q5.10.
<i>pValResultLogEnergyIndex</i>	Size of the subframe Pointer to the LSF quantization reference index of last frame. On output, points to the LSF quantization reference index of current DTX frame.
<i>pValResultDtxLsfRefIndex</i>	Pointer to the LSF quantization reference index of last frame. On output, points to the LSF quantization reference index of current DTX frame.
<i>pSrcDstQLsfIndex</i>	Pointer to the LSF residual quantization indices of last frame, in the length of 3. On output, points to the LSF residual quantization indices of current frame, in the length of 3.

<i>pSrcDstPredQErr</i>	Pointer to the fixed gain prediction error of four previous subframes for non-12.2 Kbps modes, in the length of 4, in Q5.10. On output, points to the updated fixed gain prediction error for non 12.2 Kbps modes, in the length of 4, in Q5.10.
<i>pSrcDstPredQErrMR122</i>	Pointer to the fixed gain prediction error of four previous subframes for 12.2 Kbps, in the length of 4, in Q5.10. On output, points to the updated fixed gain prediction error for 12.2 Kbps mode, in the length of 4, in Q5.10.
<i>sidFlag</i>	The SID flag of the current frame. If it is set to 1, the current frame is a SID frame, and the function will extract the LSF and energy parameters. If it is set to 0, the LSF and energy parameters will copy from previous frame.

## Description

The function `ippsEncDTXSID_GSMAMR` is declared in `ippsc.h` file. This function is called only when the current frame is a DTX frame. If the SID flag is on, the function extracts the needed parameters for the SID frame (that is, the LSF quantization parameter and the energy index parameter). If the SID flag is off, no operation is needed, and all the parameters are copied from last frame.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## EncDTXHandler\_GSMAMR

Determines the SID flag of current frame.

### Syntax

```
Ippl6s* ippsEncDTXHandler_GSMAMR_16s(Ippl6s* pValResultHangOverCount,
Ippl6s* pValResultDtxElapsedCount, Ippl6s* pValResultUsedMode, Ippl6s*
pResultSidFlag, Ippl6s vadFlag);
```

## Parameters

<i>pValResultHangOverCount</i>	Pointer to the DTX hangover count. When initialized or reset, it is set to 0. On output, points to the updated DTX hangover count.
<i>pValResultDtxElapsedCount</i>	Pointer to elapsed frame count since last non-DTX frame. When initialized or reset, it is set 0. On output, points to the updated elapsed frame count since last non-DTX frame.
<i>pValResultUsedMode</i>	Pointer to the transmission mode. At the input stage, the mode is one of the bit rate modes ranging from 4.75 Kbps to 12.2 Kbps. At the output stage, this value is either unchanged or set to the DTX frame mode.
<i>pResultSidFlag</i>	Pointer to the output SID flag, "1" indicates a SID frame, and "0" indicates a non-SID frame.
<i>vadFlag</i>	This is the VAD flag of the current frame, if it is set 1, the current frame is marked with voiced, and if it is set to 0, it is marked with unvoiced.

## Description

The function `ippsEncDTXHandler_GSMAMR` is declared in `ippsc.h` file. This function determines the SID flag of current frame, and it determines whether the current frame should use DTX encoding.

1. Update the elapsed frame count since last SID frame:  $DTX\_ElapsedCount = DTX\_ElapsedCount + 1$  (Bounded to  $0 \sim 0x7fff$ ).
2. If the VAD flag of current frame is 1 (voiced frame), the DTX hangover count is set to 7, and this function ends.
3. If the VAD flag of current frame is 0 (unvoiced frame), and the DTX hangover count is 0, the transmission mode of this frame is set to DTX frame mode, and the elapsed frame count since last DTX frame is set to 0, the SID flag is set to 1.
4. If the VAD flag of current frame is 0 (unvoiced frame), but the DTX hangover count is not 0, then decrease DTX hangover count by 1, and if  $DTX\_HangOver\_Count + DTX\_ElapsedCount < 30$  The SID flag is set to 0, and the transmission mode is set to "MRDTX".



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## EncDTXBuffer\_GSMAMR, DecDTXBuffer\_GSMAMR

*Buffer the LSP (or LSF) coefficients and previous log energy coefficients.*

---

### Syntax

```
IppStatus ippEncDTXBuffer_GSMAMR_16s(const Ipp16s* pSrcSpch, const Ipp16s*
pSrcLsp, Ipp16s* pValResultUpdateIndex, Ipp16s* pSrcDstLspBuffer, Ipp16s*
pSrcDstLogEnergyBuffer);
```

```
IppStatus ippDecDTXBuffer_GSMAMR_16s(const Ipp16s* pSrcSpch, const Ipp16s*
pSrcLsf, Ipp16s* pValResultUpdateIndex, Ipp16s* pSrcDstLsfBuffer, Ipp16s*
pSrcDstLogEnergyBuffer);
```

### Parameters

<code>pSrcSpch</code>	Pointer to the input speech signal, in the length of 160, in Q15.0.
<code>pSrcLsp</code>	Pointer to the LSP for this frame, in the length of 10, in Q0.15.
<code>pSrcLsf</code>	Pointer to the LSF coefficients of the current frame, in the length of 10, in Q0.15.
<code>pValResultUpdateIndex</code>	Pointer to the previous memory update index. On output, points to the current memory update index. It is circularly increased between 0 and 7.
<code>pSrcDstLspBuffer</code>	Pointer to the LSP coefficients of eight previous frames, in the length of 80, in Q0.15. On output, points to the LSP coefficients of eight most recent frames (including current frame), in the length of 80, in Q0.15.
<code>pSrcDstLsfBuffer</code>	Pointer to the LSF coefficients of eight previous frames, in the length of 80, in Q0.15.

*pSrcDstLogEnergyBuffer* Pointer to the logarithm energy coefficients of eight previous frames, in the length of 8, in Q5.10. On output, points to the log energy coefficients of eight most recent frames (including current frame), in the length of 8, in Q5.10.

### Description

The functions `ippsEncDTXBuffer_GSMAMR` and `ippsDecDTXBuffer_GSMAMR` are declared in `ippsc.h` file. These functions buffer the LSP (or LSF) coefficients and previous log energy coefficients. These LSPs (or LSFs) and energy coefficients will be used for SID frame to extract necessary parameters. The memory update index indicates which part of the buffer will be updated, and it saves the cost for some memory copy. The log energy is computed as follows:

$$EnLog = \log_2 \left( \frac{1}{N} \sum_{n=0}^{N-1} s^2(n) \right),$$

where  $N$  is the frame length, and  $s$  is the input speech signal.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## Post Processing

### PostFilter\_GSMAMR

*Filters the synthesized speech.*

---

#### Syntax

```
IppStatus ippsPostFilter_GSMAMR_16s(const Ipp16s* pSrcQLpc, const Ipp16s*
pSrcSpch, Ipp16s* pValResultPrevResidual, Ipp16s* pValResultPrevScalingGain,
Ipp16s* pSrcDstFormantFIRState, Ipp16s* pSrcDstFormantIIRState, Ipp16s*
pDstFltSpch, IppSpchBitRate mode);
```

## Parameters

<i>pSrcQLpc</i>	Pointer to the reconstructed LP coefficients, in the length of 44, in Q3.12.
<i>pSrcSpch</i>	Pointer to the start position of the input speech signal for current frame, in the length of 160, in Q15.0.
<i>pValResultPrevResidual</i>	On entry, pointer to the last output of the FIR filter of the formant filter for previous subframe, in Q15.0. It is the input of the tilt compensation filter. On exit, points to the last output of the FIR filter of the formant filter for this subframe (in Q15.0) and is the output of the tilt compensation filter. This value is initialized to 0 and can only be updated by this function.
<i>pValResultPrevScalingGain</i>	Pointer to the scaling factor <i>b</i> of the last signal for the previous subframe, in Q3.12. On output, points to the scaling factor <i>b</i> of the last signal for this subframe, in Q3.12.
<i>pSrcDstFormantFIRState</i>	Pointer to the state of the FIR part of the formant filter, in the length of 10, in Q15.0. On output, points to the updated state of the FIR part of the formant filter, in the length of 10, in Q15.0.
<i>pSrcDstFormantIIRState</i>	Pointer to the state of the IIR part of the formant filter, in the length of 10, in Q15.0. On output, points to the updated state of the IIR part of the formant filter, in the length of 10, in Q15.0.
<i>pDstFltSpch</i>	Pointer to the filtered speech, in the length of 160, in Q15.0.
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_4750</code> and <code>IPP_SPCHBR_12200</code> are valid.

## Description

The function `ippsPostFilter_GSMAMR` is declared in `ippsc.h` file. This function filters the synthesized speech to enhance reconstruction quality.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

`ippStsRangeErr` Indicates an error when the input variable *mode* is out of range.

## AMR Wideband Functions

The Intel IPP functions described in this section if properly combined can be used to construct the Adaptive Multi-Rate Wideband (AMR WB) Speech Codec compliant to 3rd Generation Partnership Project (3GPP) specification TS 26.173: "AMR Wideband Speech Codec; ANSI-C code". The description of AMR WB codec may be found in 3GPP TS 26.190: "AMR Wideband Speech Codec; Transcoding functions". Also it is known as ITU-T G.722.2 AMR WB codec.

The primitives are primarily designed to implement the well-defined, computationally expensive core operations that comprise the codec portion of the AMR WB system. The AMR WB codec comprises an adaptive multi-rate algorithm intended for encoding 7 kHz bandwidth speech signals at the sampling rate of 16 000 samples per second, which results in bit rates for the encoded bit stream of 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s.

The list of the Intel IPP primitives for the AMR WB codec is given in Table 9-6.

**Table 9-6. Intel IPP AMR WB Related Functions**

Function Base Name	Operation
<code>ResidualFilter_AMRWB</code>	Computes the LPC residual
<code>LPCToISP_AMRWB</code>	Performs LP to ISP coefficients conversion
<code>ISPToLPC_AMRWB</code>	Performs ISP to LP coefficients conversion.
<code>ISPToISF_Norm_AMRWB</code>	Performs ISP to ISF coefficients conversion
<code>ISFTToISP_AMRWB</code>	Performs ISF conversion to ISP
<code>OpenLoopPitchSearch_AMRWB</code>	Extracts an open-loop pitch lag estimate from the weighted input speech
<code>HighPassFilterGetSize_AMRWB</code>	Calculates the size of the high-pass filter state memory
<code>HighPassFilterInit_AMRWB</code>	Initializes the state memory of high-pass filter
<code>HighPassFilter_AMRWB</code>	Performs high-pass filtering
<code>HighPassFilterGetDlyLine_AMRWB</code>	Receives the parameters of delay line of high-pass filter
<code>HighPassFilterSetDlyLine_AMRWB</code>	Sets the parameters of delay line of high-pass filter
<code>Preemphasize_AMRWB</code>	Computes pre-emphasis of a speech signal
<code>Deemphasize_AMRWB</code>	Performs de-emphasis filtering
<code>SynthesisFilter_AMRWB</code>	Reconstructs the speech signal from LP coefficients and residuals
<code>VADGetSize_AMRWB</code>	Returns the size of the VAD module state memory
<code>VADInit_AMRWB</code>	Initializes the VAD module state memory
<code>VAD_AMRWB</code>	Performs VAD in AMR WB encoder

Function Base Name	Operation
<a href="#">VADGetEnergyLevel_AMRWB</a>	Gets the vector of energy levels from VAD memory.
<a href="#">AlgebraicCodebookSearch_AMRWB</a>	Performs the fixed (algebraic) codebook search
<a href="#">AlgebraicCodebookDecode_AMRWB</a>	Decodes the fixed (algebraic) codebook indexes
<a href="#">AdaptiveCodebookGainCoeff_AMRWB</a>	Computes the adaptive codebook gain
<a href="#">AdaptiveCodebookSearch_AMRWB</a>	Performs the adaptive codebook search
<a href="#">AdaptiveCodebookDecodeGetSize_AMRWB</a>	Queries the memory length of the adaptive codebook decode module
<a href="#">AdaptiveCodebookDecodeInit_AMRWB</a>	Initializes the adaptive codebook decode module memory
<a href="#">AdaptiveCodebookDecodeUpdate_AMRWB</a>	Updates the adaptive codebook decode module memory
<a href="#">AdaptiveCodebookDecode_AMRWB</a>	Performs the adaptive codebook decoding
<a href="#">ISFQuant_AMRWB</a>	Quantizes the ISF
<a href="#">ISFQuantDecode_AMRWB</a>	Decodes quantized ISFs from the received codebook index
<a href="#">ISFQuantDTX_AMRWB</a>	Quantizes the ISF coefficient vector in case of DTX mode
<a href="#">ISFQuantDecodedTX_AMRWB</a>	Decodes quantized ISFs in case of DTX mode
<a href="#">GainQuant_AMRWB</a>	Quantizes the adaptive codebook gains
<a href="#">DecodeGain_AMRWB</a>	Decodes adaptive and fixed-codebook gains
<a href="#">EncDTXBuffer_AMRWB</a>	Buffers the ISP coefficients and previous logarithm energy coefficients
<a href="#">DecDTXBuffer_AMRWB</a>	Buffers the ISF coefficients and previous logarithm energy coefficients.

The use of these functions is demonstrated in the Intel IPP *GSM/AMR Wideband/G.722.2 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## ResidualFilter\_AMRWB

*Computes the LPC residual*

### Syntax

```
ippStatus ippsResidualFilter_AMRWB_16s_Sfs (const Ipp16s* pSrcLpc, int order,
const Ipp16s* pSrcSpeech, Ipp16s* pDstResidualSignal, int len, int
scaleFactor);
```

## Parameters

<i>pSrcLpc</i>	Pointer to the input LPC.
<i>order</i>	Length of the LPC vector.
<i>pSrcSpeech</i>	Pointer to the input vector [ <i>-order</i> , .., -1, 0, ..., <i>len</i> -1].
<i>pDstResidualSignal</i>	Pointer to the output vector of length [ <i>len</i> ].
<i>len</i>	Length of the vectors.
<i>scaleFactor</i>	Scale factor value.

## Description

This function is declared in `ippsc.h` file. The functionality is the same as for the function [ipp-sResidualFilter\\_G729E](#). The only difference is that the function `ippResidualFilter_AM-RWB` scales the result according to *scaleFactor* value.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>order</i> or <i>len</i> is less than or equal to zero, or when <i>order</i> is greater than <i>len</i> .
<code>ippStsScaleRangeErr</code>	Indicates an error when <i>scaleFactor</i> is negative or greater than 15.

## LPC Analysis Functions

### LPCToISP\_AMRWB

*Performs LP to ISP coefficients conversion.*

---

#### Syntax

```
IppStatus ippSLPCToISP_AMRWB_16s( const Ipp16s* pSrcLpc, Ipp16s* pDstIsp,
const Ipp16s* pSrcPrevIsp);
```

#### Parameters

<i>pSrcLpc</i>	Pointer to the input predictor coefficients.
----------------	--

*pDstIsp* Pointer to the output immittance spectral pairs.  
*pSrcPrevIsp* Pointer to the input previous immittance spectral pairs.

### Description

This function is declared in `ippsc.h` file. The function performs the following steps:

1. Calculates the polynomial coefficients of  $F_1(z)$  and  $F_2(z)$ , using the following recursive relations:

$$i = 0, 1, \dots, 7,$$

$$f1(i) = a_i + a_{m-i}$$

$$f2(i) = a_i - a_{m-i}$$

$$f1(8) = 2a_8$$

$$\text{where } f_2(-2) = f_2(-1) = 1.0.$$

2. Uses Chebyshev polynomials to evaluate  $F_1(z)$  and  $F_2(z)$ . The Chebyshev polynomials are given by:

$$C_1(\omega) = \sum_{i=0}^7 f_1(i) \cos((8-i)\omega) + f_1(8)/2$$

$$C_2(\omega) = \sum_{i=0}^6 f_2(i) \cos((8-i)\omega) + f_2(7)/2$$

$F_1(z)$  and  $F_2(z)$  polynomials are evaluated at 100 points in equally spaced intervals between 0 and  $\pi$  and are checked for sign changes. A sign change indicates the existence of a root in corresponding interval, which is divided then four times to track the root.

3. If all 16 roots needed to determine ISP coefficients are not found, the function returns the previous set of ISP coefficients instead.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## ISPToLPC\_AMRWB

*Performs ISP to LP coefficients conversion.*

### Syntax

```
IppStatus ippsISPToLPC_AMRWB_16s(const Ipp16s* pSrcIsp, Ipp16s* pDstLpc, int len);
```

### Parameters

<i>pSrcIsp</i>	Pointer to the input immittance spectral pairs.
<i>pDstLpc</i>	Pointer to the output predictor coefficients.
<i>len</i>	Number of ISP and LPC.

### Description

This function is declared in `ipps.h` file. The function performs the following steps:

1. Calculates the polynomial coefficients of  $F_1(z)$  and  $F_2(z)$ , using the recursive relations from  $i$  equal to 2 to  $len/2$  by formulas

$$f_1(i) = 2q_{2i-2} * f_1(i-1) + 2f_1(i-2)$$

for  $j=i-1 \dots 2$

$$f_1(j) = f_1(j) - 2q_{2i-2} * f_1(j-1) + 2f_1(j-2) \text{ end}$$

$$f_1(1) = f_1(1) - 2q_{2i-2}$$

with initial values  $f_1(0) = 1$  and  $f_1(1) = -2q_0$ .

The coefficients  $f_2(i)$  are computed similarly by replacing  $q_{2i-2}$  by  $q_{2i-1}$  and  $len/2$  by  $len/2-1$ , and with initial conditions  $f_2(0) = 1$  and  $f_2(1) = -2q_1$ .

2. Once the coefficients  $f_1(z)$  and  $f_2(z)$  are found,  $F_2(z)$  is multiplied by  $1-z^{-2}$  to obtain  $F'_2(z)$ , that is

$$f'_2(i) = f_2(i) - f_2(i-2), \text{ for } i = 2, \dots, len/2 - 1;$$

$$f'_1(i) = f_1(i), \text{ for } i = 0, \dots, len/2.$$

Then  $F'_1(z)$  and  $F'_2(z)$  are multiplied by  $1+q_{len-1}$  and  $1-q_{len-1}$ , respectively.

3. Finally, the function computes the LP coefficients from  $f'_1(i)$  and  $f'_2(i)$  as:



$$a_i = \begin{cases} 0.5 f_1'(i) + 0.5 f_2'(i) , & i = 1, \dots, len/2 - 1 \\ 0.5 f_1'(i) - 0.5 f_2'(i) , & i = len/2 + 1, \dots, len - 1 \\ 0.5 f_1'(len/2) , & i = len/2 \\ q_{len-1} , & i = len \end{cases}$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero, or when <code>len</code> is greater than 20.

## ISPToISF\_Norm\_AMRWB

*Performs ISP to ISF coefficients conversion.*

### Syntax

```
IppStatus ippISPToISF_Norm_AMRWB_16s(const Ipp16s* pSrcIsp, Ipp16s* pDstIsf,
int len);
```

### Parameters

<code>pSrcIsp</code>	Pointer to the ISP input vector of values scaled in range [-1:1].
<code>pDstIsf</code>	Pointer to the ISF output vector of values scaled in range [0:0.5].
<code>len</code>	Number of ISP and ISF.

### Description

This function is declared in `ippsc.h` file. The function `ippISPToISF_Norm_AMRWB` converts the ISP coefficients to ISF coefficients as follows:

$$pDstIsf[i] = \begin{cases} \frac{f_s}{2\pi} \arccos(pSrcIsf[i]), & i = 0, K \text{ } len - 1 \\ \frac{f_s}{4\pi} \arccos(pSrcIsf[i]) & i = len \end{cases}$$

Here  $f_s = 12800$  is the sampling frequency. The scale factor is chosen such that the first ISF coefficient is normalized to the interval  $[0:0.5]$  by multiplying to  $1/\pi$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## ISFToISP\_AMRWB

*Performs ISF conversion to ISP.*

---

### Syntax

```
IppStatus ippISFToISP_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s* pDstIsf, int len);
```

### Parameters

<code>pSrcIsf</code>	Pointer to the ISF input vector of values scaled in range $[0:0.5]$ .
<code>pDstIsf</code>	Pointer to the ISP output vector of values scaled in range $[-1:1]$ .
<code>len</code>	Number of ISP and ISF.

### Description

This function is declared in `ippsc.h` file. It converts ISF to the ISP coefficients as given by:  
 $pDdstIsf[i] = \cos(pSrcIsf[i]), i = 0, \dots, len.$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## Open-loop Pitch Analysis Functions

### OpenLoopPitchSearch\_AMRWB

*Extracts an open-loop pitch lag estimate from the weighted input speech.*

---

#### Syntax

```
Ippl16s* ippsOpenLoopPitchSearch_AMRWB_16s(const Ippl16s* pSrcWgtSpch, const
Ippl16s* pSrcFltWgtSpch, Ippl16s* pPrevMidPitchLag, Ippl16s* pAdaptiveParam,
Ippl16s* pDstOpenLoopLag, Ippl16s* pToneFlag, Ippl16s* pDstOpenLoopGain, Ippl16s*
pSrcDstPrevPitchLag, Ippl16s* pSrcDstLagSwitcher, int len);
```

#### Parameters

<code>pSrcWgtSpch</code>	Pointer to a 320-element vector containing perceptually weighted speech.
<code>pSrcFltWgtSpch</code>	Pointer to a 320-element vector containing filtered through high-pass filter perceptually weighted speech.
<code>pPrevMidPitchLag</code>	Pointer to the median filtered pitch lag of the 5 previous voiced speech half-frames.
<code>pAdaptiveParam</code>	Pointer to the adaptive parameter.
<code>pDstOpenLoopLag</code>	Pointer to a two-element vector of open-loop pitch lags.
<code>pToneFlag</code>	Pointer to the tone flag for the VAD module.
<code>pDstOpenLoopGain</code>	Pointer to a 2-element vector containing optimal open-loop pitch gains.
<code>pSrcDstPrevPitchLag</code>	Pointer to the five-element vector that contains the pitch lags associated with the five most recent voiced speech half-frames.

*pSrcDstLagSwitcher* Switches lag weighting on and off.  
*len* Length of the frame.

### Description

This function is declared in `ippsc.h` file. The open-loop pitch (OLP) search function `ippsOpen-LoopPitchSearch_AMRWB` extracts a pitch estimate from a weighted version of the input speech.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.  
`ippStsSizeErr` Indicates an error when *len* is less than or equal to zero.

## Filtering Functions

This section describes functions that perform filtering operations specific to the AMRWB Speech Codec.

### HighPassFilterGetSize\_AMRWB

*Calculates the size of the high-pass filter state memory.*

---

#### Syntax

```
IppStatus ippsHighPassFilterGetSize_AMRWB_16s (int order, int* pDstSize);
```

#### Parameters

*order* The order of high-pass filter (orders of 2 or 3 are currently supported).  
*pDstSize* Pointer to the output value of the memory size.

#### Description

This function is declared in `ippsc.h` file. It calculates the size of memory needed for proper operation of the high-pass filter of the given *order*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsRangeErr</code>	Indicates an error when <i>order</i> is not equal to either 2 or 3.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDstSize</i> pointer is <code>NULL</code> .

## HighPassFilterInit\_AMRWB

*Initializes the state memory of high-pass filter.*

### Syntax

```
IppStatus ippHighPassFilterInit_AMRWB_16s(Ipp16s* pFilterCoeffA, Ipp16s*
pFilterCoeffB, int order, IppsHighPassFilterState_AMRWB_16s* pState);
```

### Parameters

<i>order</i>	The order of high-pass filter; orders of 2 and 3 are currently supported.
<i>pFilterCoeffA</i>	Pointer to the vector of <i>order</i> size containing the IIR part of the filter coefficients .
<i>pFilterCoeffB</i>	Pointer to the vector of <i>order</i> size containing the FIR part of the filter coefficients.
<i>pState</i>	Pointer to the memory supplied for filtering.

### Description

This function is declared in `ippsc.h` file. It sets the coefficients and initializes the state memory of the high-pass filter.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>order</i> is not equal to either 2 or 3.

## HighPassFilter\_AMRWB

*Performs high-pass filtering.*

---

### Syntax

```
IppStatus ippsHighPassFilter_AMRWB_16s_Sfs (const Ipp16s* pSrc, Ipp16s* pDst,
int len, IppsHighPassFilterState_AMRWB_16s* pState, int scaleFactor);
```

```
IppStatus ippsHighPassFilter_AMRWB_16s_ISfs (Ipp16s* pSrcDst, int len,
IppsHighPassFilterState_AMRWB_16s* pState, int scaleFactor);
```

```
IppStatus ippsHighPassFilter_Direct_AMRWB_16s (const Ipp16s* pSrcCoeff, const
Ipp16s* pSrc, Ipp16s* pDst, int len, int borderMode);
```

### Parameters

<i>pSrc</i>	Pointer to the vector containing input signal.
<i>pDst</i>	Pointer to the vector containing filtered signal.
<i>pSrcDst</i>	Pointer to the input/output vector.
<i>pState</i>	Pointer to the memory supplied for filtering.
<i>scaleFactor</i>	Scale factor of the input signal.
<i>pSrcCoeff</i>	Pointer to a 2-element vector containing the coefficients of the filter.
<i>len</i>	Length of the input and output vectors.
<i>borderMode</i>	if <i>borderMode</i> = 0 then <i>pSrc</i> [-1] and <i>pSrc</i> [ <i>len</i> ] are equal to zero; otherwise these values must be given.

### Description

These functions are declared in `ipps.h` file.

**ippsHighPassFilter\_AMRWB\_16s.** This function performs the high-pass filtering of the input signal by the following transfer function:

$$H_h(z) = \frac{b_0 + b_1 z^{-1} \dots + b_{order-1} z^{-order}}{a_0 + a_1 z^{-1} \dots + a_{order-1} z^{-order}}$$

Currently, only  $a_0 = 1.0$  is supported. The scale factor is used to update the filter memory according to the input signal.

**ippHighPassFilter\_Direct\_AMRWB\_16s**. This function performs the high-pass filtering of the input signal by the following transfer function:

$$H_h(z) = \frac{a_0}{1 + a_1 z^{-1}}$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## HighPassFilterGetDlyLine\_AMRWB

*Receives the parameters of delay line of high-pass filter.*

### Syntax

```
IppStatus ippHighPassFilterGetDlyLine_AMRWB_16s(const
IppHighPassFilterState_AMRWB_16s* pState, Ipp16s* pDlyLine, Ipp32s order);
```

### Parameters

<code>pState</code>	Pointer to the memory supplied for filtering.
<code>pDlyLine</code>	Pointer to the vector (6-element size for order 2, or 9-element size for order 3) containing the filter memory.
<code>order</code>	The order of high-pass filter; possible value is 2 or 3.

### Description

The function `ippHighPassFilterGetDlyLine_AMRWB` is declared in `ippsc.h` file. This function gets the parameters of the delay line of the high-pass filter structure. The layout of the returned filter memory is follows:

$Yh_2, Yl_2, Yh_1, Yl_1, X_0, X_1$  - for the high-pass filter of order 2.

$Yh_3, Yl_3, Yh_2, Yl_2, Yh_1, Yl_1, X_0, X_1, X_2$  - the high-pass filter of order 3 ,

where  $X_i$  - the memory of the FIR filter part, and  $Yh_i, Yl_i$  - the memories of the IIR filter part.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>order</i> is not equal 2 or 3.

## HighPassFilterSetDlyLine\_AMRWB

*Sets the parameters of delay line of high-pass filter.*

---

### Syntax

```
IppStatus ippHighPassFilterSetDlyLine_AMRWB_16s(const Ipp16s* pDlyLine,
IppHighPassFilterState_AMRWB_16s* pState, Ipp32s order);
```

### Parameters

<i>pDlyLine</i>	Pointer to the vector (6-element size for order 2, or 9-element size for order 3) containing the filter memory.
<i>pState</i>	Pointer to the memory supplied for filtering.
<i>order</i>	The order of high-pass filter; possible value is 2 or 3.

### Description

The function `ippHighPassFilterSetDlyLine_AMRWB` is declared in `ippsc.h` file. This function updates *pState* structure by *pDlyLine* vector with the parameters of the delay line of the high-pass filtering. The layout of the *pDlyLine* vector is the same as in the description of the function [ippSNR\\_AMRWB](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>order</i> is not equal 2 or 3.



## Preemphasize\_AMRWB

*Computes pre-emphasis of a speech signal.*

---

### Syntax

```
IppStatus ippsPreemphasize_AMRWB_16s_ISfs (Ipp16s gamma, Ipp16s* pSrcDst,
int len, int scaleFactor, Ipp16s* pMem);
```

### Parameters

<i>gamma</i>	Filter coefficient.
<i>pSrcDst</i>	Pointer to the input/output vector.
<i>scaleFactor</i>	Scale factor for the result.
<i>len</i>	Length of the input/output vector.
<i>pMem</i>	Pointer to the filter memory of length 1.

### Description

This function is declared in `ippsc.h` file. It computes pre-emphasis of the input speech according to the difference signal pre-emphasis equation:

$$H(z) = 1 - \gamma z^{-1}$$

The memory value `pMem[0]` is updated by `pSrcDst[n-1]`. For proper use of this function in AMR WB codec, the memory value must be initialized to zero in the user program.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrcDst</code> or <code>pMem</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.
<code>ippStsScaleRangeErr</code>	Indicates an error when <code>scaleFactor &lt; 0</code> or <code>scaleFactor &gt; 15</code> .

## Deemphasize\_AMRWB

*Performs de-emphasis filtering.*

---

### Syntax

```

IppStatus ippsDeemphasize_AMRWB_NR_16s_I(Ipp16s gamma, Ipp16s* pSrcDst,
Ipp16s len, Ipp16s* pMem);

IppStatus ippsDeemphasize_AMRWB_32s16s(Ipp16s gamma, const Ipp32s* pSrc,
Ipp16s* pDst, int len, Ipp16s* pMem);

```

### Parameters

<i>gamma</i>	De-emphasis factor.
<i>pSrc</i>	Pointer to the input vector.
<i>pDst</i>	Pointer to the output vector.
<i>pSrcDst</i>	Pointer to the input/output vector.
<i>len</i>	Length of the input/output vector.
<i>pMem</i>	Pointer to the filter memory element.

### Description

This function is declared in `ippsc.h` file. It performs de-emphasis of the input synthesized signal by filtering it through the filter with the following transfer function:

$$H(z) = 1 / (1 - \gamma * z^{-1})$$

The initial memory of the filter must be set to zero. The memory value `pMem[0]` is updated by `pDst[len - 1]` or `pSrcDst[len - 1]`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## SynthesisFilter\_AMRWB

*Reconstructs the speech signal from LP coefficients and residuals.*

---

### Syntax

```
IppStatus ippsSynthesisFilter_AMRWB_16s32s_I(const Ipp16s* pSrcLpc, int
order, const Ipp16s* pSrcExc, Ipp32s* pSrcDstSignal, int len);
```

### Parameters

<i>pSrcLpc</i>	Pointer to the LP coefficients vector $a_0, a_1, \dots, a_{order}$
<i>order</i>	Order of LP filter.
<i>pSrcExc</i>	Pointer to the excitation vector.
<i>pSrcDstSignal</i>	Pointer to the synthesized and updated speech.
<i>len</i>	Length of the filter.

### Description

This function is declared in `ippsc.h` file. It computes the filter given by:

$$H(z) = \frac{1}{a_0 + \sum_{i=1}^{order} a_i z^{-i}}$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> or <i>order</i> is less than or equal to zero.

## Discontinuous Transmission (DTX) Functions

### VADGetSize\_AMRWB

*Calculates the size of the VAD module state memory.*

---

#### Syntax

```
IppStatus ippsVADGetSize_AMRWB_16s (int* pDstSize);
```

#### Parameters

*pDstSize*                      Pointer to the output value of the memory size.

#### Description

This function is declared in `ippsc.h` file. It calculates the size of memory needed for proper operation of the VAD module in the AMR WB codec.

#### Return Values

`IppStsNoErr`                      Indicates no error.  
`IppStsNullPtrErr`                Indicates an error when the *pDstSize* pointer is NULL.

### VADInit\_AMRWB

*Initializes the VAD module state memory.*

---

#### Syntax

```
IppStatus ippsVADInit_AMRWB_16s(IppsVADState_AMRWB_16s* pState);
```

#### Parameters

*pState*                          Pointer to the VAD memory.

## Description

This function is declared in `ippsc.h` file. It initializes the state of VAD (Voice Activity Decision) module of the AMR WB codec. The structure `IppsVADState_AMRWB_16s` contains the VAD history that must be initialized prior to the use of the function `ippsVAD_AMRWB`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is <code>NULL</code> .

## VAD\_AMRWB

*Performs VAD in AMR WB encoder.*

---

## Syntax

```
IppStatus ippsVAD_AMRWB_16s(const Ipp16s* pSrcSpch, IppsVADState_AMRWB_16s*
pSrcDstVadState, Ipp16s* pToneFlag, Ipp16s* pVadFlag);
```

## Parameters

<code>pSrcSpch</code>	Pointer to the input speech signal, of length 320.
<code>pSrcDstVadState</code>	Pointer to the VAD memory.
<code>pToneFlag</code>	Pointer to the output tone flag.
<code>pVadFlag</code>	Pointer to the VAD flag of this frame.

## Description

This function is declared in `ippsc.h` file. The function `ippsVAD_AMRWB` implements the Voice Activity Decision (VAD) functionality of the AMR WB encoder. This function is used to indicate whether the input speech frame contains active speech or some other audio signal such as silence or music. The structure `IppsVADState_AMRWB_16s` contains the VAD history and is updated in the function. This structure must be initialized in advance by using the function `ippsVADInit_AMRWB`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## VADGetEnergyLevel\_AMRWB

Gets the vector of energy levels from VAD memory.

### Syntax

```
IppStatus ippSVADGetEnergyLevel_AMRWB_16s(const IppsVADState_AMRWB_16s*  
pState, Ipp16s* pEnergyLevel);
```

### Parameters

<i>pState</i>	Pointer to the VAD memory.
<i>pEnergyLevel</i>	Pointer to the energy level vector [12].

### Description

The function `ippSVADGetEnergyLevel_AMRWB` is declared in `ippsc.h` file. The function `ippSVAD_AMRWB` produces signal energy in the 12 non-uniform bands over the frequency range from 0 to 6.4 kHz for 256-sample frame. And the function `ippSVADGetEnergyLevel_AMRWB` returns normalized energy levels which are produced by dividing the energy level form each band by the width of this band in Hz. The lower index refers to the lower sub band.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## Codebook Search Functions

These functions perform algebraic and adaptive codebook search and decoding specific for the AMR WB codec.

## AlgebraicCodebookSearch\_AMRWB

Performs the fixed (algebraic) codebook search.

### Syntax

```
IppStatus ippsAlgebraicCodebookSearch_AMRWB_16s(const Ipp16s* pSrcFixedTarget,
const Ipp16s* pSrcLtpResidual, Ipp16s* pSrcDstImpulseResponse, Ipp16s*
pDstFixedVector, Ipp16s* pDstFltFixedVector, IppSpchBitRate mode, Ipp16s*
pDstIndex);
```

### Parameters

<i>pSrcFixedTarget</i>	Pointer to the target vector.
<i>pSrcLtpResidual</i>	Pointer to the long term prediction residual.
<i>pSrcDstImpulseResponse</i>	Pointer to the impulse response of the weighted synthesis filter; <i>pSrcDstImpulseResponse</i> [- <i>L_subfr</i> ..-1] must be set to zero.
<i>pDstFixedVector</i>	Pointer to the innovative codebook.
<i>pDstFltFixedVector</i>	Pointer to the filtered fixed codebook excitation.
<i>mode</i>	Coder mode.
<i>pDstIndex</i>	Pointer to the indexes of the pulses.

### Description

This function is declared in `ippsc.h` file. It performs the following steps:

1. Computes backward filtered target vector **d** by

$$d(n) = \sum_{i=n}^{63} x_2(i) \times h(i-n), \quad 0 \leq n \leq 63$$

where  $x_2(n)$  is the fixed target signal used for fixed codebook search.

2. Computes the signal  $b(n)$ , which is used for presetting the pulse amplitudes, as:

$$b(n) = \frac{\sum_{i=0}^{63} d(i)d(i)}{\sum_{i=0}^{63} res_{LTP}(i)res_{LTP}(i)}res_{LTP}(n) + \alpha d(n) , \quad 0 \leq n \leq 63$$

The scaling factor  $\alpha$  controls the amount of dependence of the reference signal on  $d(n)$ , it is lowered as the bit rate is increased. Here  $\alpha=2$  for 6.6 and 8.85 modes;  $\alpha=1$  for 12.65, 14.25 and 15.85 modes;  $\alpha=0.8$  for 18.25 mode;  $\alpha=75$  for 19.25 mode; and  $\alpha=0.5$  for 23.05 and 23.85 modes.

3. Calculates the symmetric Toepliz matrix  $\Phi$  for each mode. The element of this matrix is computed by

$$\Phi(i,j) = \sum_{n=j}^{63} h(n-i)h(n-j) , \quad i = 0, ..., 63, j = 0, ..., 63$$

4. Searches signed pulses in appropriate positions and encodes them. The pulse positions for different modes are in the tables below:

Modes 23.05 and 23.85:

Track	Pulse	Positions
1	$i_0, i_4, i_8, i_{12}, i_{16}, i_{20}$	0, 4, 8, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9, i_{13}, i_{17}, i_{21}$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61



Track	Pulse	Positions
3	$i_2, i_6, i_{10}, i_{14}, i_{18}, i_{22}$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	$i_3, i_7, i_{11}, i_{15}, i_{19}, i_{23}$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Modes 19.85:

Track	Pulse	Positions
1	$i_0, i_4, i_8, i_{12}, i_{16},$	0, 4, 8, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9, i_{13}, i_{17},$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	$i_2, i_6, i_{10}, i_{14},$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	$i_3, i_7, i_{11}, i_{15},$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Modes 18.25:

Track	Pulse	Positions
1	$i_0, i_4, i_8, i_{12},$	0, 4, 8, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9, i_{13},$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	$i_2, i_6, i_{10}, i_{14},$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62

Track	Pulse	Positions
4	$i_3, i_7, i_{11}, i_{15},$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Modes 15.85:

Track	Pulse	Positions
1	$i_0, i_4, i_8,$	0, 4, 8, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9,$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	$i_2, i_6, i_{10},$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	$i_3, i_7, i_{11},$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Modes 14.25:

Track	Pulse	Positions
1	$i_0, i_4, i_8,$	0, 4, 8, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5, i_9,$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	$i_2, i_6,$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	$i_3, i_7,$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Modes 12.65:

Track	Pulse	Positions
1	$i_0, i_4,$	0, 4, 8, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
2	$i_1, i_5,$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	$i_2, i_6,$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	$i_3, i_7,$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Modes 8.85:

Track	Pulse	Positions
1	$i_0,$	0, 4, 8, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
2	$i_1,$	1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
3	$i_2,$	2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62
4	$i_3,$	3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

Modes 6.60:

Track	Pulse	Positions
1	$i_0$	0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62
2	$i_1$	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## AlgebraicCodebookDecode\_AMRWB

*Decodes the fixed (algebraic) codebook indexes.*

### Syntax

```
IppStatus ippAlgebraicCodebookDecode_AMRWB_16s (const Ipp16s* pSrcIdxs,  
Ipp16s* pDstFixedCode, IppSpchBitRate mode);
```

### Parameters

<i>pSrcIdxs</i>	Algebraic codebook indexes.
<i>pDstFixedCode</i>	Pointer to the algebraic code vector of length 64.
<i>mode</i>	Coder mode.

### Description

This function is declared in `ippsc.h` file. The function decodes the fixed codebook vector from the received fixed codebook index.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not an element of the enumerated type <i>IppSpchBitRate</i> appropriate to AMR WB.

## AdaptiveCodebookGainCoeff\_AMRWB

*Computes the adaptive codebook gain.*

### Syntax

```
IppStatus ippAdaptiveCodebookGainCoeff_AMRWB_16s(const Ipp16s*
pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, Ipp16s* pAdptGainCoeffs,
Ipp16s* pResultAdptGain, int len);
```

### Parameters

<i>pSrcAdptTarget</i>	Pointer to the adaptive-codebook vector.
<i>pSrcFltAdptVector</i>	Pointer to the input filtered adaptive-codebook vector.
<i>pResultAdptGain</i>	Pointer to the adaptive-codebook gain in the length of 1.
<i>pAdptGainCoeffs</i>	Pointer to the output vector in the length 4; represents the adaptive-codebook gain as a fraction.
<i>len</i>	Length of vectors.

### Description

This function `ippAdaptiveCodebookGainCoeff_AMRWB` is declared in `ippsc.h` file. The adaptive codebook gain is found by:

$$g_p = \frac{\sum_{n=0}^{len-1} x(n)y(n)}{\sqrt{\sum_{n=0}^{len-1} y(n)y(n)}}$$

bounded by  $0 \leq g_p \leq 1.2$ .

Additionally, the function returns the gain in a different representation given by the following formula:

$$g_p = \frac{c_{xy} 2^{exp_{xy}}}{c_{yy} 2^{exp_{yy}}}, 1/2 \leq |c_{xy}|, |c_{yy}| \leq 1$$

The mantissas  $C_{xy}$ ,  $C_{yy}$  and exponents  $exp_{xy}$ ,  $exp_{yy}$  for both the normalized denominator and normalized numerator are returned in the *pAdptGainCoeffs* vector:

*pAdptGainCoeffs*[0] =  $C_{yy}$

*pAdptGainCoeffs*[1] =  $exp_{yy}$

*pAdptGainCoeffs*[2] =  $C_{xy}$

*pAdptGainCoeffs*[3] =  $exp_{xy}$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

## AdaptiveCodebookSearch\_AMRWB

*Performs the adaptive codebook search.*

---

### Syntax

```
IppStatus ippAdaptiveCodebookSearch_AMRWB_16s(const Ipp16s* pSrcAdptTarget,
const Ipp16s* pSrcImpulseResponse, const Ipp16s* pSrcOpenLoopLag, Ipp16s*
pPitchLag, Ipp16s* pPitchLagBounds, Ipp16s* pSrcDstExcitation, Ipp16s*
pFracPitchLag, Ipp16s* pAdptIndex, int subFrame, IppSpchBitRate mode);
```

## Parameters

<i>pSrcAdptTarget</i>	Pointer to the 64-element adaptive target signal vector.
<i>pSrcImpulseResponse</i>	Pointer to the 64-element impulse response of the weighted synthesis filter.
<i>pSrcOpenLoopLag</i>	Pointer to a two-element vector of the open-loop pitch lags.
<i>pPitchLag</i>	Pointer to the output previous integral pitch lag.
<i>pPitchLagBounds</i>	Pointer to the bounds of the output previous integral pitch lag.
<i>pSrcDstExcitation</i>	Pointer to the input/output 321-element excitation vector.
<i>pFracPitchLag</i>	Pointer to the output fractional pitch lag obtained during the adaptive codebook search.
<i>pAdptIndex</i>	Pointer to the coded closed-loop pitch index.
<i>subFrame</i>	Subframe number.
<i>mode</i>	Coder mode.

## Description

This function is declared in `ippsc.h` file. The function performs the adaptive codebook search. The adaptive codebook search consists of a closed-loop pitch search followed by computation of an adaptive excitation vector. The adaptive excitation vector is computed by interpolating the past excitation at the fractional pitch lag obtained during the closed-loop pitch search. The adaptive codebook is searched on every subframe.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> is less than 0 or greater than 3.

## AdaptiveCodebookDecodeGetSize\_AMRWB

*Queries the memory length of the adaptive codebook decode module.*

---

### Syntax

```
IppStatus ippsAdaptiveCodebookDecodeGetSize_AMRWB_16s(int* pDstSize);
```

### Parameters

<i>pDstSize</i>	Pointer to the output parameter to store the adaptive codebook decode module length in bytes.
-----------------	---

### Description

This function is declared in `ippsc.h` file. The function reports the size of memory needed for proper operation of the adaptive codebook decode module.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDstSize</i> pointer is <code>NULL</code> .

## AdaptiveCodebookDecodeInit\_AMRWB

*Initializes the adaptive codebook decode module memory.*

---

### Syntax

```
IppStatus ippsAdaptiveCodebookDecodeInit_AMRWB_16s(  
IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

### Parameters

<i>pState</i>	Pointer to the adaptive codebook decode module memory.
---------------	--

### Description

This function is declared in `ippsc.h` file. The function initializes the adaptive codebook decode module memory.



### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.

## AdaptiveCodebookDecodeUpdate\_AMRWB

*Updates the adaptive codebook decode module memory.*

---

### Syntax

```
IppStatus ippsAdaptiveCodebookDecodeUpdate_AMRWB_16s(Ipp16s valIntPitchGain,  
Ipp16s valPitchLag, IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

### Parameters

<i>valPitchGain</i>	Pitch gain.
<i>valIntPitchLag</i>	Pointer to the integral pitch lag.
<i>pState</i>	Pointer to the adaptive codebook decode module memory.

### Description

This function is declared in `ippsc.h` file. The function updates the adaptive codebook decode module memory with the given pitch lag and pitch gain.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.

## AdaptiveCodebookDecode\_AMRWB

*Performs the adaptive codebook decoding.*

---

### Syntax

```
IppStatus ippsAdaptiveCodebookDecode_AMRWB_16s(int valAdptIndex, Ipp16s*
pResultFracPitchLag, Ipp16s* pSrcDstExcitation, Ipp16s* pPitchLag, Ipp16s*
pPitchLagBounds, int subFrame, int bfi, int unusableFrame, IppSpchBitRate
mode, IppsAdaptiveCodebookDecodeState_AMRWB_16s* pState);
```

### Parameters

<i>valAdptIndex</i>	Adaptive codebook index.
<i>pResultFracPitchLag</i>	Pointer to the fractional pitch lag obtained during the adaptive codebook search.
<i>pSrcDstExcitation</i>	Pointer to the 321-element excitation vector.
<i>pPitchLag</i>	Pointer to the integral pitch lag.
<i>pPitchLagBounds</i>	Pointer to the previous integral pitch lag bounds.
<i>subFrame</i>	Subframe number.
<i>bfi</i>	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
<i>unusableFrame</i>	Value "0" signifies a lost frame with content that can not be used at all.
<i>mode</i>	Encoding mode.
<i>pState</i>	Pointer to the memory of the adaptive codebook decode module.

### Description

This function is declared in `ippsc.h` file. It performs the following steps:

1. If no errors are detected on the current frame, the function extracts integer and fractional pitch lags from the adaptive codebook indices.
2. If errors are detected, the function recovers the integer pitch either from the previous integer pitch or the LTP-lag. See clause 6.2.3.4 in 3GPP TS 26.191 V5.1.0.

3. To obtain the adaptive codebook vector, the function `ippAdaptiveCodebookDecode_AMRWB` applies the same adaptive codebook computation procedure as described in the function `ippAdaptiveCodebookSearch_AMRWB`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrame</i> is less than 0 or greater than 3.

## Quantization Functions

This section describes functions that perform quantization specific to the AMRWB speech codec.

### ISFQuant\_AMRWB

*Quantizes the ISF.*

#### Syntax

```
IppStatus ippISFQuant_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s*
pSrcDstResidual, Ipp16s* pDstQIsf, Ipp16s* pDstQIsfIndex, IppSpchBitRate
mode);
```

#### Parameters

<i>pSrcIsf</i>	Pointer to the unquantized 16-element ISF vector.
<i>pSrcDstResidual</i>	Pointer to the 16-element quantized ISF residual from the previous frame.
<i>pDstQIsf</i>	Pointer to the quantized 16-element ISF vector.
<i>pDstQIsfIndex</i>	Pointer to the 7-element vector of quantized ISF indices. For 6.60 Kbps frames, only the first five elements contain valid indices; for all other bit rates, all seven elements contain valid data.
<i>mode</i>	Coder mode.

## Description

This function is declared in `ippsc.h` file. It applies a first order moving-average (MA) prediction and quantifies the residual ISF vector using a combination of split vector quantization (SVQ) and multistage vector quantization (MSVQ).

The prediction and quantization are performed as follows. Let  $z(n)$  denotes the mean-removed ISF vector at frame  $n$ .

The prediction residual vector  $r(n)$  is given by:

$$r(n) = z(n) - p(n)$$

where  $p(n)$  is the predicted LSF vector at frame  $n$ . First order moving-average (MA) prediction is used where:

$$p(n) = 1/3 * (n-1),$$

where  $(n-1)$  is the quantized residual vector at the past frame.

The ISF residual vector  $r$  is quantized using split-multistage vector quantization S-MSVQ. The vector is split into two subvectors  $r_1(n)$  and  $r_2(n)$  of dimensions 9 and 7, respectively. The two subvectors are quantized in two stages. In the first stage  $r_1(n)$  is quantized with 8 bits and  $r_2(n)$  with 8 bits.

For 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbit/s modes, the quantization error vectors are split in the next stage into 3 and 2 subvectors, respectively.

For 6.60 kbit/s mode, the quantization error vectors

$$r_i^{(2)} = r - \hat{r}_i, i = 1, 2.$$

are split in the next stage into 2 and 1 subvectors, respectively. A squared error ISF distortion measure is used in the quantization process. In general, for an input ISF or error residual subvector  $r_i$ ,  $i = 1, 2$  a quantized vector at index  $k$ ,  $\hat{r}_i^k$  the quantization is performed by finding the index  $k$  which minimizes

$$E = \sum_{i=m}^n \left[ r_i - \hat{r}_i^k \right]^2$$

where  $m$  and  $n$  are the first and last elements of the subvector.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## ISFQuantDecode\_AMRWB

*Decodes quantized ISFs from the received codebook index.*

---

### Syntax

```
IppStatus ippISFQuantDecode_AMRWB_16s(const Ipp16s* pSrcIdxs, Ipp16s*
pDstQntIsf, Ipp16s* pSrcDstResidual, const Ipp16s* pSrcPrevQntIsf, Ipp16s*
pSrcDstIsfMemory, int bfi, IppSpchBitRate mode);
```

### Parameters

<i>pSrcIdxs</i>	Pointer to the seven-element vector containing codebook indices of the quantized LSPs.
<i>pSrcDstResidual</i>	Pointer to the 16-element quantized ISF residual from the previous frame.
<i>pSrcPrevQntIsf</i>	Pointer to the 16-element quantized ISF vector from the previous frame.
<i>pSrcDstIsfMemory</i>	Pointer to the 64-element vector containing four subframe ISF sets.
<i>pDstQntIsf</i>	Pointer to a 16-element destination vector containing quantized ISF in frequency domain (0..0.5).
<i>bfi</i>	Bad frame indicator: value "0" signifies a good frame, all other values signify a bad frame.
<i>mode</i>	Coder mode.

### Description

This function is declared in `ippsc.h` file. The function decodes quantized ISFs from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized ISFs from previous quantized ISFs using linear interpolation.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## ISFQuantDTX\_AMRWB

*Quantizes the ISF coefficient vector in case of DTX mode.*

---

### Syntax

```
IppStatus ippISFQuantDTX_AMRWB_16s(const Ipp16s* pSrcIsf, Ipp16s* pDstQntIsf,
Ipp16s* pDstIdxs);
```

### Parameters

<i>pSrcIsf</i>	Pointer to the unquantized 16-element ISF vector in the frequency domain (0..0.5) .
<i>pDstQntIsf</i>	Pointer to the 16-element quantized ISF vector.
<i>pDstIdxs</i>	Pointer to the 5-element vector quantization indices.

### Description

This function is declared in `ippsc.h` file. The function quantizes averaged ISF coefficient vector using the comfort noise ISF quantization tables.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## ISFQuantDecodeDTX\_AMRWB

*Decodes quantized ISFs in case of DTX mode.*

---

### Syntax

```
IppStatus ippsISFQuantDecodeDTX_AMRWB_16s(const Ipp16s* pSrcIdxs, Ipp16s* pDstQntIsf);
```

### Parameters

<i>pSrcIdxs</i>	Pointer to the 5-element vector quantization indices.
<i>pDstQntIsf</i>	Pointer to the 16-element ISF vector in the frequency domain (0..0.5) .

### Description

This function is declared in `ippsc.h` file. The function decodes quantized ISFs from the received codebook index in case of DTX mode.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## GainQuant\_AMRWB

*Quantizes the adaptive codebook gains.*

---

### Syntax

```
IppStatus ippsGainQuant_AMRWB_16s(const Ipp16s* pSrcAdptTarget, const Ipp16s* pSrcFltAdptVector, int valFormat, const Ipp16s* pSrcFixedVector, const Ipp16s* pSrcFltFixedVector, const Ipp16s* pSrcCorr, Ipp16s* pSrcDstEnergyErr, Ipp16s* pSrcDstPitchGain, int* pDstCodeGain, int valClipFlag, Ipp16s* pDstQGainIndex, int lenSrc, IppSpchBitRate mode);
```

### Parameters

<i>pSrcAdptTarget</i>	Pointer to the input target vector $x(n)$ .
<i>pSrcFltAdptVector</i>	Pointer to the input filtered adaptive codebook vector $y(n)$ .

<i>valFormat</i>	Format of the <i>pSrcAdptTarget</i> and <i>pSrcFltAdptVector</i> vectors.
<i>pSrcFixedVector</i>	Pointer to the input pre-filtered codebook contribution <i>c(n)</i> .
<i>pSrcFltFixedVector</i>	Pointer to the input filtered codebook vector <i>z(n)</i> .
<i>pSrcCorr</i>	Pointer to the vector of correlations between the <i>pSrcAdptTarget</i> , <i>pSrcFltAdptVector</i> , <i>pSrcFltFixedVector</i> vectors.
<i>pSrcDstEnergyErr</i>	Pointer to the input/output energy error vector for 4 previous subframes.
<i>pSrcDstPitchGain</i>	Pointer to the input/output pitch gain.
<i>pDstCodeGain</i>	Pointer to the output code gain.
<i>valClipFlag</i>	If <i>valClipFlag</i> = 1 , then limit gain pitch to 1.0.
<i>pDstQGainIndex</i>	Pointer to the output codebook indexes found.
<i>lenSrc</i>	Length of the input vectors.
<i>mode</i>	Encoding mode.

## Description

This function is declared in `ippsc.h` file. The adaptive codebook gain (pitch gain) and the fixed (algebraic) codebook gain are vector quantized using a 6-bit codebook for modes 8.85 and 6.60 kbit/s and using a 7-bit codebook for all the other modes.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>lenSrc</i> is less than or equal to zero.
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .



## DecodeGain\_AMRWB

*Decodes adaptive and fixed-codebook gains.*

### Syntax

```

IppStatus ippsDecodeGain_AMRWB_16s(int valQIndex, Ipp32s valEnergy, Ipp16s*
pDstPitchGain, int* pDstCodeGain, int bfi, int prevBfi, Ipp16s*
pSrcDstPastEnergy, Ipp16s* pPrevCodeGain, Ipp16s* pSrcDstPastCodeGain,
IppSpchBitRate mode);

```

### Parameters

<i>valQIndex</i>	Index of quantization.
<i>valEnergy</i>	Pointer to the input filtered adaptive codebook vector $y(n)$ .
<i>pDstPitchGain</i>	Pointer to the decoded pitch gain.
<i>pDstCodeGain</i>	Pointer to the decoded code gain.
<i>bfi</i>	Bad frame indicator.
<i>prevBfi</i>	Bad frame indicator of the previous frame.
<i>pSrcDstPastEnergy</i>	Past quantized energies.
<i>pPrevCodeGain</i>	Past code gain.
<i>pSrcDstPastCodeGain</i>	Past code gain for frame erasures.
<i>mode</i>	Decoding mode.

### Description

The function `ippsDecodeGain_AMRWB` is declared in `ippsc.h` file.

If there are no errors detected in the frame, the function computes predicted and fixed codebook gains.

In case of frame erasure, the gains are the attenuated versions of the previous gains.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>mode</i> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## EncDTXBuffer\_AMRWB

*Buffers the ISP coefficients and previous logarithm energy coefficients.*

---

### Syntax

```
IppStatus ippsEncDTXBuffer_AMRWB_16s(const Ipp16s* pSrcSpch, const Ipp16s* pSrcIsp, Ipp16s* pUpdateIndex, Ipp16s* pSrcDstIspBuffer, Ipp16s* pSrcDstLogEnergyBuffer, IppSpchBitRate mode);
```

### Parameters

<i>pSrcSpch</i>	Pointer to the input speech signal of length 256 .
<i>pSrcIsp</i>	Pointer to the ISP for this frame, of length 16.
<i>pUpdateIndex</i>	Holds the previous memory update index. On output, it contains the current memory update index. Circularly increased between 0 and 7.
<i>pSrcDstIspBuffer</i>	Pointer to the ISP coefficients of eight previous frames, of length 128. On output, points to the ISP coefficients of eight most recent frames (including the current frame).
<i>pSrcDstLogEnergyBuffer</i>	Pointer to the logarithm energy coefficients of eight previous frames, of length 8. On output, points to the logarithm energy coefficients of eight most recent frames (including the current frame).
<i>mode</i>	Bit rate specifier. Values between <code>IPP_SPCHBR_6600</code> and <code>IPP_SPCHBR_23850</code> are valid.

### Description

The function `ippsEncDTXBuffer_AMRWB` is declared in `ippsc.h` file. This function buffers the ISP coefficients and previous logarithm energy coefficients. The buffered ISP coefficients and energy coefficients will be used for SID frame to extract necessary parameters. The memory update index indicates which part of the buffer will be updated and thus saves the cost of some memory copy operation. The logarithm energy is computed as follows:

$$en_{\log}(i) = \frac{1}{2} \log_2 \left( \frac{1}{256} \sum_{n=0}^{255} pSrcSpch^2(n) \right)$$

After computation, the energy is adjusted according to the current encoder mode.

The function `ippsEncDTXBuffer_AMRWB` behaves similarly to the `ippsEncDTXBuffer_GSMAMR` function of the GSM-AMR codec.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>mode</code> is not a valid element of the enumerated type <code>IppSpchBitRate</code> .

## DecDTXBuffer\_AMRWB

*Buffers the ISF coefficients and previous logarithm energy coefficients.*

---

### Syntax

```
IppStatus ippsDecDTXBuffer_AMRWB_16s(const Ipp16s* pSrcSpch, const Ipp16s* pSrcIsf, Ipp16s* pUpdateIndex, Ipp16s* pSrcDstIsfBuffer, Ipp16s* pSrcDstLogEnergyBuffer);
```

### Parameters

<code>pSrcSpch</code>	Pointer to the input speech signal of length 256 .
<code>pSrcIsf</code>	Pointer to the ISP for this frame, of length 16.
<code>pUpdateIndex</code>	Holds the previous memory update index. On output, it contains the current memory update index. Circularly increased between 0 and 7.

*pSrcDstIspBuffer* Pointer to the ISP coefficients of eight previous frames, of length 128. On output, points to the ISP coefficients of eight most recent frames (including the current frame).

*pSrcDstLogEnergyBuffer* Pointer to the logarithm energy coefficients of eight previous frames, of length 8. On output, points to the logarithm energy coefficients of eight most recent frames (including the current frame).

### Description

The function `ippsDecDTXBuffer_AMRWB` is declared in `ippsc.h` file. This function buffers the ISF coefficients and previous logarithm energy coefficients. The buffered ISF coefficients and energy coefficients will be used for SID frame to extract necessary parameters. The memory update index indicates which part of the buffer will be updated and thus saves the cost of some memory copy operation. The logarithm energy is computed as follows:

$$en_{\log}(i) = \frac{1}{2} \log_2 \left( \frac{1}{256} \sum_{n=0}^{255} pSrcSpch^2(n) \right)$$

The function `ippsDecDTXBuffer_AMRWB` behaves similarly to the `ippsDecDTXBuffer_GSMAMR` function of the GSM-AMR codec.

### Return Values

`ippsStsNoErr` Indicates no error.

`ippsStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## AMR Wideband Plus Functions

The Intel IPP functions described in this section if properly combined can be used to construct the Extended Adaptive Multi-Rate Wideband (AMR WB+) Speech Codec compliant to 3rd Generation Partnership Project (3GPP) specification TS 26.273: "ANSI-C code for fixed-point Extended AMR Wideband (AMRWB+) Speech Codec". The description of AMRWB+ codec may be found in 3GPP Technical Specification 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions" [[AMRWB+](#)].

The primitives are primarily designed to implement the well-defined, computationally expensive core operations that comprise the codec portion of the AMRWB+ system. The AMRWB+ codec comprises an extended adaptive multi-rate algorithm, intended for encoding 16-bit uniform PCM at the sampling rate of 16, 24, 32 and 48 KHz. The AMRWB+ codec supports AMRWB compatible modes: 6.60, 8.85, 12.65, 14.25, 15.85, 18.25, 19.85, 23.05 or 23.85 kbps for 16 KHz PCM, four special extension stereo modes: 13.6, 18.0, 24.0 kbps and 24.0 kbps for fixed internal sampling frequency only (25600 Hz) and input PCM 16 or 24 kHz with 20ms frame length. Besides, for thirteen internal sampling frequencies: 12800, 14400, 16000, 17067, 19200, 21333, 24000, 25600, 28800, 32000, 34133, 36000, 38400 and 16, 24, 32 and 48 KHz PCM audio AMRWB+ supports 24 extended bitrates, resulted for fixed internal frequency 25600 Hz in eight mono: 10.4, 12.0, 13.6, 15.2, 16.8, 19.2, 20.8 and 24 kbps, and sixteen stereo bitrates: from 2.0 to 8.0 kbps with step of 0.4 kbps. For each internal sampling frequency ISF supported by AMRWB+ codec the correspondent 24 bitrates can be calculated by multiplication to ISF/25600, that is 1/2, 9/16, 5/8, 2/3, 3/4, 5/6, 15/16, 1, 9/8, 5/4, 4/3, 45/32 and 3/2 respectively. For example, for ISF=12800 with multiplier=1/2, the following bitrates are supported: eight mono 5.2, 6.0, 6.8, 7.6, 9.6, 10.4 and 12 kbps, and sixteen stereo bitrates: from 1.0 to 4.0 with step 0.2 kbps.

The list of Intel IPP AMR WB+ functions is given in the Table 9-7.

**Table 9-7. Intel IPP AMR WB Plus Functions**

Function Base Name	Operation
<a href="#">SNR_AMRWBE</a>	Computes the signal-to-noise ratio
<a href="#">OpenLoopPitchSearch_AMRWBE</a>	Extracts an open-loop pitch lag estimate from the weighted input speech
<a href="#">LPCToISP_AMRWBE</a>	Performs LP to ISP coefficients conversion
<a href="#">SynthesisFilter_AMRWBE</a>	Reconstructs the speech signal from LP coefficients and residuals
<a href="#">Deemphasize_AMRWBE</a>	Performs de-emphasis filtering
<a href="#">FIRGenMidBand_AMRWBE</a>	Computes a shape-constrained FIR filter using the covariance method
<a href="#">PostFilterLowBand_AMRWBE</a>	Post-processes the low-band decoded signal
<a href="#">FFTFwd_RToPerm_AMRWBE</a>	Computes the forward fast Fourier transform (FFT) of a real signal

Function Base Name	Operation
<a href="#">FFTInv_PermToR_AMRWBE</a>	Computes the inverse fast Fourier transform (FFT) of a real signal
<a href="#">AdaptiveCodebookSearch_AMRWBE</a>	Performs the adaptive codebook search
<a href="#">AdaptiveCodebookDecode_AMRWBE</a>	Decodes the adaptive codebook vector
<a href="#">Downsample_AMRWBE</a>	Performs signal downsampling
<a href="#">Upsample_AMRWBE</a>	Performs signal upsampling
<a href="#">BandSplit_AMRWBE</a>	Gets the vector of energy levels from VAD memory
<a href="#">BandJoin_AMRWBE</a>	Joins the low and high frequency signals
<a href="#">BandSplitDownsample_AMRWBE</a>	Decimates input signal and splits it into high and low frequency components
<a href="#">BandJoinUpsample_AMRWBE</a>	Joins high and low frequency signals and upsamples the result
<a href="#">ResamplePolyphase_AMRWBE</a>	Oversamples or downsamples the input signal to or from the upper frequency (44.1/48 khz)
<a href="#">ISFQuantDecode_AMRWBE</a>	Decodes quantized ISF of HF-band signal.
<a href="#">ISFQuantDecodeHighBand_AMRWBE</a>	Performs ISF quantization of HF-band encoded signal
<a href="#">ISFQuantHighBand_AMRWBE</a>	Performs ISF quantization of HF-band encoded signal
<a href="#">GainQuant_AMRWBE</a>	Quantizes the adaptive codebook gains
<a href="#">QuantTCX_AMRWBE</a>	Quantizes the pre-shaped spectrum in TCX mode
<a href="#">GainQuantTCX_AMRWBE</a>	Performs the gain optimization and quantization

Function Base Name	Operation
GainDecodeTCX_AMRWBE	Decodes the global TCX gain
EncodeMux_AMRWBE	Encodes and multiplexes subvectors into several packets
DecodeDemux_AMRWBE	Demultiplexes and decodes subvectors from several packets

## SNR\_AMRWBE

*Computes the signal-to-noise ratio.*

### Syntax

```
IppStatus ippsSNR_AMRWBE_16s(const Ipp16s* pSrcSignal, const Ipp16s* pSrcEstimatedSignal, int lenSrc, int lenSeg, Ipp16s* pDstSNR);
```

### Parameters

<i>pSrcSignal</i>	Pointer to the input signal.
<i>pSrcEstimatedSignal</i>	Pointer to the estimated signal.
<i>lenSrc</i>	Length of the signals.
<i>lenSeg</i>	Length of the segments (subframes).
<i>pDstSNR</i>	Pointer to the signal-to-noise ratio in dB.

### Description

The function `ippsSNR_AMRWBE` is declared in `ippsc.h` file. This function computes in decibels (dB) the average segmental signal-to-noise ratio between the *lenSrc* samples of the signal *pSrcSignal*(*x*) and its estimation *pSrcEstimatedSignal*(*e*). The segmental signal-to-noise ratio (*snr<sub>i</sub>*) in the *i*-th segment is defined as follows:

$$snr_i = \log \left( \frac{\sum_{j=1}^{lenSeg} x_j^2}{\sum_{j=1}^{lenSeg} (x_j - e_j)^2} \right)$$

The average segmental signal-to-noise ratio  $SNR$  is computed as:

$$SNR = \frac{1}{N_{seg}} \cdot \sum_{i=1}^{N_{seg}} snr_i$$

where  $N_{seg}$  is the number of segments.

The segment length is  $lenSeg$  samples. The results are stored in the  $pDstSNR$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when $lenSrc$ or $lenSeg$ is less than or equal to zero.



## OpenLoopPitchSearch\_AMRWBE

*Extracts an open-loop pitch lag estimate from the weighted input speech.*

---

### Syntax

```
IppStatus ippsOpenLoopPitchSearch_AMRWBE_16s (const Ipp16s* pSrcWgtSpch,
const Ipp16s* pSrcFltWgtSpch, Ipp16s* pPrevMedPitchLag, Ipp16s*
pAdaptiveParam, Ipp16s* pDstOpenLoopLag, Ipp16s* pToneFlag, Ipp16s*
pDstOpenLoopGain, Ipp16s* pSrcDstPrevPitchLag, Ipp16s* pSrcDstLagSwitcher,
int len, Ipp16s minPitchLag, Ipp16s maxPitchLag);
```

### Parameters

<i>pSrcWgtSpch</i>	Pointer to a 320-element vector containing perceptually weighted speech.
<i>pSrcFltWgtSpch</i>	Pointer to a 320-element vector containing filtered through high-pass filter perceptually weighted speech.
<i>pPrevMedPitchLag</i>	Pointer to the median filtered pitch lag of the 5 previous voiced speech half-frames.
<i>pAdaptiveParam</i>	Pointer to the adaptive parameter.
<i>pDstOpenLoopLag</i>	Pointer to a two-element vector of open-loop pitch lags.
<i>pToneFlag</i>	Pointer to the tone flag for the VAD module.
<i>pDstOpenLoopGain</i>	Pointer to a 2-element vector containing optimal open-loop pitch gains.
<i>pSrcDstPrevPitchLag</i>	Pointer to the five-element vector that contains the pitch lags associated with the five most recent voiced speech half-frames.
<i>pSrcDstLagSwitcher</i>	Switches lag weighting on and off.
<i>len</i>	Length of the frame.
<i>minPitchLag</i>	Minimum pitch lag.
<i>maxPitchLag</i>	Maximum pitch lag.

## Description

This function is declared in `ippsc.h` file. The functionality is the same as for the function `ippOpenLoopPitchSearch_AMRWB`. The only difference is that the function `ippOpenLoopPitchSearch_AMRWBE` has the variable range ( $\text{minPitchLag} \leq k \leq \text{maxPitchLag}$ ) for computation of a windowed auto correlation. With  $\text{minPitchLag}=17$  and  $\text{maxPitchLag}=115$  these functions are identical.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

## LPCToISP\_AMRWBE

*Performs LP to ISP coefficients conversion.*

---

## Syntax

```
IppStatus ippSLPCToISP_AMRWBE_16s(const Ipp16s* pSrcLpc, const Ipp16s*
pSrcPrevIsp, Ipp16s* pDstIsp, int lpOrder);
```

## Parameters

<i>pSrcLpc</i>	Pointer to the input predictor coefficients.
<i>pDstIsp</i>	Pointer to the output immittance spectral pairs.
<i>pSrcPrevIsp</i>	Pointer to the input previous immittance spectral pairs.
<i>lpOrder</i>	Order of conversion.

## Description

The `ippSLPCToISP_AMRWBE` function is declared in `ippsc.h` file. This function is the extension of the function `ippSLPCToISP_AMRWB`. If value of the parameter *lpOrder* = 16, these functions are identical.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

*ippStsNullPtrErr*

Indicates an error when one of the specified pointers is *NULL*.

## Filtering Functions

This section describes functions that perform filtering operations specific to the AMRWB+ Speech Codec.

### SynthesisFilter\_AMRWBE

*Reconstructs the speech signal from LP coefficients and residuals.*

#### Syntax

```
IppStatus ippsSynthesisFilter_AMRWBE_16s32s_I (Ipp16s* pSrcLpc, int order,
const Ipp16s* pSrcExc, Ipp32s* pSrcDstSignal, int len);
```

#### Parameters

<i>pSrcLpc</i>	Pointer to the LP coefficients vector $a_0, a_1, \dots, a_{order}$
<i>order</i>	Order of LP filter.
<i>pSrcExc</i>	Pointer to the excitation vector.
<i>pSrcDstSignal</i>	Pointer to the synthesized and updated speech.
<i>len</i>	Length of the filter.

#### Description

The function *ippsSynthesisFilter\_AMRWBE* is declared in *ippsc.h* file. This function is identical to *ippsSynthesisFilter\_AMRWB* but performs more accurate calculations and allows wider range of input data that is required for the extended codec.

#### Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when one of the specified pointers is <i>NULL</i> .
<i>ippStsSizeErr</i>	Indicates an error when <i>len</i> or <i>order</i> is less than or equal to zero.

## Deemphasize\_AMRWBE

*Performs de-emphasis filtering.*

---

### Syntax

```
IppStatus ippsDeemphasize_AMRWBE_NR_16s_I(Ipp16s gamma, Ipp32s gammaScale,
Ipp16s *pSrcDstSignal, int len, Ipp16s* pMem);
```

### Parameters

<i>gamma</i>	De-emphasis factor.
<i>gammaScale</i>	Scale factor for the parameter <i>gamma</i> .
<i>pSrcDstSignal</i>	Pointer to the source and destination vector.
<i>len</i>	Length of the source and destination vector.
<i>pMem</i>	Pointer to element of the memory for filtering.

### Description

The `ippsDeemphasize_AMRWBE` function is declared in `ippsc.h` file. This function is the extension of the `ippsDeemphasize_AMRWB` function. If the value of the parameter *gammaScale* = 15, these functions are identical.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> or <i>pMem</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to zero.

## FIRGenMidBand\_AMRWBE

*Computes a shape-constrained FIR filter using the covariance method.*

---

### Syntax

```
IppStatus ippsFIRGenMidBand_AMRWBE_16s(const Ipp16s* pSrcSignal, const Ipp16s*
pSrcSideSignal, Ipp16s* pTaps);
```

## Parameters

<i>pSrcSignal</i>	Pointer to the source signal, [-8...319].
<i>pSrcSideSignal</i>	Pointer to the real side signal [320].
<i>pTaps</i>	Pointer to the FIR coefficients of length [9].

## Description

The function `ippsFIRGenMidBand_AMRWBE` is declared in `ippsc.h` file. This function computes a shape-constrained FIR filter minimizing the expression:

$$\sum_{i=0}^{319} \left( pSrcSideSignal_i - \sum_{j=0}^8 pTaps_j \cdot pSrcSignal_{i-j} \right)^2$$

The filter coefficients are computed with the covariance method using a modified Cholesky algorithm.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clauses 5.5.2.3 [[AMRWB+](#)].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## PostFilterLowBand\_AMRWBE

Post-processes the low-band decoded signal.

### Syntax

```
IpplStatus ippsPostFilterLowBand_AMRWBE_16s(const Ippl6s* pSrcOldPitchLag,
const Ippl6s* pSrcOldPitchGain, Ippl6s* pSrcDstSignal, Ippl6s* pOldSynth,
Ippl6s* pOldNoise, Ippl6s* pFilterScale, Ippl32s pitchAdjust);
```

## Parameters

<i>pSrcOldPitchLag</i>	Pointer to the previous pitch periods for all subframes [16]
<i>pSrcOldPitchGain</i>	Pointer to the previous pitch gains for all subframes [16]
<i>pSrcDstSignal</i>	Pointer to the proceeding signal [1024].
<i>pOldSynth</i>	Pointer to the synthesis memory of post-filter [503].
<i>pOldNoise</i>	Pointer to the noise memory of post-filter [24].
<i>pFilterScale</i>	Pointer to the noise memory of post-filter scale factor.
<i>pitchAdjust</i>	Pitch adjustment flag, if it is <code>NULL</code> , then pitch adjustment is not implemented.

## Description

The function `ippPostFilterLowBand_AMRWBE` is declared in `ippsc.h` file. The decoded signal is first processed by an adaptive pitch enhancer, and then filtered through a long-term prediction filter. Then the obtained long term error signal is low-pass filtered and subtracted from the input signal.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clauses 6.1.3 [[AMRWB+](#)].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## Fast Fourier Transform Functions

This section describes functions that perform fast Fourier transforms specific to the AMRWB+ speech codec.

## FFTFwd\_RToPerm\_AMRWBE

*Computes the forward fast Fourier transform (FFT) of a real signal.*

### Syntax

```
IppStatus ippsFFTFwd_RToPerm_AMRWBE_16s(const Ipp16s* pSrc, Ipp16s* pDst,
int len);
```

### Parameters

<i>pSrc</i>	Pointer to the real-valued sequence.
<i>pDst</i>	Pointer to the transform result.
<i>len</i>	Length of the sequence, possible values 48, 96, 192, 288, 576 or 1152.

### Description

The `ippsFFTFwd_RToPerm_AMRWBE` function is declared in `ippsc.h` file. This function computes  $N_1$  DFTs of size  $N_2$  for real source sequence *pSrc*. Values of  $N_1$  and  $N_2$  are functions of the parameter *len* as follows:

<i>len</i>	48	96	192	288	576	1152
$N_1$	3	3	3	9	9	9
$N_2$	16	32	64	32	64	128

The resulting separate transforms are then combined through radix- $N_1$  transform. The final resulting sequence is stored in the Perm format.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not equal to 48, 96, 192, 288, 576 or 1152.

## FFTInv\_PermToR\_AMRWBE

*Computes the inverse fast Fourier transform (FFT) of a real signal.*

---

### Syntax

```
IppStatus ippsFFTInv_PermToR_AMRWBE_16s(const Ipp16s* pSrc, Ipp16s* pDst,
int len);
```

### Parameters

<i>pSrc</i>	Pointer to the transform coefficients.
<i>pDst</i>	Pointer to the real-valued sequence.
<i>len</i>	Length of the sequence, possible values 48, 96, 192, 288, 576 or 1152.

### Description

The `ippsFFTInv_RToPerm_AMRWBE` function is declared in `ippsc.h` file. This function performs inverse FFT transform. Source data are stored in the Perm format. The results are stored as the real-valued sequence.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not equal to 48, 96, 192, 288, 576 or 1152.

## Codebook Search Functions

This section describes functions that perform codebook searching specific to the AMRWB+ speech codec.



## AdaptiveCodebookSearch\_AMRWBE

*Performs the adaptive codebook search.*

### Syntax

```
IppStatus ippsAdaptiveCodebookSearch_AMRWBE_16s(const Ipp16s* pSrcAdptTarget,
const Ipp16s* pSrcImpulseResponse, const Ipp16s* pSrcOpenLoopLag, Ipp16s*
pPitchLag, Ipp16s* pPitchLagBounds, Ipp16s* pSrcDstExcitation, Ipp16s*
pFracPitchLag, Ipp16s* pAdptIndex, int subFrame, IppSpchBitRate mode, Ipp16s
pitchOffset);
```

### Parameters

<i>pSrcAdptTarget</i>	Pointer to the 64-element adaptive target signal vector.
<i>pSrcImpulseResponse</i>	Pointer to the 64-element impulse response of the weighted synthesis filter.
<i>pSrcOpenLoopLag</i>	Pointer to a two-element vector of the open-loop pitch lags.
<i>pPitchLag</i>	Pointer to the output previous integral pitch lag.
<i>pPitchLagBounds</i>	Pointer to the bounds of the output previous integral pitch lag.
<i>pSrcDstExcitation</i>	Pointer to the input/output 321-element excitation vector.
<i>pFracPitchLag</i>	Pointer to the output fractional pitch lag obtained during the adaptive codebook search.
<i>pAdptIndex</i>	Pointer to the coded closed-loop pitch index.
<i>subFrame</i>	Subframe number.
<i>mode</i>	Coder mode.
<i>pitchOffset</i>	Offset for the pitch adjustment, value range (-17, 17).

### Description

The `ippsAdaptiveCodebookSearch_AMRWBE` function is declared in `ippsc.h` file. This function is the extension of the function `ippsAdaptiveCodebookSearch_AMRWB`. The `pitchOffset` parameter controls the adjustment of the search ranges. If value of `pitchOffset` = 0, both functions are identical.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>pitchOffset</code> is out of the range (-17, 17), or when <code>mode</code> has an illegal value.

## AdaptiveCodebookDecode\_AMRWBE

*Decodes the adaptive codebook vector.*

---

### Syntax

```
IppStatus ippsAdaptiveCodebookDecode_AMRWBE_16s(Ipp32s valAdptIndex, Ipp16s*
pSrcDstExcitation, Ipp16s* pSrcDstPitchLag, Ipp16s* pSrcDstFracPitchLag,
Ipp16s* pSrcDstPitchLagBounds, int subFrame, Ipp32s bfi, ipp32s pitchOffset);
```

### Parameters

<code>valAdptIndex</code>	Adaptive codebook index.
<code>pSrcDstExcitation</code>	Pointer to the 321-element excitation vector.
<code>pSrcDstPitchLag</code>	Pointer to the integral pitch lag.
<code>pSrcDstFracPitchLag</code>	Pointer to the fractional pitch lag obtained during the adaptive codebook search.
<code>pPitchLagBounds</code>	Pointer to the previous integral pitch lag bounds.
<code>subFrame</code>	Subframe number.
<code>bfi</code>	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
<code>pitchOffset</code>	Offset for the pitch adjustment, value range (-17, 17).

### Description

The `ippsAdaptiveCodebookDecode_AMRWBE` function is declared in `ippsc.h` file. This function is similar to the function `ippsAdaptiveCodebookDecode_AMRWB` but has additional parameter `pitchOffset` that controls the adjustment of the search ranges. If value of `pitchOffset` = 0, both functions are identical.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>pitchOffset</i> is out of the range (-17, 17), or when <i>mode</i> has an illegal value.

## Resample Functions

This section describes functions that perform different signal resampling operations specific to the AMRWB+ speech codec.

## Downsample\_AMRWBE

*Performs the signal decimating.*

---

### Syntax

```
ippStatus ippsDownsample_AMRWBE_16s(const Ipp16s* pSrcSignal, int lenSrc,
Ipp16s* pDstSignal, Ipp16s* pMem, Ipp32s bandIdx);
```

### Parameters

<i>pSrcSignal</i>	Pointer to the source signal.
<i>lenSrc</i>	Length of the source vector, [640] for 8 kHz source signal, [1280] for 16 kHz and [1920] for 24 kHz.
<i>pDstSignal</i>	Pointer to destination signal [1024].
<i>pMem</i>	Pointer to the memory for descimating [46].
<i>bandIdx</i>	Index of interpolating band; possible values: 0 for 0..6.4 kHz band, any other for 6.4..10.8 kHz band.

### Description

The function `ippsDownsample_AMRWBE` is declared in `ippsc.h` file. This function performs downsampling of the source signal *pSrcSignal* with the following sample rate: 8, 16, 24 kHz. The function filters input signal to the band 0...6.4 kHz, or 6.4...10.8 kHz in accordance with the parameter *bandIdx*, and downsamples (upsamples for 8 kHz input signal) to 12.6 kHz.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>lenDst</code> has an illegal value.

## Upsample\_AMRWBE

*Performs signal oversampling.*

---

### Syntax

```
IppStatus ippsUpsample_AMRWBE_16s(const Ipp16s* pSrcSignal, Ipp16s*
pSrcDstSignal, int lenDst, Ipp16s* pMem, Ipp32s bandIdx, Ipp32s addFlag);
```

### Parameters

<code>pSrcSignal</code>	Pointer to source signal signal [1024].
<code>pSrcDstSignal</code>	Pointer to the destination oversampled signal.
<code>lenDst</code>	Length of destination vector, [640] for 8kHz output signal, [1280] for 16kHz and [1920] for 24kHz.
<code>pMem</code>	Pointer to the memory for oversampling [24].
<code>bandIdx</code>	Index of interpolating band; possible values: 0 for 0..6.4k band, any other value for 6.4..10.8k band.
<code>addFlag</code>	Flag for adding operation; if it equals to 1 then the result is added to the output vector, if it equals to 0 - the result is not added.

### Description

The function `ippsUpsample_AMRWBE` is declared in `ippsc.h` file. This function performs the signal oversampling of the source vector `pSrcSignal`. This operation is inverse of the operation described in the description of the function [ippsDownsample\\_AMRWBE](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>lenDst</code> has an illegal value.

## BandSplit\_AMRWBE

*Splits the signal into low and high frequency components.*

### Syntax

```
IppStatus ippsBandSplit_AMRWBE_16s(const Ipp16s* pSrcSignal, Ipp16s*
pSrcDstSig2k, Ipp16s* pDstSigHi, int len);
```

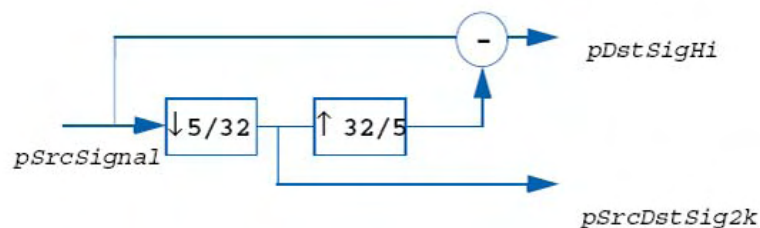
### Parameters

<i>pSrcSignal</i>	Pointer to the source vector [ <i>len</i> + 2*64].
<i>pSrcDstSig2k</i>	Pointer to the source and destination low frequency vector [ <i>len</i> *5/32 + 2*10].
<i>pDstSigHi</i>	Pointer to the destination high frequency vector [ <i>len</i> ].
<i>len</i>	Length of the sample.

### Description

The function `ippsBandSplit_AMRWBE` is declared in `ippsc.h` file. This function splits the input signal ( $f$  kHz) into two bands:  $0 \dots 5/32 * f$  kHz band, and  $5/32 * f \dots f$  kHz band. The low frequency band is critically down-sampled and the side signal is computed according to the diagram on the Figure 9-1 .

**Figure 9-1. Band Split Computing**



See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clauses 5.1.2 [[AMRWB+](#)].

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## BandJoin\_AMRWBE

*Joins the low and high frequency signals.*

---

### Syntax

```
IppStatus ippsBandJoin_AMRWBE_16s(const Ipp16s* pSrcSig2k, const Ipp16s*
pSrcSigHi, Ipp16s* pDstSignal, int len);
```

### Parameters

<code>pSrcSig2k</code>	Pointer to the source low frequency vector [ $len \cdot 5/32 + 2 \cdot 10$ ].
<code>pSrcSigHi</code>	Pointer to the source high frequency vector [ $len$ ].
<code>pDstSignal</code>	Pointer to the destination vector [ $len$ ].
<code>len</code>	Length of the sample.

### Description

The function `ippsBandJoin_AMRWBE` is declared in `ippsc.h` file. This function performs operation that is the inverse of the encoder band-splitting operation described in the description of the function [ippsBandSplit\\_AMRWBE](#), that is the source low frequency signal `pSrcSig2k` is oversampled with factor  $32/5$  and then combined with the source high frequency signal `pSrcSigHi`. The result is store in the vector `pDstSignal`.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clauses 6.3.4 [[AMRWB+](#)].

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when one of the specified pointer is `NULL`.  
`ippStsSizeErr` Indicates an error when `len` is less than or equal to zero.

## BandSplitDownsample\_AMRWBE

*Downsamples input signal and splits it into high and low frequency components.*

### Syntax

```
IppStatus ippsBandSplitDownsample_AMRWBE_16s(const Ipp16s* pSrcSig, int
lenSrc, Ipp16s* pDstSigLF, Ipp16s* pDstSigHF, int lenDst, Ipp16s* pMem,
Ipp16s* pInterFracMem, Ipp32s* pCountSamp, Ipp16s resampleFactor);
```

### Parameters

<code>pSrcSig</code>	Pointer to the source signal.
<code>lenSrc</code>	Length of the source signal, [7680] for 48 kHz signal, or [7056] for 44.1 kHz signal.
<code>pDstSigLF</code>	Pointer to the destination low frequency component.
<code>pDstSigHF</code>	Pointer to the destination high frequency component.
<code>lenDst</code>	Length of destination vectors.
<code>pMem</code>	Pointer to the memory, [1608] elements.
<code>pInterFracMem</code>	Pointer to the memory for the length of interpolating fraction.
	<code>pCountSamp</code> Number of decimated samples.
<code>resampleFactor</code>	Resampling frequency scale factor.

### Description

The function `ippsBandSplitDownsample_AMRWBE` is declared in `ippsc.h` file. This function downsamples input signal to the desired internal sampling frequency  $f$  of the encoder and splits it into high and low frequency components. Firstly the input signal `pSrcSig` is upsampled by the factor `resampleFactor`, filtered by a low pass filter, and then downsampled by the factor 180. The value of the `resampleFactor` factor depends on the desired internal sampling frequency  $f$  (see Table 9-8 below). The low frequency component is obtained by low-pass filtering downsampled signal to  $f/4$  kHz, and critically downsampling result to  $f/2$  kHz. The high frequency component is obtained by band-pass filtering to frequencies above  $f/4$  kHz, and critically downsampling result to  $f/2$  kHz.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clauses 5.1 [AMRWB+].

**Table 9-8. Dependence between Values of the parameter *resampleFactor* and the Internal Sampling Frequency**

Internal sampling frequency $\varepsilon$ , Hz	Factor for the internal frequency	Value of the parameter <i>resampleFactor</i>
12800	1/2	48
14400	9/16	54
16000	5/8	60
17067	2/3	64
19200	3/4	72
21333	5/6	80
24000	15/16	90
25600	1	96
28800	9/8	108
32000	5/4	120
34133	4/3	128
26000	45/32	135
38400	3/2	144

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>lenSrc</i> or <i>lenDst</i> is less than or equal to zero.

## BandJoinUpsample\_AMRWBE

*Joins high and low frequency signals and upsamples the result.*

### Syntax

```
ippStatus ippBandJoinUpsample_AMRWBE_16s(const Ipp16s* pSrcSigLF, const
Ipp16s* pSrcSigHF, int lenSrc, Ipp16s* pDstSig, int lenDst, Ipp16s* pMem,
Ipp16s* pInterFracMem, Ipp32s* pCountSamp, Ipp16s resampleFactor);
```



## Parameters

<i>pSrcSigLF</i>	Pointer to the low frequency signal.
<i>pSrcSigHF</i>	Pointer to the high frequency signal.
<i>lenSrc</i>	Length of low frequency and high frequency vectors.
<i>pDstSig</i>	Pointer to the upsampled signal.
<i>lenDst</i>	Length of the destination vector.
<i>pMem</i>	Pointer to the memory for resampling [72].
<i>pInterFracMem</i>	Pointer to the memory for interpolating fraction.
<i>pCountSamp</i>	Pointer to the number of oversampled samples.
<i>resampleFactor</i>	Downsampling frequency scale factor.

## Description

The function `ippsBandJoinUpsample_AMRWBE` is declared in `ippsc.h` file. This function is used by the decoder: it combines the high and low frequency signals and upsamples the result to produce the full band output signal. This operation is inverse of the encoder band-splitting operation performed by the function `ippsBandSplitDownsample_AMRWBE`.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clauses 6.6 [AMRWB+].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>lenSrc</i> or <i>lenDst</i> is less than or equal to zero.

## ResamplePolyphase\_AMRWBE

*Upsamples or downsamples the input signal to or from the upper frequency (44.1/48 khz).*

### Syntax

```
IppStatus ippsResamplePolyphase_AMRWBE_16s(const Ipp16s* pSrcSignal, int
lenSrc, Ipp16s upFactor, Ipp16s downFactor, Ipp16s* pInterpFracMem, Ipp16s*
pMem, Ipp16s* pDstSignal, int lenDst);
```

## Parameters

<i>pSrcSignal</i>	Pointer to the source signal.
<i>lenSrc</i>	Length of the input vector.
<i>upFactor</i>	Upsampling factor.
<i>downFactor</i>	Downsampling factor.
<i>pInterpFracMem</i>	Pointer to the memory for interpolating fraction.
<i>pMem</i>	Pointer to the memory for resampling, [144] for downsampling, or [44] for upsampling.
<i>pDstSignal</i>	Pointer to the destination resampled signal.
<i>lenDst</i>	Length of the destination vector.

## Description

The function `ippsResamplePolyphase_AMRWB` is declared in `ippsc.h` file. This function performs resampling of the input signal in accordance with the values of the resampling factors *upFactor* (upsampling factor) and *downFactor* (downsampling factor). If *upFactor* < *downFactor*, the function downsamples the input 44.1/48 kHz signal, if *upFactor* > *downFactor*, the function upsamples the input signal to the 44.1/48 kHz signal. The possible combinations of the input/output frequencies and corresponding values of resampling factors are listed in Tables 9-9 and 9-10.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clauses 5.1, 6.6 [AMRWB+].

**Table 9-9. Upsampling Mode**

Input frequency, kHz	Upsampling factor	Downsampling factor	Output frequency, kHz
8	12	2	48
16	12	4	48
24	12	6	48
32	12	8	48
11.025	12	3	44.1
22.05	12	6	44.1

**Table 9-13 Downsampling Mode**

Input frequency, kHz	Upsampling factor	Downsampling factor	Output frequency, kHz
48	2	12	8
48	4	12	16

Input frequency, kHz	Upsampling factor	Downsampling factor	Output frequency, kHz
48	6	12	24
48	8	12	32
44.1	3	12	11.025
44.1	6	12	22.05

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>lenDst</code> has an illegal value.

## Quantization Functions

This section describes functions that perform quantization specific to the AMRWB+ speech codec.

### ISFQuantDecode\_AMRWBE

*Decodes quantized ISFs from the received codebook index.*

#### Syntax

```
IppStatus ippsISFQuantDecode_AMRWBE_16s(const Ipp16s* pSrcIdxs, Ipp16s* pDstQntIsf, Ipp16s* pSrcDstResidual, const Ipp16s* pSrcPrevQntIsf, Ipp16s* pSrcDstIsfMemory, Ipp32s bfi, Ipp32s splitMask);
```

#### Parameters

<code>pSrcIdxs</code>	Pointer to the seven-element vector containing codebook indices of the quantized LSPs.
<code>pDstQntIsf</code>	Pointer to a 16-element destination vector containing quantized ISF in cosine domain.
<code>pSrcDstResidual</code>	Pointer to the 16-element quantized ISF residual from the previous frame.
<code>pSrcPrevQntIsf</code>	Pointer to the 16-element quantized ISF vector from the previous frame.

<i>pSrcDstIsfMemory</i>	Pointer to the 64-element vector containing four subframe ISF sets.
<i>bfi</i>	Bad frame indicator: value "0" indicates a good frame, all other values indicate a bad frame.
<i>splitMask</i>	Binary mask for the second-stage splitting. It contains 5 significant bits, low significant bit corresponds to the low number of splitting.

## Description

The function `ippsISFQuantDecode_AMRWBE` is declared in `ippsc.h` file. The function decodes quantized ISFs from the received codebook index. This function is extension of the function `ippsISFQuantDecode_AMRWE`. If the parameter *splitMask* is set to 0, both functions are identical. Otherwise the function performs the second-level splitting in accordance with the *splitMask*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## ISFQuantDecodeHighBand\_AMRWBE

Decodes quantized ISF of HF-band signal.

## Syntax

```
IppStatus ippsISFQuantDecodeHighBand_AMRWBE_16s(const Ipp16s* pSrcQuantIdxs,
Ipp16s* pSrcDstPastQISF, Ipp16s* pDstQISF, Ipp32s bfi, Ipp32s pitchAdjust);
```

## Parameters

<i>pSrcQuantIdxs</i>	Pointer to the quantization indices [2].
<i>pSrcDstPastQISF</i>	Pointer to the past quantized ISF [8].
<i>pDstQISF</i>	Pointer to the output vector of decoded quantized ISF [8].
<i>bfi</i>	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
<i>pitchAdjust</i>	Controls pitch adjustment: if it equals to 0, then the function does not perform pitch adjustment.

## Description

The function `ippsISFQuantDecodeHighBand_AMRWBE` is declared in `ippsc.h` file. This function decodes quantized ISF of HF-band signal from the received codebook index if the errors are not detected on the received frame. Otherwise, the function recovers the quantized ISFs from previous quantized ISFs using linear interpolation.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## ISFQuantHighBand\_AMRWBE

*Performs ISF quantization of HF-band encoded signal.*

---

## Syntax

```
Ippl16s* ippsISFQuantHighBand_AMRWBE_16s(const Ippl16s* pSrcISF, Ippl16s*
pSrcDstPastQISF, Ippl16s* pDstQISF, Ippl16s* pQuantIdxs, Ippl32s pitchAdjust);
```

## Parameters

<i>pSrcISF</i>	Pointer to the source vector of proceeding ISF [8].
<i>pSrcDstPastQISF</i>	Pointer to the past quantized ISF [8].
<i>pDstQISF</i>	Pointer to the output vector of quantized ISF [8].
<i>pQuantIdxs</i>	Pointer to the output quantization indices [2].
<i>pitchAdjust</i>	Controls pitch adjustment: if it equals to 0, then the function does not perform pitch adjustment.

## Description

The function `ippsISFQuantHighBand_AMRWBE` is declared in `ippsc.h` file. This function performs ISF quantization of HF-band encoded signal using one-stage VQ with 8 elements.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clauses 5.4 [[AMRWB+](#)].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when one of the specified pointer is `NULL`.

## GainQuant\_AMRWBE

*Quantizes the adaptive codebook gains.*

---

### Syntax

```
IppStatus ippGainQuant_AMRWBE_16s(Ipp16s* pSrcAdptTarget, const Ipp16s*
pSrcFltAdptVector, int valFormat, const Ipp16s* pSrcFixedVector, const Ipp16s*
pSrcFltFixedVector, int lenSrc, const Ipp16s* pSrcCorr, Ipp16s meanEnergy,
Ipp16s* pSrcDstPitchGain, Ipp16s* pDstCorrFactor, Ipp32s* pDstCodeGain,
Ipp16s* pDstQGainIndex);
```

### Parameters

<i>pSrcAdptTarget</i>	Pointer to the input target vector $x(n)$ .
<i>pSrcFltAdptVector</i>	Pointer to the input filtered adaptive codebook vector $y(n)$ .
<i>valFormat</i>	Format of the <i>pSrcAdptTarget</i> and <i>pSrcFltAdptVector</i> vectors.
<i>pSrcFixedVector</i>	Pointer to the input pre-filtered codebook contribution $c(n)$ .
<i>pSrcFltFixedVector</i>	Pointer to the input filtered codebook vector $z(n)$ .
<i>lenSrc</i>	Length of the input vectors ( $n$ ).
<i>pSrcCorr</i>	Pointer to the vector of correlations between the <i>pSrcAdptTarget</i> , <i>pSrcFltAdptVector</i> , <i>pSrcFltFixedVector</i> vectors.
<i>meanEnergy</i>	Average energy in the whole frame.
<i>pSrcDstPitchGain</i>	Pointer to the input/output pitch gain.
<i>pDstCorrFactor</i>	Pointer to the correction factor.
<i>pDstCodeGain</i>	Pointer to the output code gain.
<i>pDstQGainIndex</i>	Pointer to the output codebook indexes found.

## Description

The function `ippsGainQuant_AMRWBE` is declared in `ippsc.h` file. This function quantizes the adaptive codebook gain (pitch gain) and the fixed (algebraic) codebook gain using the same 7-bit codebook that is used in the function `ippsGainQuant_AMRWB` for modes from 12.6 to 23.8 kbit/s. However, instead of using "moving average" prediction to obtain the predicted gain  $g_c$ , it is found directly by quantizing the average energy *meanEnergy* in the whole frame.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>lenSrc</i> is less than or equal to zero.

## QuantTCX\_AMRWBE

Quantizes the pre-shaped spectrum in TCX mode.

## Syntax

```
IpplStatus ippsQuantTCX_AMRWBE_16s(const Ippl6s* pSrc, Ippl6s* pDst, int
nSubvectors, int nBits, Ippl6s* pDstNoiseFactor);
```

## Parameters

<i>pSrc</i>	Pointer to the source signal [ <i>nSubvectors</i> *8].
<i>pDst</i>	Pointer to the quantized normalized vector[ <i>nSubvectors</i> *8 + <i>nSubvectors</i> ].
<i>nSubvectors</i>	Number of subvectors.
<i>nBits</i>	Number of bits to use.
<i>pDstNoiseFactor</i>	Pointer to the comfort noise gain factor.

## Description

The function `ippsQuantTCX_AMRWBE` is declared in `ippsc.h` file. This function quantizes the pre-shaped spectrum in TCX mode. This function quantizes the pre-shaped spectrum in TCX mode using lattice quantizers method. The spectrum is quantized in 8-dimensional blocks using vector codebooks composed of subsets of the Gosset lattice. In lattice quantization, the procedure of finding the nearest neighbour of an input vector among all codebook points is reduced to a

several simple operations - rounding the components of the vector and verifying a few constraints with no exhaustive search. The binary indexes are computed only if a given TCX mode is retained as the best mode for a frame. It allows to reduce computation complexity.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clause 5.3.5.7 [AMRWB+].

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## GainQuantTCX\_AMRWBE

*Performs the gain optimization and quantization.*

---

### Syntax

```
IppStatus ippGainQuantTCX_AMRWBE_16s(const Ipp16s* pSrcSignal, Ipp16s
srcScale, const Ipp16s* pSrcQuantSignal, int len, Ipp32s quantFlag, Ipp32s*
pGain, Ipp16s* pQuantIdx);
```

### Parameters

<code>pSrcSignal</code>	Pointer to the original weighted signal.
<code>srcScale</code>	Format of input signal.
<code>pSrcQuantSignal</code>	Pointer to the quantized weighted signal.
<code>len</code>	Length of the source vector.
<code>quantFlag</code>	If equals to 0, then the function computes and returns the optimal gain only, else computes and returns the quantized gain and quantization index.
<code>pGain</code>	Pointer to the output quantized or optimal gain.
<code>pQuantIdx</code>	Pointer to the output index of quantization.

### Description

The function `ippGainQuantTCX_AMRWBE` is declared in `ippsc.h` file. This function performs the gain optimization and quantization to maximize the correlation between the original weighted signal  $pSrcSignal(x)$  and the quantized weighted signal  $pSrcQuantSignal(x')$ . The optimal gain  $g^*$  between  $x$  and  $x'$  is computed in accordance with the following formula:



$$g^* = \frac{\sum_{i=0}^{len-1} x_i \cdot \hat{x}_i}{\sum_{i=0}^{len-1} \hat{x}_i \cdot \hat{x}_i}$$

The gain  $g^*$  is quantized on the logarithmic scale to a 7-bit index in the following way:

- compute the energy of the quantized weighted signal

$$E = \sum_{i=0}^{len-1} \hat{x}_i^2;$$

- compute the RMS value  $rms = 4(E/len)^{1/2}$ ;
- compute the index  $index = 28\log(rms \cdot g^*) + 0.5$ ;
- if  $index < 0$ , then set  $index = 0$ ; if  $index > 127$ , then set  $index = 127$ ;
- and the quantized gain  $g^* = 10^{index/28rms}$

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clause 5.3.5.10 [AMRWB+].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## GainDecodeTCX\_AMRWBE

*Decodes the global TCX gain.*

---

### Syntax

```
IppStatus ippsGainDecodeTCX_AMRWBE_16s(const Ipp16s* pSrcQuantSignal, int
len, Ipp16s quantIdx, Ipp32s bfi, Ipp16s* pSrcDstRMSval, Ipp32s* pGain);
```

### Parameters

<i>pSrcQuantSignal</i>	Pointer to the quantized signal.
<i>len</i>	Length of the source signal.
<i>quantIdx</i>	Index of quantization.
<i>bfi</i>	Bad frame indicator. Value "0" signifies a good frame; any other value signifies a bad frame.
<i>pSrcDstRMSval</i>	Pointer to the root mean square value.
<i>pGain</i>	Pointer to the output global TCX gain.

### Description

The function `ippsGainDecodeTCX_AMRWBE` is declared in `ipps.h` file. This function decodes the global TCX gain  $g_{TCX}$  by inverting the 7-bit logarithmic quantization calculated in the TCX encoder as in the function [ippsGainQuantTCX\\_AMRWBE](#).

First the RMS value of the TCX target signal  $pSrcQuantSignal(x')$  is computed as

$$rms = \sqrt{\frac{\sum_{i=0}^{len-1} \hat{x}_i^2}{len}}$$

From the received 7-bit index  $0 \leq quantIdx \leq 127$ , the TCX gain  $pGain$  can be computed

$$pGain* = 10^{quantIdx/28*4_{rms}}$$

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clause 6.1.2 [[AMRWB+](#)].

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to zero.

## EncodeMux\_AMRWBE

*Encodes and multiplexes subvectors into several packets.*

---

### Syntax

```
IpStatus ippsEncodeMux_AMRWBE_16s(const Ippl6s* pSrc, int nSubvectors, const
int* pPacketSizes, Ippl6s* pDstParams, int nPackets);
```

### Parameters

<code>pSrc</code>	Pointer to the rounded subvectors [ $nSubvectors*8 + nSubvectors$ ].
<code>nSubvectors</code>	Number of subvectors.
<code>pPacketSizes</code>	Pointer to the vector of size of each packet [ $nPackets$ ].
<code>pDstParams</code>	Multiplexed parameters [ $(pPacketSizes[0]+3)/4 + \dots + (pPacketSizes[nPackets]+3)/4$ ].
<code>nPackets</code>	Number of packets.

### Description

The function `ippsEncodeMux_AMRWBE` is declared in `ippsc.h` file. This function encodes the TCX parameters and puts them in one or several binary packets for transmission.

See also 3GPP TS 26.290: "Extended AMR Wideband Speech Codec; Transcoding functions", clause 5.6 [AMRWB+].

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## DecodeDemux\_AMRWBE

*Demultiplexes and decodes subvectors from several packets.*

---

### Syntax

```
IppStatus ippsDecodeDemux_AMRWBE_16s(const Ipp16s* pSrcParams, const Ipp32s*
pPacketSizes, const Ipp32s* pBFI, int nPackets, Ipp16s* pDst, int
nSubvectors);
```

### Parameters

<i>pSrcParams</i>	Demultiplexed parameters $[(pPacketSizes[0]+3)/4 + \dots + (pPacketSizes[nPackets]+3)/4]$ .
<i>pPacketSizes</i>	Pointer to the vector of size of each packet $[nPackets]$ .
<i>pBFI</i>	Pointer to the vector of bad frame indicator for each packet $[nPackets]$ .
<i>nPackets</i>	Number of packets.
<i>pDst</i>	Pointer to the rounded subvectors $[nSubvectors*8]$ .
<i>nSubvectors</i>	Number of subvectors.

### Description

The function `ippsDecodeDemux_AMRWBE` is declared in `ippsc.h` file. This function demultiplexes and decodes subvectors from several packets, that is performs inverse operation to the encoding procedure performed by the function [ippsEncodeMux\\_AMRWBE](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## GSM Full Rate Functions

This section describes the Intel IPP functions that can be used in implementing speech codecs following the GSM 06.10, 06.11, 06.12, 06.31, and 06.32 recommendations. The list of these functions is given in Table 9-11.

Table 9-11. Intel IPP GSM Full Rate Functions

Function Base Name	Operation
RPEQuantDecode_GSMFR	Performs APCM inverse quantization.
Deemphasize_GSMFR	Performs de-emphasis filtering.
ShortTermAnalysisFilter_GSMFR	Performs short-term analysis filtering.
ShortTermSynthesisFilter_GSMFR	Performs short-term synthesis filtering.
HighPassFilter_GSMFR	Filters input speech signal through a high-pass filter.
Schur_GSMFR	Estimates the reflection coefficients by Schur recursion.
WeightingFilter_GSMFR	Calculates the weighting filter.
Preemphasize_GSMFR	Computes pre-emphasis of a speech signal.

RPEQuantDecode\_GSMFR

Performs APCM inverse quantization.

Syntax

```
IppStatus ippsRPEQuantDecode_GSMFR_16s (const Ipp16s* pSrc, Ipp16s ampl, Ipp16s amplSfs, Ipp16s* pDst);
```

Parameters

<i>pSrc</i>	Pointer to the input vector [13] of the RPE samples.
<i>ampl</i>	The block amplitude.
<i>amplSfs</i>	Scale factor of the block amplitude.
<i>pDst</i>	Pointer to the output reconstructed long-term residual vector [13].

Description

The function ippsRPEQuantDecode\_GSMFR is declared in ippsc.h file. This function performs APCM inverse quantization of the input RPE samples. The output reconstructed long-term residual vector is formed as

$$pDst[i] = (2 \cdot pSrc[i] - 7) \cdot \frac{amp}{2^{amplSfs}}$$

## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointer is `NULL`.

`ippStsScaleRangeErr` Indicates an error when `amp1Sfs` is less than 0.

## Deemphasize\_GSMFR

*Performs de-emphasis filtering.*

---

### Syntax

```
IppStatus ippDeemphasize_GSMFR_16s_I (Ipp16s* pSrcDst, int len, Ipp16s* pMem);
```

### Parameters

`pSrcDst` Pointer to the input short-term synthesized signal and output post-processed speech vector [`len`].

`len` Length of the input residual and output speech vectors.

`pMem` Pointer to the filter memory element.

### Description

The function `ippDeemphasize_GSMFR` is declared in `ippsc.h` file. This function performs de-emphasis of the input synthesized signal by filtering it through the filter with the following transfer function:

$$H(z) = 1 / (1 - \alpha * z^{-1})$$

where  $\psi = 0.86$  (28180 in Q15).

The initial memory of the filter will be set to zero. The filtered speech signal is scaled up by multiple of 2 and then stored in `pSrcDst` with truncation of the three least significant bits.

### Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointer is `NULL`.

`ippStsRangeErr` Indicates an error when `len` is less than or equal to 0.

## ShortTermAnalysisFilter\_GSMFR

*Performs short-term analysis filtering.*

---

### Syntax

```
IppStatus ippsShortTermAnalysisFilter_GSMFR_16s_I (const Ipp16s* pRC, Ipp16s*
pSrcDstSpch, int len, Ipp16s* pMem);
```

### Parameters

<i>pRC</i>	Pointer to the input reflection coefficients vector [8]: $a_1, a_2, \dots, a_8$ .
<i>pSrcDstSpch</i>	Pointer to the input pre-processed speech and output short term residual vector [ <i>len</i> ].
<i>len</i>	Length of the input speech and output residual vectors.
<i>pMem</i>	Pointer to the filter memory vector [8]: $m_0, m_1, \dots, m_7$ .

### Description

The function `ippsShortTermAnalysisFilter_GSMFR` is declared in `ippsc.h` file. This function performs filtering of the input pre-processed speech vector  $s(n)$  and stores the result in the output short-term residual vector  $r(n)$  as given below:  $r_0 = s(n)$

$$r_i = r_{i-1} + a_i * m_{i-1}, i = 1, \dots, 8$$

$$m_0 = s(n)$$

$$m_i = m_{i-1} + a_i * r_{i-1}, i = 1, \dots, 7$$

$$r(n) = r_8$$

where  $m_i, i = 0, \dots, 7$  is the filter memory and  $r_i, i = 0, \dots, 8$  is the reusable local memory.

The initial filter memory vector will be zeroed.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## ShortTermSynthesisFilter\_GSMFR

*Performs short-term synthesis filtering.*

---

### Syntax

```
IppStatus ippsShortTermSynthesisFilter_GSMFR_16s (const Ipp16s* pRC, const Ipp16s* pSrcResidual, Ipp16s* pDstSpch, int len, Ipp16s* pMem);
```

### Parameters

<i>pRC</i>	Pointer to the input reflection coefficients vector [8]: Pointer to the input reflection coefficients vector [8]: $a_1, a_2, \dots, a_8$ .
<i>pSrcResidual</i>	Pointer to the input reconstructed short term residual vector [ <i>len</i> ].
<i>pDstSpch</i>	Pointer to the output speech vector [ <i>len</i> ].
<i>len</i>	Length of the input residual and output speech vectors.
<i>pMem</i>	Pointer to the filter memory vector [8]: $m_0, m_1, \dots, m_7$ .

### Description

The function `ippsShortTermSynthesisFilter_GSMFR` is declared in `ippsc.h` file. This function performs filtering of the input reconstructed short-term residual  $r(n)$  and stores the result in the output speech vector  $s(n)$  as given below:

$$s_0 = r(n)$$

$$s_i = s_{i-1} - a_{9-i} * m_{8-i}, i = 1, \dots, 8$$

$$m_{8-i} = m_{7-i} + -a_{9-i} * s_i, i = 1, \dots, 7$$

$$s(n) = s_8$$

$$m_0 = s(n)$$

where  $m_i, i = 0, \dots, 7$  is the filter memory and  $s_i, i = 0, \dots, 8$  is the reusable local memory.

The initial filter memory vector will be zeroed.

### Return Values

`ippStsNoErr` Indicates no error.



<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## HighPassFilter\_GSMFR

*Performs high-pass filtering of the input speech signal.*

---

### Syntax

```
IppStatus ippsHighPassFilter_GSMFR_16s (const Ipp16s* pSrc, Ipp16s* pDst,
int len, int* pMem);
```

### Parameters

<code>pSrc</code>	Pointer to the source speech vector [ <code>len</code> ].
<code>pDst</code>	Pointer to the destination filtered vector [ <code>len</code> ].
<code>len</code>	Length of the source and destination vectors.
<code>pMem</code>	Pointer to the filter memory vector [2].

### Description

The function `ippsHighPassFilter_GSMFR` is declared in `ippsc.h` file. This function filters the input speech signal according to the transfer function:

$$H(z) = 0.5(1 - z^{-1}) / (1 - \alpha * z^{-1})$$

where  $\psi = 0.99899$ .

The initial filter memory will be set to zero.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Schur\_GSMFR

*Estimates the reflection coefficients by Schur recursion.*

---

### Syntax

```
IppStatus ippsSchur_GSMFR_32s16s (const Ipp32s* pSrc, Ipp16s* pDst, int dstLen);
```

### Parameters

<i>pSrc</i>	Pointer to the input autocorrelation vector [ <i>dstLen</i> +1].
<i>pDst</i>	Pointer to the output reflection coefficients vector [ <i>dstLen</i> ].
<i>dstLen</i>	The number of reflection coefficients to estimate.

### Description

The function `ippsSchur_GSMFR` is declared in `ippsc.h` file. This function implements the Schur algorithm according to GSM 06.10 clause 4.2.5. See also the function [ippsSchur](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>dstLen</i> is less or equal to 0, or <i>dstLen</i> is less than 9.

## WeightingFilter\_GSMFR

*Calculates the weighting filter.*

---

### Syntax

```
IppStatus ippsWeightingFilter_GSMFR_16s (const Ipp16s* pSrc, Ipp16s* pDst, int dstLen);
```

## Parameters

<i>pSrc</i>	Pointer to the input long-term residual vector [-5,..., <i>dstLen</i> +4].
<i>pDst</i>	Pointer to the filtered output vector [ <i>dstLen</i> ].
<i>dstLen</i>	The number of filtered elements to calculate.

## Description

The function `ippWeightingFilter_GSMFR` is declared in `ippsc.h` file. This function performs filtering of the input signal by symmetric FIR filter with predefined taps given below:

*taps*[i]=[-134, -374, 0, 2054, 5741, 8192, 5741, 2054, 0, -374, -134], i=0,..,10

The filtering is performing according to formula:

$$dst[n] = \sum_{i=0}^{10} taps[i] \cdot src[n+i-5], \quad n = 0, \dots, dstLen-1$$

The result of filtering is stored in *pDst*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Preemphasize\_GSMFR

*Computes pre-emphasis of a speech signal.*

---

### Syntax

```
IppStatus ippPreemphasize_GSMFR_16s(const Ipp16s* pSrc, Ipp16s* pDst, int*
pMem, int len);
```

## Parameters

<i>pSrc</i>	Pointer to the offset-free input speech signal.
<i>pDst</i>	Pointer to the pre-emphasized output signal.
<i>pMem</i>	Pointer to the filter memory value.
<i>len</i>	Length of the input and output signals.

## Description

The function `ippsPreemphasize_GSMFR` is declared in `ippsc.h` file. This function computes pre-emphasis of the input speech according to the difference signal pre-emphasis equation:

$$H(z) = 1 - \gamma z^{-1}$$

for  $\phi = -0.86$  and  $pSrc[-1] = pMem[0]$ .

The result of filtering is stored in *pDst*. The memory value *pMem*[0] is updated by *pSrc*[n-1]. For proper use of this function in GSM Full Rate codec, the memory value will be initialized to zero.

The function `ippsPreemphasize_GSMFR` performs NR rounding (see Rounding mode ).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

# G.722.1 Functions

This chapter describes the Intel IPP functions that can be used in implementing speech codecs following the ITU-T recommendations G.722.1. The list of these functions is given in Table 9-12.

**Table 9-12. Intel IPP G.722.1 Related Functions**

Function Base Name	Operation
<code>DCTFwd_G722</code> , <code>DCTFwd_G7221</code>	Compute the forward discrete cosine transform (DCT) of a signal.
<code>DCTInv_G722</code> , <code>DCTInv_G7221</code>	Compute the inverse discrete cosine transform (DCT) of a signal

Function Base Name	Operation
<a href="#">DecomposeMLTToDCT_G722</a> , <a href="#">DecomposeMLTToDCT_G7221</a>	Decomposes the MLT input signal to the form of the DCT input signal.
<a href="#">DecomposeDCTToMLT_G722</a> , <a href="#">DecomposeDCTToMLT_G7221</a>	Decomposes inverse DCT output signal to the form of the MLT transform output signal.
<a href="#">HuffmanEncode_G722</a>	Performs Huffman encoding of the quantized amplitude envelope indexes.

These functions are used in the Intel IPP *G.722.1 Speech Encoder-Decoders* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## DCTFwd\_G722, DCTFwd\_G7221

*Computes the forward discrete cosine transform (DCT) of a signal.*

### Syntax

```
IppStatus ippsDCTFwd_G722_16s(const Ipp16s* pSrc, Ipp16s* pDst);  
IppStatus ippsDCTFwd_G7221_16s (const Ipp16s* pSrc, Ipp16s* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements on the source and destination vectors, possible values: 320, 640.

### Description

The functions `ippsDCTFwd` are declared in `ippsc.h` file. The function `ippsDCTFWD_G722` computes the forward discrete cosine transform (DCT) of the fixed length [320]. The function `ippsDCT-FWD_G7221` computes the forward DCT of the length *len*. The calculation is performed according to the following formula:

$$y^{(m)} = \sum_{n=0}^{len-1} \sqrt{\frac{2}{len}} \cos\left(\frac{\pi}{len} \cdot (n+0.5) \cdot (m+0.5)\right) \cdot x(n), \quad 0 \leq m \leq len$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when the <code>len</code> has an illegal value.

## DCTInv\_G722, DCTInvG7221

*Computes the inverse discrete cosine transform (DCT) of a signal.*

---

### Syntax

```
ippstatus ippsDCTInv_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst);
ippstatus ippsDCTInv_G7221_16s (const Ipp16s* pSrc, Ipp16s* pDst, int len);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>len</code>	Number of elements on the source and destination vectors, possible values: 320, 640.

### Description

The functions `ippsDCTInv` are declared in `ippsc.h` file. The function `ippsDCTInv_G722` computes the inverse discrete cosine transform (DCT) of the fixed length [320]. The function `ippsDCTInv_G7221` computes the inverse DCT of the length `len`. The calculation is performed according to the following formula:

$$x(n) = \sum_{m=0}^{len-1} \sqrt{\frac{2}{len}} \cos\left(\frac{\pi}{len} \cdot (m+0.5) \cdot (n+0.5)\right) \cdot y(m), \quad 0 \leq n \leq len$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when one of the specified pointer is `NULL`.  
`ippStsSizeErr` Indicates an error when the `len` has an illegal value.

## DecomposeMLTToDCT\_G722,DecomposeMLTToDCT\_G7221

*Decomposes the MLT input signal to the form of the DCT input signal.*

### Syntax

```
IppStatus ippDecomposeMLTToDCT_G722_16s(const Ipp16s* pSrcSpch, Ipp16s*
pSrcDstSpchOld, Ipp16s* pDstSpchDecomposed);
```

```
IppStatus ippDecomposeMLTToDCT_G7221_16s(const Ipp16s* pSrcSpch, Ipp16s*
pSrcDstSpchOld, Ipp16s* pDstSpchDecomposed, int len);
```

### Parameters

*pSrcSpch* Pointer to the source vector.  
*pSrcSpchOld* Pointer to the source/destination vector of of the previous speech frame.  
*pDstSpchDecomposed* Pointer to the destination decomposed speech vector.  
*len* Number of elements on the source and destination vectors, possible values: 320, 640.

### Description

The functions `ippDecomposeMLTToDCT` are declared in the `ippsc.h` file. These functions decompose the input signal by windowing, overlapping and summation to the form suitable for the DCT operation required for modulated lapped transform (MLT) coefficients calculation. The function `ippDecomposeMLTToDCT_G722` operates with the vector of the fixed length [320]. The function `ippDecomposeMLTToDCT_G7221` operates with the vector of the `len` length.

If  $l_1 = len$ ,  $l_2 = len / 2$ , and  $v = pDstSpchDecomposed$ , then the calculation is performed as follows:

$$\begin{aligned} v(n) &= w(l_2 - 1 - n) \cdot x(l_2 - 1 - n) + w(l_2 + n) \cdot x(l_2 + n), & 0 \leq n < l_2 \\ v(n + l_2) &= w(1 - n) \cdot x(1 + n) + w(n) \cdot x(2l - n), & 0 \leq n < l_2 \end{aligned}$$

where

$$w(n) = \sin\left(\frac{\pi}{21} \cdot (n + 0.5)\right), \quad 0 \leq n < 1$$

and

$$x(n) = \begin{cases} pSrcDstSpchOld[n], & 0 \leq n < 1 \\ pSrcSpch[n - 1], & 1 \leq n < 21 \end{cases}$$

The input signal *pSrcSpch* is stored in *pSrcDstSpchOld* for use in the next frame.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when the <i>len</i> has an illegal value.

## DecomposeDCTToMLT\_G722, DecomposeDCTToMLT\_G7221

*Decomposes inverse DCT output signal to the form of the MLT output signal.*

---

### Syntax

```

IppStatus ippDecomposeDCTToMLT_G722_16s(const Ipp16s* pSrcSpchDecomposed,
Ipp16s* pSrcDstSpchDecomposedOld, Ipp16s* pDstSpch);

IppStatus ippDecomposeDCTToMLT_G7221_16s(const Ipp16s* pSrcSpchDecomposed,
Ipp16s* pSrcDstSpchDecomposedOld, Ipp16s* pDstSpch, int len);

```

### Parameters

<i>pSrcSpchDecomposed</i>	Pointer to the source vector Pointer to the source windowed speech vector [320 or <i>len</i> ].
<i>pSrcDstSpchDecomposedOld</i>	Pointer to the source and destination windowed speech vector [160 or <i>len</i> /2] of the previous frame.



*pDstSpch* Pointer to the destination speech vector [320 or *len*].  
*len* Number of elements on the source and destination vectors, possible values: 320, 640.

## Description

The functions `ippsDecomposeDCTToMLT` are declared in the `ippsc.h` file. These functions perform inverse decomposition of the inverse DCT output by windowing, overlapping and summation, and restore speech signal. The function `ippsDecomposeDCTToMLT_G722` operates with the source vector of the fixed length [320]. The function `ippsDecomposeDCTToMLT_G7221` operates with the source vector of the *len* length.

If  $l = len$ ,  $l_2 = len / 2$ , and  $u = pSrcSpchDecomposed$ ,  $u_{old} = pSrcDstSpchDecomposedOld$ ,  $y = pDstSpch$ , then the calculation is performed as follows:

$$\begin{aligned} y(n) &= w(n) \cdot u(l_2 - 1 - n) + w(l - n) \cdot u_{old}(n), & 0 \leq n < l_2 \\ y(n + l_2) &= w(l_2 + n) \cdot u(n) - w(l_2 - 1 - n) \cdot u_{old}(l_2 - 1 - n), & 0 \leq n < l_2 \end{aligned}$$

where

$$w(n) = \sin\left(\frac{\pi}{2l} \cdot (n + 0.5)\right), \quad 0 \leq n < l$$

The unused second half of the input signal is stored for use with the next frame:

$$u_{old}(n) = u(l_2 + n), \quad 0 \leq n < l.$$

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointer is `NULL`.  
`ippStsSizeErr` Indicates an error when the *len* has an illegal value.

## HuffmanEncode\_G722

*Performs Huffman encoding of the quantized amplitude envelope indexes.*

---

### Syntax

```
IppStatus ippHuffmanEncode_G722_16s32u(int category, int qntAmpEnvIndex,
const Ipp16s* pSrcMLTCoeffs, Ipp32u* pDstCode, int* pCodeLength);
```

### Parameters

<i>category</i>	The category of the Modulated Lapped Transform (MLT) region in the range of [0-6].
<i>qntAmpEnvIndex</i>	The quantized amplitude envelope index in the range of [0-63].
<i>pSrcMLTCoeffs</i>	Pointer to the source vector [20] of raw MLT coefficients.
<i>pDstCode</i>	Pointer to the output Huffman code.
<i>pCodeLength</i>	Pointer to the output Huffman code length in bits.

### Description

The function `ippHuffmanEncode_G722` is declared in `ippsc.h` file. This function performs Huffman encoding of the quantized index of the amplitude envelope of one of the twenty regions of the MLT coefficients.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>IppStsScaleRangeErr</code>	Indicates an error when <i>category</i> or <i>qntAmpEnvIndex</i> is out of proper range.

## G.726 Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following the ITU-T recommendations G.726 with Annex A.

These functions are used in the Intel IPP G.726 *Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

The list of these functions is given in Table 9-13.

**Table 9-13. Intel IPP G.726.1 Related Functions**

Function Base Name	Operation
<a href="#">EncodeGetStateSize_G726</a>	Informative function, returns the number of bytes needed for encoder memory.
<a href="#">EncodeInit_G726</a>	Initializes the memory for the ADPCM encoder.
<a href="#">Encode_G726</a>	Performs ADPCM compression of the uniform PCM input.
<a href="#">DecodeGetStateSize_G726</a>	Informative function, returns the number of bytes needed for decoder memory.
<a href="#">DecodeInit_G726</a>	Initializes the memory for the G726 decoder.
<a href="#">Decode_G726</a>	Performs decompression of the ADPCM bit-stream.

## EncodeGetStateSize\_G726

*Informative function, returns the number of bytes needed for encoder memory.*

### Syntax

```
IppStatus ippsEncodeGetStateSize_G726_16s8u (unsigned int* pEncSize);
```

### Parameters

*pEncSize*                      Pointer to the output memory size in bytes.

### Description

The function `ippsEncodeGetStateSize_G726` is declared in `ippsc.h` file. This function gets information about the amount of memory needed to process the G.726 ADPCM compression.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`IppStsNullPtrErr`                Indicates an error when the *pEncSize* pointer is NULL.

## EncodeInit\_G726

*Initializes the memory for the ADPCM encoder.*

---

### Syntax

```
IppStatus ippsEncodeInit_G726_16s8u (IppsEncoderState_G726_16s* pEncMem,
IppSpchBitRate rate);
```

### Parameters

<i>pEncMem</i>	Pointer to the input memory buffer of size needed to properly initialize the G.726 encoder.
<i>rate</i>	Encode bit rate of the G.726 encoder, must be one of IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000.

### Description

The function `ippsEncodeInit_G726` is declared in `ippsc.h` file. This function initializes the memory given by the pointer *pEncMem* to enable G.726 ADPCM compression starting from the reset state.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pEncMem</i> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>rate</i> is not equal to one of the admissible encoding bit rates: IPP_SPCHBR_16000, IPP_SPCHBR_24000, IPP_SPCHBR_32000, or IPP_SPCHBR_40000.

## Encode\_G726

*Performs ADPCM compression of the uniform PCM input.*

---

### Syntax

```
IppStatus ippsEncode_G726_16s8u (IppsEncoderState_G726_16s* pEncMem , const
Ipp16s* pSrc, Ipp8u* pDst, unsigned int len);
```

## Parameters

<i>pEncMem</i>	Pointer to the memory buffer that has been initialized for ADPCM encode.
<i>pSrc</i>	Pointer to the uniform PCM input speech vector.
<i>pDst</i>	Pointer to the ADPCM bit-stream output vector.
<i>len</i>	The length of input/output vectors.

## Description

The function `ippsEncode_G726` is declared in `ipps.h` file. This function performs ADPCM compression of the 14-bit uniform PCM speech input (Recommendation G.726, Annex A) with the bit rate on which the G.726 encoder (with memory pointed to by *pEncMem*) was initialized to operate. Each byte of the output vector contains ADPCM compressed value of two, three, four, or five binary digits for 16, 24, 32 or 40 Kbit/s bit-rate ADPCM compression, respectively.

The Mu-Law or A-Law PCM input should be expanded to 14-bit uniform PCM prior to ADPCM compression (see Recommendation G.726). This expansion may be done, for example, by first applying the functions `ippsMuLawToLin_8u16s` or `ippsALawToMuLaw_8u16s`, which expand 8-bit Mu-Law or A-Law PCM, respectively, into linear 16-bit PCM.

The linear 16-bit PCM input must be shifted two bits to the right (divided by four) to achieve the 14-bit uniform PCM input appropriate for the function `ippsEncode_G726`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when the <i>len</i> is less than or equal to 0.

## DecodeGetStateSize\_G726

*Informative function, returns the number of bytes needed for decoder memory.*

---

### Syntax

```
IppStatus ippsDecodeGetStateSize_G726_8u16s (unsigned int* pDecSize);
```

### Parameters

<i>pDecSize</i>	Pointer to the output memory size in bytes.
-----------------	---

## Description

The function `ippSdsDecodeGetStateSize_G726` is declared in `ippsc.h` file. This function reports the amount of memory needed to process the G.726 ADPCM decompression.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <code>pDecSize</code> pointer is <code>NULL</code> .

## DecodeInit\_G726

*Initializes the memory for the G726 decoder.*

---

### Syntax

```
IppStatus ippSdsDecodeInit_G726_8u16s (IppsDecoderState_G726_16s* pDecMem,
IppSpchBitRate rate, IppPCMLaw law);
```

### Parameters

<code>pDecMem</code>	Pointer to the input memory buffer of size needed to properly initialize the G.726 decoder.
<code>rate</code>	Decode bit rate of the G.726 decoder, must be one of <code>IPP_SPCHBR_16000</code> , <code>IPP_SPCHBR_24000</code> , <code>IPP_SPCHBR_32000</code> , or <code>IPP_SPCHBR_40000</code> .
<code>law</code>	Output speech PCM law: must be one of <code>IPP_PCM_MULAW</code> , <code>IPP_PCM_ALAW</code> or <code>IPP_PCM_LINEAR</code> .

## Description

The function `ippSdsDecodeInit_G726` is declared in `ippsc.h` file. This function initializes the memory given by the pointer `pDecMem` to enable ADPCM decompression starting from the reset state.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <code>pDecMem</code> pointer is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <code>rate</code> or <code>law</code> has an illegal value.

## Decode\_G726

*Performs decompression of the ADPCM bit-stream.*

### Syntax

```
IppStatus ippsDecode_G726_8u16s (IppsDecoderState_G726_16s* pDecMem, const
Ipp8u* pSrc, Ipp16s* pDst, unsigned int len);
```

### Parameters

<i>pDecMem</i>	Pointer to the memory buffer that has been initialized for G.726 ADPCM decoder.
<i>pSrc</i>	Pointer to the input vector that contains two, three, four, or five binary digits per byte for 16, 24, 32, or 40 Kbit/s ADPCM bit-stream, respectively.
<i>pDst</i>	Pointer to the 16-bit linear PCM speech output vector.
<i>len</i>	The length of input/output vectors.

### Description

The function `ippsDecode_G726` is declared in `ippsc.h` file. This function performs decompression of the ADPCM bit-stream input (Recommendation G.726) into 16-bit linear PCM. The input bit-stream must be ADPCM compressed on the bit rate for which the G.726 decoder was initialized to operate.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when the <i>len</i> is less than or equal to 0.

## G.728 Functions

This section describes Intel IPP functions that can be used in implementing speech codecs following ITU-T\* recommendation G.728 with Annexes I, H [[ITU728](#)].

These functions are used in the Intel IPP *G.728 Speech Encoder-Decoder* sample downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

The list of these functions is given in Table 9-14.

**Table 9-14. Intel IPP G.728 Functions**

Function Base Name	Operation
IIRGetStateSize_G728	Gets the size of IIR state structure to be used.
IIRInit_G728	Initializes the IIR state structure.
IIR_G728	Applies IIR filter to multiple samples.
SynthesisFilterGetStateSize_G728	Gets the size of synthesis filter state structure.
SynthesisFilterInit_G728	Initializes the synthesis filter state structure.
SynthesisFilter_G728	Applies the synthesis filter to multiple samples.
CombinedFilterGetStateSize_G728	Gets the size of combined filter state structure.
CombinedFilterInit_G728	Initializes the combined filter state structure.
CombinedFilter_G728	Applies the combined filter to multiple samples.
PostFilterGetStateSize_G728	Gets the size of the post filter state structure.
PostFilterInit_G728	Initializes the post filter state structure.
PostFilter_G728	Applies the post filter to multiple samples.
PostFilterAdapterGetStateSize_G728	Gets the size of the IppsPostFilterAdapterStatestructure to be used.
PostFilterAdapterInit_G728	Initializes the IppsPostFilterAdapterStatestructure.
LPCInverseFilter_G728	Computes the LPC prediction residual.
PitchPeriodExtraction_G728	Extracts pitch period from the LPC prediction residual.
WinHybridGetStateSize_G728	Gets the size of hybrid windowing module state structure.
WinHybridInit_G728	Initializes the hybrid windowing module state structure.
WinHybrid_G728	Applies the hybrid windowing.
LevinsonDurbin_G728	Calculates LP coefficients from the autocorrelation coefficients.
CodebookSearch_G728	Searches the codebook for the best code vector.
CodebookSearchTCQ_G728	Performs codebook search by Trellis-Coded Quantization.
ImpulseResponseEnergy_G728	Implements shape codebook vector convolution and energy calculation.



## IIRGetStateSize\_G728

*Gets the size of IIR state structure to be used.*

---

### Syntax

```
IppStatus ippsIIR16sGetStateSize_G728_16s (int* pSize);
```

### Parameters

*pSize*                                      Pointer to the output IIR state size value.

### Description

The function `ippsIIR16sGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the IIR filter.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error when the *pSize* pointer is NULL.

## IIR\_Init\_G728

*Initializes the IIR state structure.*

---

### Syntax

```
IppStatus ippsIIR16sInit_G728_16s (IppsIIRState16s_G728_16s* pMem);
```

### Parameters

*pMem*                                      Pointer to the memory allocated for IIR filter.

### Description

The function `ippsIIR16sInit_G728` is declared in `ippsc.h` file. This function initializes the IIR state structure using the given memory block.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error when the *pMem* pointer is NULL.

## IIR\_G728

*Applies IIR filter to multiple samples.*

---

### Syntax

```
IppStatus ippsIIR16s_G728_16s (const Ipp16s* pCoeffs, const Ipp16s*
pSrcQntSpeech, Ipp16s* pDstWgtSpeech, int len, IppsIIRState16s_G728_16s*
pMem);
```

### Parameters

<i>pCoeffs</i>	Pointer to the filter coefficients vector [20]: $b_0, \dots, b_9, a_0, \dots, a_9$ (in Q14).
<i>pSrcQntSpeech</i>	Pointer to the source vector [ <i>len</i> ].
<i>pDstWgtSpeech</i>	Pointer to the destination vector [ <i>len</i> ].
<i>len</i>	The number of source and destination samples.
<i>pMem</i>	Pointer to the IIR filter state structure.

### Description

The function `ippsIIRState16s_G728` is declared in `ippsc.h` file. This function calculates the synthesized speech output by filtering the input quantized speech through the IIR filter one at a time according to the transfer function:

$$\frac{1 + \sum_{i=0}^9 b_i \cdot z^{-i}}{1 + \sum_{i=0}^9 a_i \cdot z^{-i}}$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## SynthesisFilterGetStateSize\_G728

*Gets the size of synthesis filter state structure.*

### Syntax

```
IppStatus ippsSynthesisFilterGetStateSize_G728_16s (int* pSize);
```

### Parameters

*pSize*                      Pointer to the output size value of the synthesis filter state structure.

### Description

The function `ippsSynthesisFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the synthesis filter.

### Return Values

`ippStsNoErr`                Indicates no error.  
`ippStsNullPtrErr`        Indicates an error when the *pSize* pointer is NULL.

## SynthesisFilterInit\_G728

*Initializes the synthesis filter state structure.*

### Syntax

```
IppStatus ippsSynthesisFilterInit_G728_16s (IppsSynthesisFilterState_G728_16s* pMem);
```

### Parameters

*pMem*                      Pointer to the memory allocated for synthesis filter.

### Description

The function `ippsSynthesisFilterInit_G728` is declared in `ippsc.h` file. This function initializes synthesis filter state structure using the given memory block.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is <code>NULL</code> .

## SyntesisFilter\_G728

Applies the synthesis filter to multiple samples.

### Syntax

```

IppStatus ippsSyntesisFilterZeroInput_G728_16s (const Ipp16s* pCoeffs, Ipp16s*
pSrcDstExc, Ipp16s excSfs, Ipp16s* pDstSpeech, Ipp16s* pSpeechSfs,
IppsSynthesisFilterState_G728_16s* pMem);

```

### Parameters

<i>pCoeffs</i>	Pointer to the filter coefficients vector [51]: $a_0, \dots, a_{50}$ in Q14.
<i>pSrcDstExc</i>	Pointer to the input/output gain-scaled excitation vector [5].
<i>excSfs</i>	The input scale of the previous gain-scaled excitation vector.
<i>pDstSpeech</i>	Pointer to the output quantized speech vector [5].
<i>pSpeechSfs</i>	The output scale of the quantized speech vector.
<i>pMem</i>	Pointer to the synthesis filter state structure.

### Description

The function `ippsSyntesisFilterZeroInput_G728` is declared in `ippsc.h` file. This function computes the decoded speech vector as the sum of the zero-input response and the zero-state response of the synthesis filter according to the transfer function:

$$\frac{1}{1 + \sum_{i=1}^{50} a_i \cdot z^{-i}}$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## CombinedFilterGetStateSize\_G728

*Gets the size of combined filter state structure.*

---

### Syntax

```
IppStatus ippCombinedFilterGetStateSize_G728_16s (int* pSize);
```

### Parameters

<i>pSize</i>	Pointer to the output size value of the combined filter state structure.
--------------	--

### Description

The function `ippCombinedFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the combined filter.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is <code>NULL</code> .

## CombinedFilterInit\_G728

*Initializes the combined filter state structure.*

---

### Syntax

```
IppStatus ippCombinedFilterInit_G728_16s (IppsCombinedFilterState_G728_16s* pMem);
```

### Parameters

<i>pMem</i>	Pointer to the memory allocated for the combined filter.
-------------	--

## Description

The function `ippsCombinedFilterInit_G728` is declared in `ippsc.h` file. This function initializes combined filter state structure using the given memory block.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is NULL.

## CombinedFilter\_G728

Applies the combined filter to multiple samples.

## Syntax

```
IppStatus ippsCombinedFilterZeroInput_G728_16s(const Ipp16s* pSyntCoeff,
const Ipp16s* pWgtCoeff, Ipp16s* pDstWgtZIR, IppsCombinedFilterState_G728_16s*
pMem);
```

```
IppStatus ippsCombinedFilterZeroState_G728_16s(const Ipp16s* pSyntCoeff,
const Ipp16s* pWgtCoeff, Ipp16s* pSrcDstExc, Ipp16s excSfs, Ipp16s*
pDstSpeech, Ipp16s* pSpeechSfs, IppsCombinedFilterState_G728_16s* pMem);
```

## Parameters

<i>pSyntCoeff</i>	Pointer to the filter coefficients vector [50]: $a_1, \dots, a_{50}$ in Q14.
<i>pWgtCoeff</i>	Pointer to the filter coefficients vector [20]: $B_1, \dots, B_{10}, A_1, \dots, A_{10}$ in Q14.
<i>pSrcDstExc</i> <i>excSfs</i>	Pointer to the output gain-scaled excitation vector [5]. The input scale of the previous gain-scaled excitation vector.
<i>pDstWgtZIR</i>	Pointer to the output zero input response vector [5] of the combined filter.
<i>pDstSpeech</i> <i>pSpeechSfs</i>	Pointer to the output quantized speech vector [5]. The output scale of the quantized speech vector.
<i>pMem</i>	Pointer to the combined filter state structure.

## Description

The functions `ippsCombinedFilterZeroInput_G728` and `ippsCombinedFilterZeroState_G728` are declared in `ippsc.h` file.

**`ippsCombinedFilterZeroInput_G728`.** This function calculates the zero-input response of the combined filter by superposing two filters, specifically, the 50s-order synthesis filter and the 10s-order IIR filter according to the transfer function:

$$\frac{1}{1 + \sum_{i=1}^{50} a_i \cdot z^{-i}} \cdot \frac{1 + \sum_{i=1}^{10} B_i \cdot z^{-i}}{1 + \sum_{i=1}^{10} A_i \cdot z^{-i}}$$

**`ippsCombinedFilterZeroState_G728_16s`.** This function first performs filtering of the gain-scaled excitation vector through the zero-state combined filter (see the above function). The memory of combined filter is then updated by adding zero-state responses of the synthesis and the IIR filters which it is combined of. The quantized speech output vector is obtained as a by-product of the memory updates.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## PostFilterGetStateSize\_G728

*Gets the size of the post filter state structure.*

### Syntax

```
IppStatus ippsPostFilterGetStateSize_G728_16s (int* pSize);
```

### Parameters

<i>pSize</i>	Pointer to the output size value of the post filter state structure.
--------------	--

## Description

The function `ippsPostFilterGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the post filter.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.

## PostFilterInit\_G728

*Initializes the post filter state structure.*

---

### Syntax

```
IppStatus ippsPostFilterInit_G728_16s (IppsPostFilterState_G728_16s* pMem);
```

### Parameters

*pMem*                                      Pointer to the memory allocated for the post filter.

### Description

The function `ippsPostFilterInit_G728` is declared in `ippsc.h` file. This function initializes the post filter state structure using the given memory block.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is NULL.

## PostFilter\_G728

*Applies the post filter to multiple samples.*

---

### Syntax

```
IppStatus ippsPostFilter_G728_16s (Ipp16s gl, Ipp16s glb, Ipp16s kp, Ipp16s tiltz, const Ipp16s* pCoeffs, const Ipp16s* pSrc, Ipp16s* pDst, IppsPostFilterState_G728_16s* pMem);
```



## Parameters

<i>gl</i>	The LTP scaling factor.
<i>glb</i>	The LTP product term.
<i>kp</i>	The LTP lag, pitch period of the current frame.
<i>tiltz</i>	The STP tilt-compensation coefficient.
<i>pCoeffs</i>	Pointer to the post filter coefficients vector [20]: $B_1, \dots, B_{10}, A_1, \dots, A_{10}$ .
<i>pSrc</i>	Pointer to the input speech vector [5]; elements $-kp, \dots, -1$ must be given as memory of the LTP filter.
<i>pDst</i>	Pointer to the output post-filtered speech vector [5].
<i>pMem</i>	Pointer to the post filter state structure.

## Description

The function `ippsPostFilter_G728` is declared in `ippsc.h` file. This function performs filtering of input samples by one at a time according to the transfer function that is comprised of LTP filter and STP filter parts:

$$gl \cdot (1 - glb \cdot z^{-kp}) \cdot \frac{1 - \sum_{i=1}^{10} B_i \cdot z^{-i}}{1 - \sum_{i=1}^{10} A_i \cdot z^{-i}} \cdot (1 + tiltz \cdot z^{-1})$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## PostFilterAdapterGetStateSize\_G728

*Gets the size of the `IppsPostFilterAdapterState` structure to be used.*

---

### Syntax

```
IppStatus ippsPostFilterAdapterGetStateSize_G728 (int* pSize);
```

### Parameters

<i>pSize</i>	Pointer to the output <code>IppsPostFilterAdapterState</code> structure size value.
--------------	---

### Description

The function `ippsPostFilterAdapterGetStateSize_G728` is declared in `ippsc.h` file.

This function returns the minimal size of memory to be allocated for proper use of the postfilter adapter.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is <code>NULL</code> .

## PostFilterAdapterStateInit\_G728

*Initializes the `IppsPostFilterAdapterState` structure.*

---

### Syntax

```
IppStatus ippsPostFilterAdapterStateInit_G728( IppsPostFilterAdapterState* pMem );
```

### Parameters

<i>pMem</i>	Pointer to the memory allocated for postfilter adapter.
-------------	---

## Description

The function `ippsPostFilterAdapterStateInit_G728` is declared in `ippsc.h` file. This function initializes the `IppsPostFilterAdapterState` structure using the memory block given by `pMem`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMem</code> pointer is <code>NULL</code> .

## LPCInverseFilter\_G728

*Computes the LPC prediction residual.*

---

## Syntax

```
IppStatus ippsLPCInverseFilter_G728_16s (const Ipp16s* pSrcSpeech, const
Ipp16s* pCoeffs, Ipp16s* pDstResidual, IppsPostFilterAdapterState_G728*
pMem);
```

## Parameters

<code>pSrcSpeech</code>	Pointer to the quantized speech buffer [-240...0...4].
<code>pCoeffs</code>	Pointer to the 10 <sup>th</sup> -order LPC filter coefficients [10].
<code>pDstResidual</code>	Pointer to the LPC prediction residual vector [-140...0...99].
<code>pMem</code>	Pointer to the postfilter adapter memory.

## Description

The function `ippsLPCInverseFilter_G728_16s` is declared in `ippsc.h` file. This function computes the LPC prediction residual vector for the current decoded speech vector. The 10<sup>th</sup>-order LPC inverse filter implements the following function:

$$A(z) = 1 - \sum_{i=0}^9 a_i \cdot z^{-(i+1)}$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

## PitchPeriodExtraction\_G728

*Extracts pitch period from the LPC prediction residual.*

---

### Syntax

```
IppStatus ippSPitchPeriodExtraction_G728_16s (const Ipp16s* pSrcResidual,
int* pPitchPeriod, IppsPostFilterAdapterState_G728* pMem);
```

### Parameters

<i>pSrcResidual</i>	Pointer to the LPC undecimated residual vector <code>d[-140,99]</code> .
<i>pPitchPeriod</i>	Pointer to the pitch period of the previous frame.
<i>pMem</i>	Pointer to the postfilter adapter memory.

### Description

The function `ippSPitchPeriodExtraction_G728` is declared in `ippsc.h` file. The function extracts the pitch period once per frame. It assumes that the input vector contains the LPC prediction residual of previous frames (samples `[-139,80]`) and of four vectors of the current frame stored, respectively, in elements `[80,84]`, `[85,89]`, `[90,94]`, and `[95,99]`. The prediction history for the previous frame is stored in the elements `[-139,80]`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## WinHybridGetStateSize\_G728

*Gets the size of hybrid windowing module state structure.*

---

### Syntax

```
IppStatus ippsWinHybridGetStateSize_G728_16s (int M, int L, int N, int DIM,
int* pSize);
```

### Parameters

<i>M</i>	The input length of the LPC window.
<i>L</i>	The input adaptation cycle size in samples.
<i>N</i>	The input number of non-recursive window samples.
<i>DIM</i>	The input block size used for block scaling of the input speech by the function <code>ippsWinHybrid_G728</code> .
<i>pSize</i>	Pointer to the output size value of the hybrid windowing module state structure.

### Description

The function `ippsWinHybridGetStateSize_G728` is declared in `ippsc.h` file. This function returns the minimal size of memory to be allocated for proper use of the hybrid windowing module according to the given window parameters.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>L</i> , <i>M</i> or <i>N</i> are less than or equal to zero.

## WinHybridInit\_G728

*Initializes the hybrid windowing module state structure.*

---

### Syntax

```
IppStatus ippsWinHybridInit_G728_16s (const Ipp16s* pWinTab, int M, int L,
int N, int DIM, Ipp16s a2L, IppsWinHybridState_G728_16s* pMem);
```

## Parameters

<i>pWinTab</i>	The input vector of windowing coefficients.
<i>M</i>	The input length of LPC window. Must be not less than 10.
<i>L</i>	The input adaptation cycle size in samples.
<i>N</i>	The input number of non-recursive window samples.
<i>a<sup>2L</sup></i>	The $a^{2L}$ multiple used in calculation of the recursive component in hybrid windowing module.
<i>N</i>	The input number of non-recursive window samples.

## Description

The function `ippsWinHybridInit_G728` is declared in `ipps.h` file. This function initializes the hybrid windowing module state structure using the given memory block. The recursive component and the previous speech samples are zeroed.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pMem</i> pointer is NULL.

## WinHybrid\_G728

Applies the hybrid windowing.

### Syntax

```

IppStatus ippsWinHybridBlock_G728_16s(Ipp16s bfi, const Ipp16s* pSrc, const
Ipp16s* pSrcSfs, Ipp16s* pDst, IppsWinHybridState_G728_16s* pMem);

IppStatus ippsWinHybrid_G728_16s(Ipp16s bfi, const Ipp16s* pSrc, const Ipp16s*
pSrcSfs, Ipp16s* pDst, IppsWinHybridState_G728_16s* pMem);

```

### Parameters

<i>bfi</i>	The input bad frame indicator: "1" signifies a bad frame, any other value signifies a good frame.
<i>pSrc</i>	Pointer to the input speech vector [20].

<i>pSrcSfs</i>	Pointer to the input vector [5] of scale factors for elements of the input speech vector. Each element of the <i>pSrcSfs</i> vector may specify a separate scale factor for a block of input speech vector elements. Thus, <i>pSrcSfs</i> [0] is the scale factor of the elements <i>pSrc</i> [0],.. <i>pSrc</i> [ <i>DIM</i> -1]; <i>pSrcSfs</i> [1] is the scale factor for <i>pSrc</i> [ <i>DIM</i> ],.. <i>pSrc</i> [2* <i>DIM</i> -1], and so on, where <i>DIM</i> is a block length that must be defined by the function <a href="#">ippsWinHybridInit_G728</a> .
<i>pDst</i>	Pointer to the output reflection coefficients vector [M+1], where M is the length of the LPC window defined in the function <a href="#">ippsWinHybridInit_G728</a> .
<i>pMem</i>	Pointer to the post filter state structure.

## Description

The function `ippsWinHybrid_G728` is declared in `ippsc.h` file. This function first applies the window to the input speech vector and then calculates the autocorrelation coefficients needed in LPC analysis by the formula:

$$R_m(i) = r_m(i) + \sum_{k=m-N}^{m-1} s_m(k) s_m(k-i), m = 0, \dots, M$$

where the recursive component of adaptation cycle is calculated as follows:

$$r_m(i) = \alpha^{2L} \cdot r_{m-L}(i) + \sum_{k=m-L-N}^{m-N-1} s_m(k) s_m(k-i)$$

The recursive part is calculated using the data calculated in previous adaptation cycle and stored in module memory.

A *white noise correction* is applied to autocorrelation coefficients by increasing the energy as follows:

$$r_m(0) = 257/256 \cdot r_m(0)$$

The recursive component and previous speech samples are stored in the module memory and may be used in the next adaptation cycle.

If the bad frame indicator is on, then only 10 autocorrelation coefficients are calculated and output.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## LevinsonDurbin\_G728

*Calculates LP coefficients from the autocorrelation coefficients.*

---

### Syntax

```

IppStatus ippLevinsonDurbin_G728_16s_Sfs(const Ipp16s* pSrcAutoCorr, int
order, Ipp16s* pDstLPC, Ipp16s* pDstResidualEnergy, Ipp16s* pDstScaleFactor);

IppStatus ippLevinsonDurbin_G728_16s_ISfs(const Ipp16s* pSrcAutoCorr, int
numSrcLPC, int order, Ipp16s* pSrcDstLPC, Ipp16s* pSrcDstResidualEnergy,
Ipp16s* pSrcDstScaleFactor);

```

### Parameters

<code>pSrcAutoCorr</code>	Pointer to the input autocorrelation coefficients vector [ <code>order+1</code> ].
<code>order</code>	The input number of LP coefficients to calculate.
<code>numSrcLPC</code>	The input number of pre-calculated LPCs.
<code>pDstLPC</code>	Pointer to the output LPC vector [ <code>order</code> ].
<code>pSrcDstLPC</code>	Pointer to the input/output LPC vector [ <code>order</code> ].
<code>pDstResidualEnergy</code>	Pointer to the output residual energy.
<code>pSrcDstResidualEnergy</code>	Pointer to the input/output residual energy of the pre-calculated LPC.
<code>pDstScaleFactor</code>	Pointer to the output scale factor of the LPC vector.
<code>pSrcDstScaleFactor</code>	Pointer to the input scale factor of the pre-calculated LPC and the output scale factor of the output LPC vector.



## Description

The function `ippsLevinsonDurbin_G728` is declared in `ippsc.h` file.

**`ippsLevinsonDurbin_G728_16s_sfs`**. This function may be used to calculate Linear Prediction (LP) coefficients by solving the following set of linear equations:

$$\sum_{i=0}^{order-1} a_i \cdot r(i-k) = r(k) \quad k = 1, 2, \dots, order$$

where  $a_i$ ,  $i = 0, 1, \dots, order-1$  are the LP coefficients to be calculated and stored in the output LPC vector. The description of the Levinson-Durbin algorithm used by this function may be found in the description of the function [ippsLevinsonDurbin\\_G729](#).

Both `ippsLevinsonDurbin_G728` and `ippsLevinsonDurbin_G729B` calculate mathematically the same, but not bit exact LPCs. The difference is that the `ippsLevinsonDurbin_G729B` function outputs LPCs in Q12, while the `ippsLevinsonDurbin_G728` function automatically rescales the LPCs if overflow occurs.

**`ippsLevinsonDurbin_G728_16s_1sfs`**. This function may be used to continue Levinson-Durbin recursion for bigger *order*. The LPCs, their scale factor and the residual energy of the previous recursion and the additional autocorrelation coefficients are used to resume recursion.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>order</i> is less than or equal to 0, or when $order < numSrcLPC$ for the in-place function.

## CodebookSearch\_G728

*Searches the codebook for the best code vector.*

### Syntax

```
IpplStatus ippsCodebookSearch_G728_16s(const Ippl16s* pSrcCorr, const Ippl16s*
pSrcEnergy, int* pDstShapeIdx, int* pDstGainIdx, short* pDstCodebookIdx,
IpplSpchBitRate rate);
```

## Parameters

<i>pSrcCorr</i>	Pointer to the input vector [5] of time-reversed convolution of the target signal.
<i>pSrcEnergy</i>	Pointer to the input vector [128] of the energy of convolved shape codevector.
<i>pDstShapeIdx</i>	Pointer to the output best 7-bit shape codebook index.
<i>pDstGainIdx</i>	Pointer to the output best 3-bit gain codebook index.
<i>pDstCodebookIdx</i>	Pointer to the output best codebook index to be transmitted.
<i>rate</i>	Input coding bit rate.

## Description

The function `ippsCodebookSearch_G728` is declared in `ippsc.h` file. This function implements the “Error calculator and best codebook index selector” block used to search through the gain codebook and the shape codebook for the best combination of the gain and shape codebook indexes.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .
<code>IppStsRangeErr</code>	Indicates an error when <i>rate</i> is not one of the acceptable values <code>IPP_SPCHBR_16000</code> , <code>IPP_SPCHBR_12800</code> or <code>IPP_SPCHBR_9600</code> for the 16, 12.8 or 9.6 Kbit/s coding bit rates, respectively.

## CodebookSearchTCQ\_G728

*Performs codebook search by trellis-coded quantization.*

---

### Syntax

```

IppStatus ippsCodebookSearchTCQ_G728_16s8u (const Ipp16s pSrc[5], const
Ipp16s pSrcLPC[10], Ipp8u pDst[5], Ipp16s pSrcDstWindow[20], Ipp16s
pSrcDstResidual[8], Ipp32s* pSrcDstBestNode, Ipp32s_EC_Sfs invGain,
Ipp32s_EC_Sfs excGain);

```

## Parameters

<i>pSrc</i>	Pointer to the input speech frame vector.
<i>pSrcLPC</i>	Pointer to the LPC coefficients vector.
<i>pDst</i>	Pointers to the output vector containing the indexes of the “best path”.
<i>pSrcDstWindow</i>	Pointer to the synthesis filter hybrid window vector.
<i>pSrcDstResidual</i>	Pointer to the input/output quantized residuals vector.
<i>pSrcDstBestNode</i>	Pointer to input/output trellis survivor node value.
<i>invGain</i>	Inverted gain scaled value, structure with gain value and its scale.
<i>excGain</i>	Linear excitation gain scaled value.

## Description

The `ipps ippsCodebookSearchTCQ_G728` function is declared in the `ippsc.h` file. The function performs codebook search by the trellis-coded quantization (TCQ) for given gains. TCQ trellis is parsed for best path and survivor node. For every next stage one of the two incoming branches is selected and marked as 'new' survivor. The final trellis survivor node and best path indexes are stored in *pSrcDstBestNode* and *pDst* respectively. For detailed description of the TCQ see [ITU728] Annex J, clause 4.1.1.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## ImpulseResponseEnergy\_G728

*Implements shape codebook vector convolution and energy calculation.*

---

## Syntax

```
Ippl6s* ippsImpulseResponseEnergy_G728_16s(const Ippl6s* pSrcImpResp,
Ippl6s* pDstEnergy);
```

## Parameters

<i>pSrcImpResp</i>	Pointer to the input impulse response vector [5] of the synthesis and weighted filter.
<i>pDstEnergy</i>	Pointer to the output energy vector [128] of the convolved shape codevector.

## Description

The function `ippsImpulseResponseEnergy_G728` is declared in `ippsc.h` file. This function implements the “*Shape codevector convolution and energy calculator*” block used to calculate the energy of the convolved shape codevector.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

# Voice Enhancement Functions

This section describes the Intel IPP functions for echo control, noise removal, automatic audio level control.

The complete list of the Intel IPP voice enhancement functions is given in Table 9-15.

**Table 9-15. Intel IPP Voice Enchantment Functions**

Function Base Name	Operation
Echo Canceller Functions	
<code>SubbandProcessGetSize</code>	Calculates size of the subband process state structure.
<code>SubbandProcessInit</code>	Initializes the subband process state structure.
<code>SubbandAnalysis</code>	Decomposes a frame into a complex subband representation
<code>SubbandSynthesis</code>	Reconstructs frame from a complex subband representation.
<code>SubbandControllerGetSize_EC</code>	Calculates size of the subband controller state structure.
<code>SubbandControllerInit_EC</code>	Initializes the subband controller state structure.
<code>SubbandControllerReset_EC</code>	Resets subband controller state.
<code>SubbandController_EC</code>	Updates filter coefficients and returns output gain coefficients.
<code>SubbandControllerUpdate_EC</code>	Updates controller state and returns the step sizes.

Function Base Name	Operation
<code>SubbandControllerDTGetSize_EC</code>	Calculates size of the subband DT controller state structure.
<code>SubbandControllerDTInit_EC</code>	Initializes the subband DT controller state structure.
<code>SubbandControllerDTReset_EC</code>	Resets subband DTcontroller state.
<code>SubbandControllerDT_EC</code>	Updates filter coefficients and returns output gain coefficients.
<code>SubbandControllerDTUpdate_EC</code>	Updates DT controller state and returns the step sizes.
<code>ToneDetectGetStateSize_EC</code>	Calculates size of the tone detector state structure.
<code>ToneDetectInit_EC</code>	Initializes the tone detector state structure.
<code>ToneDetect_EC</code>	Detects the signal of 2100 Hz frequency with every 450 ms phase reversal.
<code>FullbandControllerGetSize_EC</code>	Calculates size of the fullband controller state structure.
<code>FullbandControllerInit_EC</code>	Initializes the fullband controller state structure.
<code>FullbandControllerUpdate_EC</code>	Updates the fullband controller state and returns the step sizes.
<code>FullbandController_EC</code>	Updates filter coefficients and returns output gain coefficients.
<code>FullbandControllerReset_EC</code>	Resets fullband controller state.
<code>FIR_EC</code>	Computes FIR filter results.
<code>FIRSubband_EC, FIRSubbandLow_EC</code>	Computes the frequency-domain adaptive filter output.
<code>FIRSubbandCoeffUpdate_EC, FIRSubbandLowCoeffUpdate_EC</code>	Updates the adaptive filter coefficients.
<code>NLMS_EC</code>	Performs FIR filtering with coefficients update.
<b>Noise Reduction Functions</b>	
<code>FilterNoiseGetStateSize</code>	Calculates the size of the state structure for the noise reduction filter.
<code>FilterNoiseInit</code>	Initializes the state structure for the noise reduction filter.
<code>FilterNoiseLevel</code>	Sets the amplifier of the noise reduction filter.
<code>FilterNoiseDetect_EC</code>	Performs the noise detection.
<code>FilterNoiseDetectModerate_EC</code>	Performs the noise detection on signals with echo.
<code>FilterNoiseSetMode_EC</code>	Sets type of the smoothing filter.
<code>FilterNoise</code>	Performs the noise reduction filtering.
<b>Automatic Audio Level Control Functions</b>	
<code>ALCGetStateSize_G169</code>	Returns the size of the ALC state structure.
<code>ALCInit_G169</code>	Initializes ALC state structure.
<code>ALCSetLevel_G169</code>	Sets ALC algorithm target and clipping levels.

Function Base Name	Operation
<a href="#">ALCSetGain_G169</a>	Sets ALC algorithm maximum gain value.
<a href="#">ALC_G169</a>	Performs the automatic level control.

## Echo Canceller Functions

The functions described in this section implement building blocks that can be used to create an acoustic and network echo cancellers compliant to the appropriate ITU-T Recommendations: G.168-2000, and G.167 (superseded by G.161, P.340).

## SubbandProcessGetSize

*Calculates size of the subband process state structure.*

### Syntax

```
IppStatus ippsSubbandProcessGetSize_32f(int order, int windowLen, int*
pStateSize, int* pInitBufSize, int* pBufSize);

IppStatus ippsSubbandProcessGetSize_16s(int order, int windowLen, int*
pStateSize, int* pInitBufSize, int* pBufSize);
```

### Parameters

<i>order</i>	Number of subbands is equal to $2^{order - 1} + 1$ .
<i>windowLen</i>	Length of window.
<i>pStateSize</i>	Pointer to the computed value of buffer size for the subband process state structure.
<i>pInitBufSize</i>	Pointer to the computed buffer size for use in the initialization function.
<i>pBufSize</i>	Pointer to the computed size of work buffer.

### Description

The functions `ippsSubbandProcessGetSize` are declared in the `ippsc.h` file. These functions calculate the sizes of memory buffers required for the functions [ippsSubbandAnalysis](#) and [ippsSubbandSynthesis](#) to operate.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>order</i> is less than or equal to 0 or when <i>windowLen</i> is not divisible by $2^{\text{order}}$ .

## SubbandProcessInit

*Initializes the subband process state structure.*

### Syntax

```
IppStatus ippsSubbandProcessInit_32f(IppsSubbandProcessState_32f* pState,
int order, int frameSize, int windowLen, const Ipp32f* pWindow, Ipp8u*
pInitBuf);
```

```
IppStatus ippsSubbandProcessInit_16s(IppsSubbandProcessState_16s* pState,
int order, int frameSize, int windowLen, const Ipp32s* pWindow, Ipp8u*
pInitBuf);
```

### Parameters

<i>pState</i>	Pointer to the subband process state structure.
<i>order</i>	Number of subbands is equal to $2^{\text{order} - 1} + 1$ .
<i>frameSize</i>	Size of frame. Must be in range from 1 to $2^{\text{order}}$ .
<i>windowLen</i>	Window length.
<i>pWindow</i>	Pointer to window coefficients . May be <code>NULL</code> if the <i>order</i> , <i>frameSize</i> and <i>windowLen</i> are equal to one of the predefined sets: (5, 24, 128) or (6, 44, 256). In this case the predefined window is used.
<i>pInitBuf</i>	Pointer to the initialized buffer.

### Description

The functions `ippsSubbandProcessInit` are declared in the `ippsc.h` file. These functions initialize the state structure for the subband process in the external buffer. The size of this buffer must be calculated by the function `ippsSubbandProcessGetSize` beforehand. The functions `ippsSubbandAnalysis` and `ippsSubbandSynthesis` use this state structure in their operation.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>order</i> is less or equal to 0, or when <i>windowLen</i> is not divisible by $2^{\textit{order}}$ , or when <i>frameSize</i> is less than or equal to 0, or when <i>pWindow</i> is <code>NULL</code> and the appropriate internal window does not exist.

## SubbandAnalysis

*Decomposes a frame into a complex subband representation.*

---

### Syntax

```
IppStatus ippsSubbandAnalysis_32f32fc(const Ipp32f* pSignal, Ipp32fc*
pSubbands, IppsSubbandProcessState_32f* pState, Ipp8u* pBuf);

IppStatus ippsSubbandAnalysis_16s32sc_Sfs(const Ipp16s* pSignal, Ipp32sc*
pSubbands, IppsSubbandProcessState_16s* pState, int scaleFactor, Ipp8u*
pBuf);
```

### Parameters

<i>pState</i>	Pointer to the subband process state structure.
<i>pSignal</i>	Pointer to the source vector. Vector length must be equal to the frame size which was used to initialize the subband process algorithm by the function <a href="#">ippsSubbandProcessInit</a> .
<i>pSubbands</i>	Pointer to the resulting subband vector. Vector length is equal to the number of subbands calculated as $2^{\textit{order}-1} + 1$ , where <i>order</i> is specified in the function <a href="#">ippsSubbandProcessInit</a> .
<i>pBuf</i>	Pointer to the work buffer of size specified by the parameter <i>pBufSize</i> in the <code>ippsSubbandProcessGetSize</code> function. This buffer must be allocated by the user.



## Description

The functions `ippsSubbandAnalysis` are declared in the `ippsc.h` file. These functions perform subband analysis using the oversampled DFT filter bank.

Computation steps are as follows:

1) Calculate the weighted sum as

$$X(j) = \sum_{i=0}^{windowLen/fftLen-1} window(i \cdot fftLen + j) \cdot x(i \cdot fftLen + j)$$

for  $j = 0, \dots, fftLen - 1$ ,  $fftLen = 2$  order.

Here  $x$  denotes the input sample buffer of size `windowLen`.

2) Calculate and return the Fast Fourier Transform of  $X$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## SubbandSynthesis

*Reconstructs frame from a complex subband representation.*

---

### Syntax

```
IppStatus ippsSubbandSynthesis_32fc32f(const Ipp32fc* pSubbands, Ipp32f*
pSignal, IppsSubbandProcessState_32f* pState, Ipp8u* pBuf);
```

```
IppStatus ippsSubbandSynthesis_32sc16s_Sfs(const Ipp32sc* pSubbands, Ipp16s*
pSignal, IppsSubbandProcessState_16s* pState, int scaleFactor, Ipp8u* pBuf);
```

### Parameters

<code>pState</code>	Pointer to the subband process state structure.
---------------------	---

<i>pSubbands</i>	Pointer to the resulting subband vector. Vector length is equal to the number of subbands calculated as $2^{order-1} + 1$ , where <i>order</i> is specified in the function <code>ippsSubbandProcessInit</code> .
<i>pSignal</i>	Pointer to the destination signal vector. Vector length is equal to frame size which was used to initialize the subband process algorithm by the function <code>ippsSubbandProcessInit</code> .
<i>pBuf</i>	Pointer to the work buffer of size computed by the function <code>ippsSubbandProcessGetSize</code> (returned via <i>pBufSize</i> parameter).

## Description

The functions `ippsSubbandSynthesis` are declared in the `ippsc.h` file. These functions perform subband synthesis using the oversampled DFT filter bank.

Computation steps are as follows:

- 1) Calculates *IX* as the inverse Fast Fourier Transform of the input subband sample *X*. Transform length is  $fftLen = 2^{order}$ .
- 2) Shifts internal buffer by *frameSize* and sets to zero the undefined buffer values.
- 3) Updates buffer values as  

$$x(j) := x(j) + window(j) * IX(j \% fftLen),$$
for  $j = 0, \dots, windowLen - 1$
- 4) Returns most recent *frameSize* samples from the buffer.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## SubbandControllerGetSize\_EC

*Calculates size of the subband controller state structure.*

---

### Syntax

```
IppStatus ippsSubbandControllerGetSize_EC_32f(int numSubbands, int frameSize,  
int numSegments, ippECFrequency sampleFreq, int* pSize);  
  
IppStatus ippsSubbandControllerGetSize_EC_16s(int numSubbands, int frameSize,  
int numSegments, ippECFrequency sampleFreq, int* pSize);
```

### Parameters

<i>numSubbands</i>	Number of subbands.
<i>frameSize</i>	Size of the frame.
<i>numSegments</i>	Number of segments.
<i>sampleFreq</i>	Sample frequency.
<i>pSize</i>	Pointer to the computed buffer size value.

### Description

The functions `ippsSubbandControllerGetSize_EC` are declared in the `ippsc.h` file. These functions calculate the sizes of memory required for the function `ippsSubbandControllerUpdate_EC` and `ippsSubbandController_EC` to operate.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>numSubbands</i> is less than or equal to 0, or <i>numSegment</i> is less than or equal to 0, or <i>frameSize</i> is less than or equal to 0.

## SubbandControllerInit\_EC

Initializes the subband controller state structure.

### Syntax

```
IppStatus ippsSubbandControllerInit_EC_32f(IppsSubbandControllerState_EC_32f*  
pState, int numSubbands, int frameSize, int numSegments, ippECFrequency  
sampleFreq);
```

```
IppStatus ippsSubbandControllerInit_EC_16s(IppsSubbandControllerState_EC_16s*  
pState, int numSubbands, int frameSize, int numSegments, ippECFrequency  
sampleFreq);
```

### Parameters

<i>pState</i>	Pointer to the subband controller state structure.
<i>numSubbands</i>	Number of subbands.
<i>frameSize</i>	Size of the frame.
<i>numSegments</i>	Number of segments.
<i>sampleFreq</i>	Sample frequency.

### Description

The functions `ippsSubbandControllerInit_EC` are declared in the `ippsc.h` file. These functions initialize the state structure and the internal data of the subband controller algorithm. The functions `ippsSubbandControllerUpdate_EC` and `ippsSubbandController_EC` use this memory in their operation.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>numSubbands</i> is less than or equal to 0, or <i>numSegments</i> is less than or equal to 0, or <i>frameSize</i> is less than or equal to 0.

## SubbandControllerUpdate\_EC

Updates controller state and returns the step sizes.

### Syntax

```
IppStatus ippsSubbandControllerUpdate_EC_32f(const Ipp32f* pSrcRin, const
Ipp32f* pSrcSin, const Ipp32fc** ppSrcRinSubbandsHistory, const Ipp32fc*
pSrcSinSubbands, double* pDstStepSize, IppsSubbandControllerState_EC_32f*
pState);
```

```
IppStatus ippsSubbandControllerUpdate_EC_16s(const Ipp16s* pSrcRin, const
Ipp16s* pSrcSin, const Ipp32sc** ppSrcRinSubbandsHistory, const Ipp32sc*
pSrcSinSubbands, IppAECscaled32s* pDstStepSize,
IppsSubbandControllerState_EC_16s* pState);
```

### Parameters

<i>pState</i>	Pointer to to the subband controller state structure.
<i>pSrcRin</i>	Pointer to receive-in signal frame.
<i>pSrcSin</i>	Pointer to send-in signal frame. Frame size is specified in the function <a href="#">ippsSubbandControllerInit_EC</a> .
<i>ppSrcRinSubbandsHistory</i>	Pointer to an array of pointers to the most recent receive-in blocks. Size of the array is equal to <i>numSegments</i> specified in the function <a href="#">ippsSubbandControllerInit_EC</a> .
<i>pSrcSinSubbands</i>	Pointer to subband representation of send-in signal frame (or NULL). Size of the array is equal to <i>numSubbands</i> specified in the function <a href="#">ippsSubbandControllerInit_EC</a> .
<i>pDstStepSize</i>	Pointer to the vector of step sizes. Vector length is equal to <i>numSubbands</i> specified in the function <a href="#">ippsSubbandControllerInit_EC</a> .

### Description

The functions `ippsSubbandControllerUpdate_EC` are declared in the `ippsc.h` file. These functions update the internal state of the subband controller and return the step size values for further use in the adaptation of the coefficients.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## SubbandController\_EC

*Updates filter coefficients and returns output gain coefficients.*

---

### Syntax

```
IppStatus ippsSubbandController_EC_32f(const Ipp32fc* pSrcAdaptiveFilterErr,
const Ipp32fc* pSrcFixedFilterErr, Ipp32fc** ppDstAdaptiveCoefs, Ipp32fc**
ppDstFixedCoefs, Ipp32f* pDstSGain, IppsSubbandControllerState_EC_32f*
pState);
```

```
IppStatus ippsSubbandController_EC_16s(const Ipp32sc* pSrcAdaptiveFilterErr,
const Ipp32sc* pSrcFixedFilterErr, Ipp32sc** ppDstAdaptiveCoefs, Ipp32sc**
ppFixedCoefs, Ipp32s* pDstSGain, IppsSubbandControllerState_EC_16s* pState);
```

### Parameters

<code>pState</code>	Pointer to the subband state structure.
<code>pSrcAdaptiveFilterErr</code>	Pointer to the adaptive filter error vector. Vector length is equal to <i>numSubbands</i> , specified in the function <a href="#">ippsSubbandControllerInit_EC</a> .
<code>pSrcFixedFilterErr</code>	Pointer to the fixed filter error vector. Vector length is equal to <i>numSubbands</i> .
<code>ppDstAdaptiveCoefs</code>	Pointer to an array of pointers to the adaptive filter coefficients vectors. Size of the array is equal to <i>numSegments</i> specified in the function <a href="#">ippsSubbandControllerInit_EC</a> .
<code>ppDstFixedCoefs</code>	Pointer to an array of pointers to the fixed filter coefficients vectors. Size of the array is equal to <i>numSegments</i> specified in the function <a href="#">ippsSubbandControllerInit_EC</a> .
<code>pDstSGain</code>	Pointer to the send gain coefficient.

## Description

The functions `ippsSubbandController_EC` are declared in the `ippsc.h` file.

These functions compare powers of adaptive filter error and fixed filter error and track changes of adaptive filter coefficients power.

If the adaptive filter has a significantly smaller error and is stable (which is indicated by coefficients power changing slowly), its coefficients are copied to the fixed filter. This case corresponds to “*no double-talk*” mode.

If the fixed filter has a significantly smaller error, its coefficients are copied to the adaptive filter. This case corresponds to the double-talk mode.

Send gain coefficient calculation (non-linear processor technology) is based on presence of the double-talk mode and changes of receive-in and send-in signal powers.

The subband controller state structure `pState` must be initialized by the function `ippsSubbandControllerInit_EC` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## SubbandControllerReset\_EC

*Resets the subband controller state.*

---

### Syntax

```
IppStatus ippsSubbandControllerReset_EC_32f(IppsSubbandControllerState_EC_32f* pState);
```

```
IppStatus ippsSubbandControllerReset_EC_16s(IppsSubbandControllerState_EC_16s* pState);
```

### Parameters

<code>pState</code>	Pointer to the subband controller state structure.
---------------------	--

## Description

The functions `ippsSubbandControllerReset_EC` are declared in the `ippsc.h` file. These functions reset the subband controller state. You can use these functions after an interruption in adaptation process occurred.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.

## SubbandControllerDTGetSize\_EC

*Calculates the size of the subband DT controller state structure.*

---

### Syntax

```
IppStatus ippsSubbandControllerDTGetSize_EC_16s(int numSubbands, int
frameSize, int numSegments, ippPCMFrequency sampleFreq, int* pSize);
```

### Parameters

<i>numSubbands</i>	Number of subbands.
<i>frameSize</i>	Size of the frame.
<i>numSegments</i>	Number of segments.
<i>sampleFreq</i>	Sample frequency.
<i>pSize</i>	Pointer to the computed buffer size value.

### Description

The function `ippsSubbandControllerDTGetSize_EC` is declared in the `ippsc.h` file. This function calculates the size of memory required for the state structure and the internal data of the subband DT controller algorithm.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is NULL.



`ippStsBadArgErr` Indicates an error when *numSubbands* is less than or equal to 0, or *numSegment* is less than or equal to 0, or *frameSize* is less than or equal to 0.

## SubbandControllerDTInit\_EC

*Initializes the subband DT controller state structure.*

---

### Syntax

```
IppStatus
ippsSubbandControllerDTInit_EC_16s(IppsSubbandControllerDTState_EC_16s*
pState, int numSubbands, int frameSize, int numSegments, ippPMCFrequency
sampleFreq);
```

### Parameters

<i>pState</i>	Pointer to the subband controller DT state structure to be created.
<i>numSubbands</i>	Number of subbands.
<i>frameSize</i>	Size of the frame.
<i>numSegments</i>	Number of segments.
<i>sampleFreq</i>	Sample frequency.

### Description

The functions `ippsSubbandControllerDTInit_EC` are declared in the `ippsc.h` file.

This function initializes the state structure and the internal data of the subband DT controller algorithm. The size of the required memory must be computed using the function `beforehandippsSubbandControllerDTGetSize_EC`.

This state structure is used by the functions `ippsSubbandControllerDT_EC` and `ippsSubbandControllerDTUpdate_EC`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointer <i>pState</i> is NULL.

`ippStsBadArgErr` Indicates an error when *numSubbands* is less than or equal to 0, or *numSegments* is less than or equal to 0, or *frameSize* is less than or equal to 0.

## SubbandControllerDTReset\_EC

*Resets the subband DT controller state.*

---

### Syntax

```
IppStatus
ippsSubbandControllerDTReset_EC_16s( IppsSubbandControllerState_EC_16s*
pState );
```

### Parameters

*pState* Pointer to the subband controller DT state structure.

### Description

The function `ippsSubbandControllerDTReset_EC` is declared in the `ippsc.h` file. This function resets the subband controller state. It can be used after an interruption in adaptation process occurred.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the pointer *pState* is NULL.

## SubbandControllerDT\_EC

*Updates filter coefficients and returns output gain coefficients.*

---

### Syntax

```
IppStatus ippsSubbandControllerDT_EC_16s(const Ipp32sc* pSrcAdaptiveFilterErr,
const Ipp32sc* pSrcFixedFilterErr, Ipp32sc** ppDstAdaptiveCoefs, Ipp32sc**
ppDstFixedCoefs, Ipp64s* pSrcDstFilterPwr, int* pDstStsAdapt, Ipp64s pwrDelta,
int filterUpdateEnabled, int adaptationEnabled, int startSubband,
IppsSubbandControllerDTState_EC_16s* pState);
```

## Parameters

<i>pState</i>	Pointer to state structure of the subband DT controller.
<i>pSrcAdaptiveFilterErr</i>	Pointer to the input adaptive filter error vector. Vector length is equal to <i>numSubbands</i> , specified in the function <a href="#">ippsSubbandControllerDTInit_EC</a> .
<i>pSrcFixedFilterErr</i>	Pointer to the input fixed filter error vector. Vector length is equal to <i>numSubbands</i> .
<i>ppDstAdaptiveCoefs</i>	Pointer to an array of pointers to the adaptive filter coefficients vectors. Size of the array is equal to <i>numSegments</i> specified in <a href="#">ippsSubbandControllerDTInit_EC</a> .
<i>ppDstFixedCoefs</i>	Pointer to an array of pointers to the fixed filter coefficients vectors. Size of the array is equal to <i>numSegments</i> specified in <a href="#">ippsSubbandControllerDTInit_EC</a> .
<i>pSrcDstFilterPwr</i>	Pointer to an input/output power of adaptive filter coefficient.
<i>pDstStsAdapt</i>	<p>Pointer to output flag which shows the state of the controller.</p> <p><i>pDstStsAdapt</i> = -1 - coefficients of the adaptive filter have been copied to the fixed filter;</p> <p><i>pDstStsAdapt</i> = 0 - coefficients of the adaptive and fixed filters are not changed;</p> <p><i>pDstStsAdapt</i> = 1 - coefficients of the fixed filter have been copied to the fixed filter.</p>
<i>pwrDelta</i>	Difference between the power of the adaptive filter coefficients of the current frame and the power of the adaptive filter coefficients of the previous frame.
<i>filterUpdateEnabled</i>	Difference between the power of the adaptive filter coefficients of the current frame and the power of the adaptive filter coefficients of the previous frame.
<i>adaptationEnabled</i>	Indicates if an adaptation is enabled or not.
<i>startSubband</i>	Indicates the number of subband the filtering starts from ( $0 \leq \text{startSubband} < \text{numSubbands}$ ).

## Description

The function `ippsSubbandControllerDT_EC` is declared in the `ippsc.h` file.

This function compares powers of the adaptive filter error and fixed filter error, and tracks changes of the adaptive filter coefficients power.

If the adaptive filter has a significantly smaller error and is stable (which is indicated by coefficients power changing slowly), its coefficients are copied to the fixed filter. If the fixed filter has a significantly smaller error or adaptive filter is not stable (which is indicated by coefficients power changing rapidly), its coefficients are copied to the adaptive filter. Output flag `pDstStsAdapt` shows if there were conservations or restorations of the adaptive filter coefficients.

The subband DT controller state structure `pState` must be initialized by the function `ippsSubbandControllerDTInit_EC` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <code>startSubband</code> is less than 0 or greater than or equal to the <code>numSubbands</code> .

## SubbandControllerDTUpdate\_EC

*Updates the state of the DT controller and returns the step sizes.*

---

### Syntax

```
IppStatus ippsSubbandControllerDTUpdate_EC_16s(const Ipp32sc**
ppSrcRinSubbandsHistory, const Ipp32sc* pSrcSinSubbands, const Ipp32sc*
pSrcFilterErr, Ipp32s_EC_Sfs* pDstStepSize, int* pIsDT, int* pDisabledNLP,
int startSubband, IppsSubbandControllerDTState_EC_16s* pState);
```

### Parameters

`pState`                      Pointer to state structure of the subband DT controller.

<i>ppSrcRinSubbandsHistory</i>	Pointer to an array of pointers to the most recent receive-in blocks. Size of the array is equal to <i>numSegments</i> specified in the function <code>ippsSubbandControllerDTInit_EC</code> .
<i>pSrcSinSubbands</i>	Pointer to subband representation of send-in signal frame (or <code>NULL</code> ). Size of the array is equal to <i>numSubbands</i> specified in the function <code>ippsSubbandControllerDTInit_EC</code> .
<i>pSrcFilterErr</i>	Pointer to the input filter error vector. Vector length is equal to <i>numSubbands</i> specified in <code>ippsSubbandControllerDTInit_EC</code> .
<i>pDstStepSize</i>	Pointer to the vector of step sizes. Vector length is equal to <i>numSubbands</i> specified in <code>ippsSubbandControllerDTInit_EC</code> .
<i>isDT</i>	Pointer to the indicator that indicates the double-talk condition.
<i>disabledNLP</i>	Pointer to the indicator that indicates whether NLP is disable. If it equals 1 - NLP is disabled, if it equals 0 - NLP is not disabled.
<i>startSubband</i>	Indicates the number of subband the filtering starts from ( $0 \leq \text{startSubband} < \text{numSubbands}$ ).

## Description

The functions `ippsSubbandControllerDTUpdate_EC` are declared in the `ippsc.h` file.

This function updates the internal state of the subband DT controller and return the step size values for further use in the coefficients' adaptation process.

The subband DT controller state structure *pState* must be initialized by the function `ippsSubbandControllerDTInit_EC` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>startSubband</i> is less than 0 or greater than or equal to the <i>numSubbands</i> .

## ToneDetectGetStateSize\_EC

*Calculates size of the tone detector state structure.*

---

### Syntax

```
IppStatus ippstToneDetectGetStateSize_EC_16s(ippECFrequency sampleFreq, int* pSize);
```

```
IppStatus ippstToneDetectGetStateSize_EC_32f(ippECFrequency sampleFreq, int* pSize);
```

### Parameters

<i>sampleFreq</i>	Sample frequency.
<i>pSize</i>	Pointer to the computed buffer size value.

### Description

The functions `ippstToneDetectGetStateSize_EC` are declared in the `ippsc.h` file. These functions calculate the size of memory buffers required for the function `ippstToneDetect_EC` to operate.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when <i>sampleFreq</i> is not a valid element of the enumerated type <code>IppFrequency</code> .

## ToneDetectInit\_EC

*Initializes the tone detector state structure.*

---

### Syntax

```
IppStatus ippstToneDetectInit_EC_16s(IppstToneDetectState_EC_16s* pState, ippECFrequency sampleFreq);
```

```
IppStatus ippstToneDetectInit_EC_32f(IppstToneDetectState_EC_32f* pState, ippECFrequency sampleFreq);
```

## Parameters

<i>pState</i>	Pointer to the tone detector state structure.
<i>sampleFreq</i>	Sample frequency.

## Description

The functions `ippsToneDetectInit_EC` are declared in the `ipps.h` file. These functions initialize the tone detector state structure that is used by the function `ippsToneDetect_EC` in the external buffer. Its size must be calculated by the function `ippsToneDetectGetStateSize_EC` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pState</i> pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>sampleFreq</i> is not a valid element of the enumerated type <code>IppFrequency</code> .

## ToneDetect\_EC

*Detects the signal of 2100 Hz frequency with every 450 ms phase reversal.*

---

## Syntax

```
IppStatus ippsToneDetect_EC_16s(const Ipp16s* pSignal, int len, int* pResult,
IppsToneDetectState_EC_16s* pState);

IppStatus ippsToneDetect_EC_32f(const Ipp32f* pSignal, int len, int* pResult,
IppsToneDetectState_EC_32f* pState);
```

## Parameters

<i>pState</i>	Pointer to the tone detector state structure.
<i>pSignal</i>	Pointer to signal vector.
<i>len</i>	Number of samples in signal vector.
<i>pResult</i>	Pointer to the result value. If the value is not zero, this means that the tone was detected.

## Description

The functions `ippsToneDetect_EC` are declared in the `ippsc.h` file.

These functions apply an infinite impulse response (IIR) band-pass filter centered at 2100 Hz frequency. Functions compare powers of the signal before and after the filter.

High-level signal after the filter with periodical short blips is considered as the tone detected.



---

**NOTE.** Detection can occur when summary length of processed signal vectors is equal to or greater than 900 ms.

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## FullbandControllerGetSize\_EC

*Calculates size of the fullband controller state structure.*

---

### Syntax

```
IppStatus ippsFullbandControllerGetSize_EC_32f(int frameSize, int tapLen,  
ippECFrequency sampleFreq, int* pSize);
```

```
IppStatus ippsFullbandControllerGetSize_EC_16s(int frameSize, Int tapLen,  
ippECFrequency sampleFreq, int* pSize);
```

### Parameters

<i>frameSize</i>	Size of the frame.
<i>tapLen</i>	Number of tap values.
<i>sampleFreq</i>	Sample frequency.
<i>pSize</i>	Pointer to the computed buffer size value.



## Description

The functions `ippsFullbandControllerGetSize_EC` are declared in the `ippsc.h` file. These functions calculate the sizes of memory buffers required for the functions `ippsFullbandControllerUpdate_EC` and `ippsFullbandController_EC` to operate.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>frameSize</i> is less than or equal to 0, or <i>tapLen</i> is less than or equal to 0.

## FullbandControllerInit\_EC

Initializes the fullband controller state structure.

### Syntax

```
IppStatus
ippsFullbandControllerInit_EC_32f(IppsFullbandControllerState_EC_32f* pState,
int frameSize, int tapLen, ippECFrequency sampleFreq);

IppStatus
ippsFullbandControllerInit_EC_16s(IppsFullbandControllerState_EC_16s* pState,
int frameSize, int tapLen, ippECFrequency sampleFreq);
```

### Parameters

<i>pState</i>	Pointer to the memory buffer required for the fullband controller state structure. to be initialized. The size of the memory buffer must be computed by <code>ippsFullbandControllerGetSize_EC</code> function and returned via <i>pSize</i> parameter.
<i>frameSize</i>	Size of the frame.
<i>tapLen</i>	Number of tap values.
<i>sampleFreq</i>	Sample frequency.

## Description

The functions `ippsFullbandControllerInit_EC` are declared in the `ippsc.h` file. These functions initialize in the external buffer the fullband controller state structure used by the functions `ippsFullbandControllerUpdate_EC` and `ippsFullbandController_EC`. The size of the buffer must be calculated by the function `ippsFullbandControllerGetSize_EC` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pState</i> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>frameSize</i> is less than or equal to 0, or <i>tapLen</i> is less than or equal to 0.

## FullbandControllerUpdate\_EC

*Updates the fullband controller state and returns the step sizes.*

---

### Syntax

```
IppStatus ippsFullbandControllerUpdate_EC_32f(const Ipp32f* pSrcRin, const
Ipp32f* pSrcSin, Ipp32f* pDstStepSize, IppsFullbandControllerState_EC_32f*
pState);
```

```
IppStatus ippsFullbandControllerUpdate_EC_16s(const Ipp16s* pSrcRin, const
Ipp16s* pSrcSin, IppAECScaled32s* pDstStepSize,
IppsFullbandControllerState_EC_16s* pState);
```

### Parameters

<i>pState</i>	Pointer to the fullband controller state structure.
<i>pSrcRin</i>	Pointer to the receive-in signal history. The history length is <i>tapLen</i> + <i>frameSize</i> , where <i>tapLen</i> and <i>frameSize</i> are specified in the function <code>ippsFullbandControllerInit_EC</code> .
<i>pSrcSin</i>	Pointer to the send-in signal frame. The frame size is specified in the function <code>ippsFullbandControllerInit_EC</code> .

*pDstStepSize* Pointer to the vector of step sizes. Vector length is equal to *frameSize* specified in the function `ippsFullbandControllerInit_EC`.

### Description

The functions `ippsFullbandControllerUpdate_EC` are declared in the `ippsc.h` file. These functions update the internal state of fullband controller and return the step size values for use in the coefficients' adaptation process.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is NULL.

## FullbandController\_EC

*The main fullband controller function which updates filter coefficients and returns output gain coefficients.*

---

### Syntax

```
IppStatus ippsFullbandController_EC_32f(const Ipp32f* pAdaptiveFilterErr,
const Ipp32f* pFixedFilterErr, Ipp32f* pAdaptiveCoefs, Ipp32f* pFixedCoefs,
Ipp32f* pSGain, IppsFullbandControllerState_EC_32f* pState);
```

```
IppStatus ippsFullbandController_EC_16s(const Ipp16s* pAdaptiveFilterErr,
const Ipp16s* pFixedFilterErr, Ipp16s* pAdaptiveCoefs, Ipp16s* pFixedCoefs,
Ipp32s* pSGain, IppsFullbandControllerState_EC_16s* pState);
```

### Parameters

*pState* Pointer to the fullband controller state structure.  
*pAdaptiveFilterErr* Pointer to the vector of adaptive filter output. Vector length is equal to *frameSize*, specified in the function `ippsFullbandControllerInit_EC`.  
*pFixedFilterErr* Pointer to the vector of fixed filter output. Vector length is equal to *frameSize*, specified in the function `ippsFullbandControllerInit_EC`.

<i>pAdaptiveCoefs</i>	Pointer to the adaptive filter coefficients vectors. Vector length is equal to <i>tapLen</i> , specified in the function <code>ippsFullbandControllerInit_EC</code> .
<i>pFixedCoefs</i>	Pointer to the fixed filter coefficients vectors. Vector length is equal to <i>tapLen</i> , specified in the function <code>ippsFullbandControllerInit_EC</code> .
<i>pSGain</i>	Pointer to the send gain coefficient.

## Description

The functions `ippsFullbandController_EC` are declared in the `ippsc.h` file.

These functions compare powers of adaptive filter error and fixed filter error and track changes of adaptive filter coefficients power.

If the adaptive filter has a significantly smaller error and is stable (which is indicated by coefficients power changing slowly), its coefficients are copied to the fixed filter. This case corresponds to “no double-talk” mode.

If the fixed filter has a significantly smaller error, its coefficients are copied to the adaptive filter. This case corresponds to the double-talk mode.

Send gain coefficient calculation (non-linear processor technology) is based on presence of the double-talk mode and changes of receive-in and send-in signal powers.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## FullbandControllerReset\_EC

*Resets fullband controller state.*

---

### Syntax

```
IppStatus
ippsFullbandControllerReset_EC_32f(IppsFullbandControllerState_EC_32f*
pState);

IppStatus
ippsFullbandControllerReset_EC_16s(IppsFullbandControllerState_EC_16s*
pState);
```

## Parameters

*pState* Pointer to the fullband controller state structure.

## Description

The functions `ippsFullbandControllerReset_EC` are declared in the `ippsc.h` file. These functions reset the fullband controller state. You can use these functions after an interruption in adaptation process occurred.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the *pState* pointer is NULL.

## FIR\_EC

*Computes FIR filter results.*

---

## Syntax

```
IppStatus ippsFIR_EC_16s(const Ipp16s* pSrcSpchRef, const Ipp16s* pSrcSpch,
Ipp16s* pDstSpch, int len, Ipp16s* pSrcTaps, Int tapsLen);
```

```
IppStatus ippsFIR_EC_32f(const Ipp32f* pSrcSpchRef, const Ipp32f* pSrcSpch,
Ipp32f* pDstSpch, int len, Ipp32f* pSrcTaps, Int tapsLen);
```

## Parameters

*pSrcSpchRef* Pointer to the source original receive-out signal ( $R_{out}$ ) of length ( $tapsLen + len$ ) .  
*pSrcSpch* Pointer to the send-in signal ( $S_{in}$ ) with echo path.  
*pDstSpch* Pointer to the destination send-out signal ( $S_{out}$ ) which is echo-free.  
*len* Length of source and destination signals.  
*pSrcTaps* Vector of FIR filter taps .  
*tapsLen* Number of taps in the FIR filter.

## Description

The functions `ippsFIR_EC` are declared in the `ippsc.h` file.

These function perform finite impulse response (FIR) filtering.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## FIRSubband\_EC, FIRSubbandLow\_EC

*Computes the frequency-domain adaptive filter output*

---

### Syntax

```

IppStatus ippsFIRSubband_EC_32fc(Ipp32fc** ppSrcSignalIn, Ipp32fc**
ppSrcCoefs, Ipp32fc* pDstSignalOut, int numSegments, int len);

IppStatus ippsFIRSubband_EC_32sc_Sfs(Ipp32sc** ppSrcSignalIn, Ipp32sc**
ppSrcCoefs, Ipp32sc* pDstSignalOut, int numSegments, int len, int
scaleFactor);

IppStatus ippsFIRSubbandLow_EC_32sc_Sfs(const Ipp32sc** ppSrcSignal, const
Ipp32sc** ppCoefs, int numSegments, Ipp32sc* pDstSignal, int startSubband,
int numSubbands, int scaleFactor);

```

### Parameters

<code>ppSrcSignalIn,</code> <code>ppSrcSignal</code>	Pointer to the two-dimensional vector of size <code>[numSegments]*[len]</code> containing the pointers to the most recent complex-valued FFT spectra in input audio signal.
<code>ppSrcCoefs, ppCoefs</code>	Pointer to the two-dimensional vector of size <code>[numSegments]*[len]</code> containing the pointers to the filter coefficients vector of size <code>[len]</code> .
<code>pDstSignalOut,</code> <code>pDstSignal</code>	Pointer to the complex-valued filter output vector.
<code>numSegments</code>	Number of filter segments ( $0 < \text{numSegments} < 256$ ).
<code>len, numSubbands</code>	Number of filter subbands ( $0 < \text{len}, \text{numSubbands} < 4097$ ).

<i>startSubband</i>	Number of subbands to skip before filtering ( $0 \leq \text{startSubband} < \text{numSubbands}$ ).
<i>scaleFactor</i>	Saturation fixed scale factor ( $-32 < \text{scaleFactor} < 32$ ), for <code>ippsFIRSubbandLow_EC_32sc_Sfs</code> function ( $0 \leq \text{scaleFactor} < 32$ ).

### Description

The functions `ippsFIRSubband_EC` and `ippsFIRSubbandLow_EC` are declared in the `ippsc.h` file. These functions perform FIR filtering of the input two-dimensional spectra vector.

The `ippsFIRSubbandLow_EC_32sc_Sfs` function performs filtering without 64-bit integer overflow check. For *startSubband*=0, if no overflow occurs this function is equivalent to the function `ippsFIRSubband_EC_32sc_Sfs` with *len* equals to *numSubbands*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> has an illegal value.
<code>ippStsRangeErr</code>	Indicates an error when <i>numSegments</i> , <i>startSubband</i> or <i>scaleFactor</i> is out of the specified range.

## FIRSubbandCoeffUpdate\_EC, FIRSubbandLowCoeffUpdate\_EC

*Updates the adaptive filter coefficients*

### Syntax

```
IppStatus ippsFIRSubbandCoeffUpdate_EC_32fc_I(const double* pSrcStepSize,
const Ipp32fc** ppSrcFilterInput, const Ipp32fc* pSrcError, Ipp32fc**
ppSrcDstCoefs, int numSegments, int len);
```

```
IppStatus ippsFIRSubbandCoeffUpdate_EC_32sc_I(const Ipp32s_EC_Sfs*
pSrcStepSize, const Ipp32sc** ppSrcFilterInput, const Ipp32sc* pSrcError,
Ipp32sc** ppSrcDstCoefsQ15, int numSegments, int len, int scaleFactorCoef);
```

```
IppStatus ippsFIRSubbandLowCoeffUpdate_EC_32sc_I(const Ipp32sc**
ppSrcFilterInput, const Ipp32sc* pSrcError, Ipp32sc** ppSrcDstCoefsQ15, int
numSegments, Ipp32sc* pDstProdStepErrQ, const Ipp32s_EC_Sfs*
pSrcAdaptStepSize, int startSubband, int numSubbands, int scaleFactorCoef);
```

## Parameters

<i>pSrcStepSize,</i> <i>pSrcAdaptStepSize</i>	Pointer to the adaptive filter step size vector of size <i>len</i> or <i>numSubbands</i> . For the integer functions step size vector elements are represented as a scaled integer values where variable <i>x</i> of type <code>Ipp32s_EC_Sfs</code> corresponds to the floating point value $x.val * 2^{x.sf}$ .
<i>ppSrcFilterInput</i>	Pointer to the array of pointers to the most recent input blocks (for example, $X_n, X_{n-1}, \dots, X_{n-L+1}$ ). These are the complex-valued vectors that contain the FFT of the input signal. The dimension of <i>ppSrcFilterInput</i> is $[numSegments] * [len]$ or $[numSegments] * [numSubbands]$ .
<i>pSrcError</i>	Pointer to the complex-valued vector containing the filter error. Its dimension is $[len]$ or $[numSubbands]$ .
<i>ppSrcDstCoefs,</i> <i>ppSrcDstCoefsQ15</i>	Pointer to the array of pointers to the filter coefficient vectors. They are the complex-valued vectors containing the filter coefficients. The dimension of <i>ppSrcDstCoefs</i> is $[numSegments] * [len]$ or $[numSegments] * [numSubbands]$ .
<i>startSubband</i>	Number of subbands to skip before filtering ( $0 \leq startSubband < numSubbands$ ).
<i>numSegments</i>	Number of filter segments ( <i>L</i> ) ( $0 < numSegments < 256$ ).
<i>len, numSubbands</i>	Number of adaptive filter subbands and length of the input and output vectors.
<i>scaleFactorCoef</i>	Fixed scale factor for filter coefficients ( $0 < scaleFactorCoef < 32$ ).
<i>pDstProdStepErrQ</i>	Pointer to the output vector of filter error and step size product.

## Description

The functions `ippsFIRSubbandCoeffUpdate_EC` and `ippsFIRSubbandLowCoeffUpdate_EC` are declared in the `ippsc.h` file. These functions update the adaptive filter coefficients according to the given step size and filter error independently in each frequency subband.



The function `ippsFIRSubbandLowCoeffUpdate_EC_32sc_I` computes the product between step size, error and coefficients without 64-bit integer overflow check. For *startSubband*=0, if no overflow occurs this function is equivalent to `ippsFIRSubbandCoeffUpdate_EC_32sc_I` with *len* equal to *numSubbands*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error when <i>len</i> has an illegal value.
<code>ippStsRangeErr</code>	Indicates an error when <i>pSrcStepSize</i> [ <i>i</i> ] < 0, or when <i>numSegments</i> or <i>scaleFactorCoef</i> is out of the specified range.

## NLMS\_EC

Performs FIR filtering with coefficients update.

### Syntax

```
ippStatus ippsNLMS_EC_16s(const Ipp16s* pSrcSpchRef, const Ipp16s* pSrcSpch,
const Ipp32s* pStepSize, Ipp16s* pSrcDstErr, Ipp16s* pDstSpch, int len,
Ipp16s* pSrcDstTaps, int tapsLen);
```

```
ippStatus ippsNLMS_EC_32f(const Ipp32f* pSrcSpchRef, const Ipp32f* pSrcSpch,
const Ipp32f* pStepSize, Ipp32f* pSrcDstErr, Ipp32f* pDstSpch, int len,
Ipp32f* pSrcDstTaps, int tapsLen);
```

### Parameters

<i>pSrcSpchRef</i>	Pointer to the source original receive-out signal ( $R_{out}$ ) of length ( <i>tapsLen</i> + <i>len</i> ) .
<i>pSrcSpch</i>	Pointer to the send-in signal ( $S_{in}$ ) with echo path.
<i>pStepSize</i>	Pointer to the vector of step sizes. Vector length is equal to <i>len</i> .
<i>pSrcDstErr</i>	Pointer to the last error value.
<i>pDstSpch</i>	Pointer to the destination send-out signal ( $S_{out}$ ) which is echo-free.
<i>len</i>	Length of source and destination signals.
<i>pSrcDstTaps</i>	Source and destination vector of FIR filter taps.

*tapsLen*                      Number of taps.

### Description

The functions `ippsNLMS_EC` are declared in the `ippsc.h` file.

These function perform FIR filtering with normalized Least Mean Square (LMS) coefficients adaptation.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`               Indicates an error when one of the specified pointers is `NULL`.

## Noise Reduction Functions

The functions described here implement building blocks that can be used to create an background noise canceller or other noise processing components, such as Noise Pre-Processing compliant to the Microsoft RT Audio codec.

## FilterNoiseGetStateSize

*Calculates the size of the state structure for the noise reduction filter*

---

### Syntax

```

IppStatus ippsFilterNoiseGetStateSize_EC_32f(IppPCMFrequency pcmFreq, int*
pSize);

IppStatus ippsFilterNoiseGetStateSize_RTA_32f(IppPCMFrequency pcmFreq, int*
pSize);

```

### Parameters

*pcmFreq*                      Sampling frequency, possible values see below.  
*pSize*                          Pointer to the calculated value of the state structure size.

## Description

The functions `ippsFilterNoiseGetStateSize` are declared in the `ippsc.h` file. These functions calculate the size of the state structure for noise reduction filtering and store the result in `pSize`. The sampling frequency `pcmFreq` must be specified, the supported values are as follows:

`IPP_PCM_FREQ_8000`, `IPP_PCM_FREQ_16000` for the EC noise filtering;

`IPP_PCM_FREQ_8000`, `IPP_PCM_FREQ_16000`, `IPP_PCM_FREQ_22050`, `IPP_PCM_FREQ_32000` for the RTA noise filtering.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSize</code> pointer is <code>NULL</code> .
<code>IppStsRangeErr</code>	Indicates an error when the <code>pcmFreq</code> has an illegal value.

## FilterNoiseInit

*Initializes the state structure for the noise reduction filter*

---

### Syntax

```
IppStatus ippsFilterNoiseInit_EC_32f(IppPCMFrequency pcmFreq,
IppsFilterNoiseState_EC_32f* pNRStateMem);

IppStatus ippsFilterNoiseInit_RTA_32f(IppPCMFrequency pcmFreq,
IppsFilterNoiseState_RTA_32f* pNRStateMem);
```

### Parameters

<code>pcmFreq</code>	Sampling frequency, the following values are supported: <code>IPP_PCM_FREQ_8000</code> , <code>IPP_PCM_FREQ_16000</code> , <code>IPP_PCM_FREQ_22050</code> or <code>IPP_PCM_FREQ_32000</code> .
<code>pNRStateMem</code>	Pointer to the state structure for noise reduction filter.

### Description

The functions `ippsFilterNoiseInit` are declared in the `ippsc.h` file. These functions initialize the noise reduction filter state structure `pNRStateMem`. Before calling these functions the memory must be allocated for the state structure. Its size must be calculated by the functions [ippsFilterNoiseGetStateSize](#).

When the structure is initialized the noise residual level is set to the default values: `ippsNrMedium` for EC, and `ippsNrNormal` for RTA. This level can be changed using the function `ippsFilterNoiseLevel`.

### Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error when the <code>pNRStateMem</code> pointer is <code>NULL</code> .
<code>ippsStsRangeErr</code>	Indicates an error when <code>pcmFreq</code> has an illegal value.

## FilterNoiseLevel

*Sets the level of the noise attenuation.*

---

### Syntax

```
IppStatus ippsFilterNoiseLevel_EC_32f( IppsNRLevel level,  
IppsFilterNoiseState_EC_32f* pNRStateMem);  
  
IppStatus ippsFilterNoiseLevel_RTA_32f(IppsNRLevel level,  
IppsFilterNoiseState_RTA_32f* pNRStateMem);
```

### Parameters

<code>level</code>	Degree of noise attenuation; possible values are: <code>ippsNrLow</code> , <code>ippsNrMedium</code> , <code>ippsNrNormal</code> , <code>ippsNrHigh</code> , <code>ippsNrNone</code> .
<code>pNRStateMem</code>	Pointer to the memory supplied for the filter state.

### Description

The functions `ippsFilterNoiseLevel` are declared in the `ippsc.h` file. These functions set the level of the noise attenuation performed by the filter. The values in dB of the noise attenuation depend on noise filter algorithms and are given in the table below.

#### Levels of Noise Attenuation (in dB)

Value of the Parameter <code>level</code>	EC Noise Filter	RTA Noise Filter
<code>ippsNrLow</code>	8	6
<code>ippsNrMedium</code>	10	10
<code>ippsNrNormal</code>	16	22
<code>ippsNrHigh</code>	39	not supported

---

Value of the Parameter	<i>level</i>	EC Noise Filter	RTA Noise Filter
<code>ippsNrNone</code>		0	0

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pNRStateMem</i> pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>level</i> has an illegal value.

## FilterNoiseDetect\_EC

*Performs noise detection for EC noise filtering.*

---

### Syntax

```
IpplStatus ippsFilterNoiseDetect_EC_32f64f(const Ipp32f pSrc[16], Ipp64f*  
pNoisePower, Ipp32f* pMean, int* pNoiseFlag, IpplFilterNoiseState_EC_32f*  
pNRStateMem);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector.
<i>pNoisePower</i>	Pointer to the output power of noise pattern.
<i>pMean</i>	Pointer to the weighted mean value of noise.
<i>pNoiseFlag</i>	Pointer to the flag indicated the noise presence: 1 - noise is detected, 0 - no noise is detected.
<i>pNRStateMem</i>	Pointer to the filter state structure.

### Description

The function `ippsFilterNoiseDetect_EC` is declared in the `ippsc.h` file.

The function performs detection of noise only in cases voice versus voice, and voice mixed with noise. The detection result is returned by *pNoiseFlag*. The value of the average noise power is stored in the *pNoisePower*. The weighted mean value of noise *pMean* can be considered as the DC offset. The filter state *pNRStateMem* must be initialized beforehand by the function [ippsFilterNoiseInit\\_EC](#). If the noise is detected, then the filter state memory is updated with the signal history, noise statistics, and noise pattern.

Code [Examples 9-1](#) and [9-2](#) show how the function `ippsFilterNoiseDetect_EC` can be used.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## FilterNoiseDetectModerate\_EC

*Performs noise detection for EC noise filtering.*

---

### Syntax

```
IppStatus ippsFilterNoiseDetectModerate_EC_32f64f(const Ipp32f* pSrcSin[16],
const Ipp32f* pSrcRin[16], Ipp64f* pNoisePower, Ipp32f* pMean, int*
pNoiseFlag, IppsFilterNoiseState_EC_32f* pNRStateMem);
```

### Parameters

<code>pSrcSin</code>	Pointer to the input vector containing echo.
<code>pSrcRin</code>	Pointer to the vector containing echo source.
<code>pNoisePower</code>	Pointer to the output power of noise pattern.
<code>pMean</code>	Pointer to the weighted mean value of noise.
<code>pNoiseFlag</code>	Pointer to the flag indicated the noise presence: 1 - noise is detected, 0 - no noise is detected.
<code>pNRStateMem</code>	Pointer to the filter state structure.

### Description

The function `ippsFilterNoiseDetectModerate_EC` is declared in the `ippsc.h` file.

The function performs detection of noise similar to the function `ippsFilterNoiseDetect_EC`. The difference is that the noise detection performed on the signals containing echo provides higher noise mitigation by the following processing with the function `ippsFilterNoise_EC`.

Using the function `ippsFilterNoiseDetectModerate_EC` in the echo canceller operations gives no misdetects during the normal speech and provides more stable NLP block performance on the signals with echo.

The detection result is returned by `pNoiseFlag`. The value of the average noise power is stored in the `pNoisePower`, it can be used for dynamic noise level control by the function `ippsFilterNoiseLevel_EC`. The weighted mean value of noise `pMean` can be considered as the DC

offset. The filter state *pNRStateMem* must be initialized beforehand by the function [ippsFilterNoiseInit\\_EC](#). If the noise is detected, then the filter state memory is updated with the signal history, noise statistics, and noise pattern.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointer is <code>NULL</code> .

## FilterNoiseSetMode\_EC

*Sets type of the smoothing filter.*

---

### Syntax

```
IppStatus ippsFilterNoiseSetMode_EC_32f(IppsNrSmoothMode mode,
IppsFilterNoiseState_EC_32f* pNRStateMem);
```

### Parameters

<i>mode</i>	Specifies the smoothing mode; possible values are: <code>ippsNrSmoothDynamic</code> , <code>ippsNrSmoothStatic</code> , <code>ippsNrSmoothOff</code> .
<i>pNRStateMem</i>	Pointer to the filter state structure.

### Description

The function `ippsFilterNoiseSetMode_EC` is declared in the `ippsc.h` file.

The function sets the type of smoothing filter that is applied by the function `ippsFilterNoise_EC` [ippsFilterNoise\\_EC](#) to denoise a signal. A smoothing filter is chosen from the predefined filter bank depending on the *mode* value.

`ippsNrSmoothDynamic` - smoothing filter is chosen dynamically depending on the input SNR. The bigger SNR the wider window is used for smoothing.

`ippsNrSmoothStatic` - default smoothing filter is applied.

`ippsNrSmoothOff` - denoising without smoothing, preferable for signal with low level of noise.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pNRStateMem</code> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <code>mode</code> has an illegal value.

## FilterNoise

*Performs noise reduction filtering*

---

### Syntax

```

IppStatus ippFilterNoise_RTA_32f_I(Ipp32f pSrcDst[160],
IppsFilterNoiseState_RTA_32f* pNRStateMem);

IppStatus ippFilterNoise_RTA_32f(const Ipp32f pSrc[160], Ipp32f pDst[160],
IppsFilterNoiseState_RTA_32f* pNRStateMem);

IppStatus ippFilterNoise_EC_32f_I(const Ipp32f pSrcDst[16], IppsNrMode
filterMode, IppsFilterNoiseState_EC_32f* pNRStateMem);

IppStatus ippFilterNoise_EC_32f(const Ipp32f pSrc[16], Ipp32f pDst[16],
IppsNrMode filterMode, IppsFilterNoiseState_EC_32f* pNRStateMem);

```

### Parameters

<code>pSrc</code>	Pointer to the input vector.
<code>pDst</code>	Pointer to the output vector.
<code>pSrcDst</code>	Pointer to the input and output vector
<code>filterMode</code>	Specifies how the filter state structure is updated: <code>ippsNrNon</code> - no update, <code>ippsNrUpdate</code> - signal statistics update, <code>ippsNrUpdateAll</code> - signal and noise statistics update.
<code>pNRStateMem</code>	Pointer to the filter state structure.

### Description

The functions `ippFilterNoise` are declared in the `ipps.h` file. These functions perform noise reduction filtering of the noisy speech signal. The filter state structure must be initialized beforehand using the respective `ippFilterNoiseInit` function. The functions `ippFilterNoise` use the predefined levels of the noise attenuation. Their default value set during the state structure's initialization can be changed using the function `ippFilterNoiseLevel`. The parameter `filterMode` specified how the filter state structure is updated.

Code [Examples 9-1](#) and [9-2](#) show how the function `ippFilterNoise` can be used.





---

**NOTE.** The higher noise attenuation level is set, the more undesirable distortions of the speech signal can occur.

---



---

**NOTE.** When the speech signal has low signal-to-noise ratio (SNR), some degradation may be observed after the noise reduction.

---

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>IppStsRangeErr</code>	Indicates an error when <i>filterMode</i> has an invalid value.

### Code Examples

The following code examples show how the noise reduction functions can be used for EC noise filtering.

Code Example 9-1 below demonstrates how to reduce noise in the signal that is distorted version of the original noisy signal:

#### Example 9-1 Noise Reduction in the Distorted Signal

```
void ReduceNoiseExample1(Ipp32f
    *pSig, Ipp32f *pSigDistorted,  IppsFilterNoiseState_EC_32f *state1,
    IppsFilterNoiseState_EC_32f *state2)
{
    Ipp64f fooPwr;

    Ipp32f fooMean;

    int    flag;
```

```

    IppsNrMode nMode;

    /* Detect noise and update signal and noise statistic and noise pattern */
    ippsNoiseDetect_EC_32f64f(pSig,&fooPwr,&fooMean,&flag,statel);

    /* Filter noise out from distorted signal using noise pattern aligned with original
    signal noise statistic.

    In noisy case update all statistics and noise pattern, otherwise update only statistic.
    */
    nMode=sinNoiseFlag?ippsNrUpdateAll:ippsNrUpdate;
    ippsFilterNoise_EC_32f_I(pSigDistorted,nMode,state2);
}

```

The following code Example 9-2 demonstrates filtering of the original noisy signal.

### Example 9-2 Noise Reduction in the Noisy Signal

```

void ReduceNoiseExample2(Ipp32f* pSig, IppsFilterNoiseState_EC_32f *state)
{
    Ipp64f fooPwr;

    Ipp32f fooMean;

    int    fooFlag;

    /* Detect noise and update signal and noise statistic and noise pattern */
    ippsNoiseDetect_EC_32f64f(pSig,&fooPwr,&fooMean,&fooFlag,state);

    /* Filter noise out using existent noise pattern */
    ippsFilterNoise_EC_32f_I(pSig, ippsNrNoUpdate,state);
}

```

## Automatic Audio Level Control Functions

This section describes the Intel IPP functions for automatic control of audio level.

## ALCGetStateSize\_G169

*Returns the size of the ALC state structure.*

---

### Syntax

```
IppStatus ippALCGetStateSize_G169_16s(int* pSize);
```

### Parameters

*pSize* Pointer to the calculated value of the memory size.

### Description

The function *ippALCGetStateSize\_G169* is declared in the *ippsc.h* file. The functions calculate the size of the automatic level control (ALC) state structure and store the result in *pSize*.

### Return Values

*ippStsNoErr* Indicates no error.  
*ippStsNullPtrErr* Indicates an error when the *pSize* pointer is NULL.

## ALCInit\_G169

*Initializes ALC state structure.*

---

### Syntax

```
IppStatus ippALCInit_G169_16s(IppsALCState_G169_16s* pALCMem);
```

### Parameters

*pALCMem* Pointer to the ALC state structure.

### Description

The function *ippALCInit\_G169s* is declared in the *ippsc.h* file. The function initializes the ALC state structure *pALCMem* in the buffer of size returned by the function [ippALCGetStateSize\\_G169](#). The speech clipping and target levels are set to default values: 0 dBm0 - clipping level, -26.1 dBm0 - target level, and 11.6 dB - maximum gain level. These values can be changed by using the functions [ippALCSetLevel\\_G169](#) and [ippALCSetGain\\_G169](#) respectively.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pALCMem</code> pointer is <code>NULL</code> .

## ALCSetLevel\_G169

*Sets ALC algorithm target and clipping levels.*

---

### Syntax

```
IppStatus ippALCSetLevel_G169_16s(Ipp32f targetLevel, Ipp32f clipLevel,
IppsALCState_G169_16s* pALCMem);
```

### Parameters

<code>targetLevel</code>	Automatic level control target level in dBm0 after dropping the sign.
<code>clipLevel</code>	Automatic level control clipping level in dBm0 after dropping the sign.
<code>pALCMem</code>	Pointer to the ALC state structure.

### Description

The functions `ippALCSetLevel_G169` is declared in the `ippsc.h` file. This function set the target and clipping level for the automatic level control. The ALC state structure must be initialized using the function `ippALCInit_G169` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pALCMem</code> pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when the <code>targetLevel</code> or <code>clipLevel</code> is negative.

## ALCSetGain\_G169

*Sets ALC algorithm maximum gain value.*

---

### Syntax

```
IppStatus ippsALCSetGain_G169_16s(Ipp32f maxGain, IppsALCState_G169_16s*
pALCMem);
```

### Parameters

<i>maxGain</i>	Automatic level control maximum gain limit in dB.
<i>pALCMem</i>	Pointer to the ALC state structure.

### Description

The function `ippsALCMaxGain_G169_16s` is declared in the `ippsc.h` file. This function set the target level for the automatic level control. The ALC state structure must be initialized using the function `ippsALCInit_G169` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pALCMem</i> pointer is NULL.
<code>ippStsRangeErr</code>	Indicates an error when the <i>maxGain</i> is negative.

## ALC\_G169

*Performs the automatic level control.*

---

### Syntax

```
IppStatus ippsALC_G169_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
IppsALCState_G169_16s* pALCMem);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector.
<i>pDst</i>	Pointer to the output vector.
<i>len</i>	Length of the vectors.
<i>pALCMem</i>	Pointer to the ALC state structure.

## Description

The function `ippALC_G169` is declared in the `ippsc.h` file. The function perform noise automatic level control using predefined clipping limit and target level which can be changed by the functions `ippALCSetLevel_G169` and `ippALCSetGain_G169` respectively. The ALC state structure must be initialized using the function `ippALCInit_G169` beforehand. The length of input and output vectors must be multiple of 8.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <code>len</code> is less or equal 0, or is not multiple of 8.

# G722 Sub-Band ADPCM Speech Codec Functions

The Intel IPP functions described later in this section if properly combined may be used to construct the G.722 speech codec compliant to the reference implementation given by ITU-T library (Rec.192, Clause 7: *The ITU-T 64, 56 and 48 kbps wide-band speech coding algorithm*). The description of G.722 speech codec also known as Sub-Band ADPCM compression may be found in ITU-T G.722 Recommendation: *7KHz coding within 64 kbps*.

The Intel IPP functions described below are primarily concerned with the well-defined, computationally expensive core operations that comprise the G.722 subband codec.

The list of Intel IPP Subband ADPCM functions is given in Table 9-17.

**Table 9-17 Intel IPP Subband ADPCM Functions**

Function Base Name	Operation
<code>SBADPCMEncodeStateSize_G722</code>	Calculates the memory size required for G.722 Sub-Band ADPCM encoder.
<code>SBADPCMEncodeInit_G722</code>	Initializes the memory buffer for the Sub-Band ADPCM encoder.
<code>SBADPCMEncode_G722</code>	Performs Sub-Band ADPCM encoding of the synthesis QMF samples.
<code>QMFEncode_G722</code>	Performs QMF analysis of the uniform PCM input.
<code>SBADPCMDecodeStateSize_G722</code>	Calculates the memory size required for Sub-Band ADPCM decoder.
<code>SBADPCMDecodeInit_G722</code>	Initializes the memory buffer for the Sub-Band ADPCM decoder.
<code>SBADPCMDecode_G722</code>	Performs decoding of the Sub-Band ADPCM compressed bit-stream.

Function Base Name	Operation
QMFDecode_G722	Performs QMF synthesis of recovered samples.

## SBADPCMEncodeStateSize\_G722

*Calculates the memory size required for G.722 Sub-Band ADPCM encoder.*

### Syntax

```
IppStatus ippSBADPCMEncodeStateSize_G722_16s (int* pEncMemSize);
```

### Parameters

*pEncMemSize*                      Pointer to the output memory size in bytes.

### Description

The function `ippSBADPCMEncodeStateSize_G722` is declared in the `ippsc.h` file.

This function gets the information about the amount of memory (in bytes) needed to process the G.722 Sub-Band ADPCM compression.

### Return Values

`ippStsNoErr`                      Indicates no error.

`IppStsNullPtrErr`               Indicates an error when the *pEncMemSize* pointer is `NULL`.

## SBADPCMEncodeInit\_G722

*Initializes the memory buffer for the Sub-Band ADPCM encoder.*

### Syntax

```
IppStatus ippSBADPCMEncodeInit_G722_16s (IppsEncoderState_G722_16s* pEncMem);
```

### Parameters

*pEncMem*                              Pointer to the input memory buffer of size required to properly initialize the G.722 Sub-Band ADPCM encoder.

## Description

The function `ippsSBADPCMEncodeInit_G722` is declared in the `ippsc.h` file.

This function initializes the memory buffer referenced by the pointer `pEncMem`. This buffer is required to enable G.722 Sub-Band ADPCM compression (encoding) starting from the reset state.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <code>pEncMem</code> pointer is <code>NULL</code> .

## SBADPCMEncode\_G722

*Performs Sub-Band ADPCM encoding of the synthesis QMF samples.*

---

## Syntax

```
IppStatus ippsSBADPCMEncode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int
len, IppsEncoderState_G722_16s* pEncMem);
```

## Parameters

<code>pSrc</code>	Pointer to the input vector of synthesis QMF samples.
<code>pDst</code>	Pointer to the ADPCM bit-stream output vector.
<code>len</code>	The length of input vector, must be a multiple of two.
<code>pEncMem</code>	Pointer to the state memory of the Sub-Band ADPCM encoder.

## Description

The function `ippsSBADPCMEncode_G722_16s` is declared in the `ippsc.h` file.

This function performs the Sub-Band ADPCM compression (encoding) of the synthesized Quadrature Mirror Filter (QMF) samples within a bit rate of 64 kbps. The function uses the state memory referenced by `pEncMem` to encode the input data and update it accordingly. Before using this function, you must initialize the state memory by calling the [ippsSBADPCMEncodeInit\\_G722](#) function. The function `ippsSBADPCMEncode_G722` splits the frequency band of the input signal into two sub-bands (higher and lower) and encodes the signals in each



sub-band using ADPCM compression. Each element of the output vector contains six binary digits of the lower sub-band and two binary digits of the higher sub-band for the 64 kbps bit rate ADPCM encoding.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not a multiple of two.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## QMFEncode\_G722

*Performs QMF analysis of the uniform PCM input.*

### Syntax

```
ippStatus ippSQMFEncode_G722_16s(const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s* delay);
```

### Parameters

<i>pSrc</i>	Pointer to the uniform PCM input speech vector.
<i>pDst</i>	Pointer to the synthesis QMF samples output vector.
<i>len</i>	The length of input/output vectors, must be a multiple of two.
<i>delay</i>	Pointer to the buffer of a delay line.

### Description

The function `ippSQMFEncode_G722_16s` is declared in the `ippsc.h` file.

This function uses Quadrature Mirror Filtering (QMF) of the 14-bit uniform PCM speech input to compute the lower and higher sub-band signal components for the Sub-Band ADPCM encoder. Computations are done in accordance with ITU-T Recommendation G.722.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not a multiple of two.

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.

## SBADPCMDecodeStateSize\_G722

*Calculates the memory size required for Sub-Band ADPCM decoder.*

---

### Syntax

```
IppStatus ippSBADPCMDecodeStateSize_G722_16s (int* pDecMemSize);
```

### Parameters

*pDecMemSize* Pointer to the output size of the memory needs to perform G.722 SB-ADPCM decoder.

### Description

The function `ippSBADPCMDecodeStateSize_G722` is declared in the `ippsc.h` file.

This function gets the information about the amount of memory (in bytes) needed to process the G.722 Sub-Band ADPCM decompression.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the *pDecMemSize* pointer is `NULL`.

## SBADPCMDecodeInit\_G722

*Initializes the memory buffer for the Sub-Band ADPCM decoder.*

---

### Syntax

```
IppStatus ippSBADPCMDecodeInit_G722_16s (IppsDecoderState_G722_16s* pDecMem);
```

### Parameters

*pDecMem* Pointer to the input memory buffer of size required to properly initialize the G.722 Sub-Band ADPCM decoder.

## Description

The function `ippsSBADPCMDecodeInit_G722_16s` is declared in the `ippsc.h` file.

This function initializes the memory buffer referenced by the pointer `pDecMem`. This buffer is required to enable G.722 Sub-Band ADPCM decompression (decoding) starting from the reset state.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when the <code>pDecMem</code> pointer is <code>NULL</code> .

## SBADPCMDecode\_G722

*Performs decoding of the Sub-Band ADPCM compressed bit-stream.*

---

## Syntax

```
IppStatus ippsSBADPCMDecode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int
len, Ipp16s mode, IppsDecoderState_G722_16s* pDecMem);
```

## Parameters

<code>pSrc</code>	Pointer to the input vector that contains the even number of samples. The even element of this vector contains six binary digits of the lower sub-band and the odd element of this vector contains two binary digits of the higher sub-band.
<code>pDst</code>	Pointer to the output vector that contains decoded low-band and high-band portions of the recovered samples.
<code>len</code>	Length of the input vector.
<code>mode</code>	Decode bit rate of the G.722 decoder. The values of <code>mode</code> have the following meaning: <code>mode</code> = 1 - sets 64 kbps bit rate; <code>mode</code> = 2 - sets 56 kbps bit rate; <code>mode</code> = 3 - sets 48 kbps bit rate.

*pDecMem*

Pointer to the state memory of the Sub-Band ADPCM decoder.

## Description

The function `ippsSBADPCMDecode_G722_16s` is declared in the `ippsc.h` file.

This function decodes the lower-band and upper-band portions of the Sub-Band ADPCM compressed bit-stream into the low-band and upper-band portions of the recovered samples, respectively. This operation is the reverse of the Sub-Band ADPCM compression (see [ippsSBADPCMEncode\\_G722](#)). Note that the effective audio coding bit rate varies between 64, 56, or 48 kbps depending on the mode of operation.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## QMFDecode\_G722

*Performs QMF synthesis of recovered samples.*

---

### Syntax

```
IppStatus ippsQMFDecode_G722_16s (const Ipp16s* pSrc, Ipp16s* pDst, int len,
Ipp16s* pDelay);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector that contains decoded low-band and high-band portions of the recovered samples.
<i>pDst</i>	Pointer to the 16-bit linear PCM speech output vector.
<i>len</i>	The length of input/output vectors, must be a multiple of two.
<i>pDelay</i>	Pointer to the buffer of a delay line.

### Description

The function `ippsQMFDecode_G722_16s` is declared in the `ippsc.h` file.

This function performs Quadrature Mirror Filter (QMF) synthesis of the low-band and high-band portions of the recovered samples to reconstruct the output signal. Computations are done in accordance with ITU-T Recommendation G.722.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsBadArgErr</code>	Indicates an error when <i>len</i> is than less or equal to 0.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not a multiple of two.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

Companding Functions

The functions described in this section perform an operation of data compression by using a logarithmic encoder-decoder, referred to as *companding*. Companding allows to maintain a constant percentage error by logarithmically spacing the quantization levels [Rab78].

The list of the Intel IPP companding functions is given in Table 9-18.

Table 9-18. Companding Functions

Function Base Name	Operation
<code>MuLawToLin</code>	Decodes samples from 8-bit $\mu$ -law encoded format to linear samples
<code>LinToMuLaw</code>	Encodes the linear samples using 8-bit $\mu$ -law format and stores them in a vector
<code>ALawToLin</code>	Decodes the 8-bit A-law encoded samples to linear samples.
<code>LinToALaw</code>	Encodes the linear samples using 8-bit A-law format and stores them in an array
<code>MuLawToALaw</code>	Converts samples from 8-bit $\mu$ -law encoded format to 8-bit A-law encoded format
<code>ALawToMuLaw</code>	Converts samples from 8-bit A-law encoded format to 8-bit $\mu$ -law encoded format.

The Intel IPP companding functions perform the following conversion operations of signal samples:

- From 8-bit  $\mu$ -law encoded format to PCM-linear or vice-versa
- From 8-bit A-law encoded format to PCM-linear or vice-versa
- From 8-bit  $\mu$ -law encoded format to A-law encoded or vice-versa.

Samples encoded in  $\mu$ -law or A-law format are non-uniformly quantized. The quantization functions used by these formats are designed to reduce the dependency of signal-to-noise ratio on the magnitude of the encoded signal. This is achieved by quantization (companding) at a finer resolution near zero, and at a coarse resolution at larger positive or negative levels. The output values are normalized to be in the range [-1; +1].

These functions perform the  $\mu$ -law and A-law companding in compliance with the CCITT G.711 specification. For the conversion rules and more details, refer to [CCITT].

Code [example 9-3](#) shows how to use companding functions.

## MuLawToLin

*Decodes samples from 8-bit  $\gamma$ -law encoded format to linear samples.*

---

### Syntax

```
ippStatus ippMuLawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
ippStatus ippMuLawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of samples in the vector.

### Description

The function `ippMuLawToLin` is declared in the `ipps.h` file. This function decodes the 8-bit  $\mu$ -law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## LinToMuLaw

*Encodes the linear samples using 8-bit  $\gamma$ -law format and stores them in a vector.*

---

### Syntax

```
IppStatus ippsLinToMuLaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToMuLaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of samples in the vector.

### Description

The function `ippsLinToMuLaw` is declared in the `ipps.h` file. This function encodes the PCM-linear samples in the vector *pSrc* using 8-bit  $\mu$ -law format and stores them in the vector *pDst*.

The formula for  $\mu$ -law companding is as follows:

$$|C_{\mu}(x)| = \frac{\ln(1 + 255 \cdot |x|)}{\ln(256)} \cdot 128, \quad -1 \leq x \leq 1$$

where  $x$  is the linear signal sample and  $C_{\mu}(x)$  is the  $\mu$ -law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of  $\mu$ -law format is usually performed using look-up *Tables 2a/G.711* and *2b/G.711* shown in the CCITT specification *G.711* [CCITT]. Refer to the G.711 specification for details.

Code [example 9-3](#) shows how to use the function `ippsLinToMuLaw_32f8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## ALawToLin

*Decodes the 8-bit A-law encoded samples to linear samples.*

---

### Syntax

```

IppStatus ippALawToLin_8u16s(const Ipp8u* pSrc, Ipp16s* pDst, int len);
IppStatus ippALawToLin_8u32f(const Ipp8u* pSrc, Ipp32f* pDst, int len);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of samples in the vector.

### Description

The function `ippALawToLin` is declared in the `ipps.h` file. This function decodes the 8-bit A-law encoded samples in the vector *pSrc* to PCM-linear samples and stores them in the vector *pDst*. Code [example 9-3](#) shows how to use the function `ippALawToLin_8u32f`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.



## LinToALaw

*Encodes the linear samples using 8-bit A-law format and stores them in an array.*

---

### Syntax

```
IppStatus ippsLinToALaw_16s8u(const Ipp16s* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLinToALaw_32f8u(const Ipp32f* pSrc, Ipp8u* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of samples in the vector.

### Description

The function `ippsLinToALaw` is declared in the `ipps.h` file. This function encodes the PCM-linear samples in the vector *pSrc* using 8-bit A-law format and stores them in the vector *pDst*.

The formula for A-law companding is as follows:

$$|C_A(x)| = \begin{cases} \frac{87.56|x|}{1 + \ln 87.56} \cdot 128, & 0 \leq |x| \leq \frac{1}{87.56} \\ \frac{1 + \ln(87.56|x|)}{1 + \ln 87.56} \cdot 128, & \frac{1}{87.56} < |x| \leq 1 \end{cases},$$

where  $x$  is the linear signal sample and  $C_A(x)$  is the A-law encoded sample.

The formula is shown in terms of absolute values of both the original and compressed signals since positive and negative values are compressed in an identical manner. The sign of the input is preserved in the output.

The formula shown above should not be implemented directly, since such an implementation would be slow. Encoding or decoding of A-law format is usually performed using look-up *Tables 1a/G.711* and *1b/G.711* shown in the CCITT specification *G.711* [CCITT]. Refer to the *G.711* specification for details.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## MuLawToALaw

*Converts samples from 8-bit  $\gamma$ -law encoded format to 8-bit A-law encoded format.*

---

### Syntax

```
IppStatus ippMuLawToALaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pDst</code>	Pointer to the destination vector.
<code>len</code>	Number of samples in the vector.

### Description

The function `ippMuLawToALaw` is declared in the `ipps.h` file. This function converts signal samples from 8-bit  $\mu$ -law encoded format in the vector `pSrc` to 8-bit A-law encoded format and stores them in the vector `pDst`.

The conversion of  $\mu$ -law format to A-law format is usually performed using look-up Table 3/G.711 shown in the CCITT specification G.711 [CCITT]. Refer to the G.711 specification for details.

Code [example 9-3](#) shows how to use the function `ippMuLawToALaw_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pDst</code> or <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## ALawToMuLaw

*Converts samples from 8-bit A-law encoded format to 8-bit  $\gamma$ -law encoded format.*

---

### Syntax

```
IppStatus ippsALawToMuLaw_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of samples in the vector.

### Description

The function `ippsMuLawToALaw` is declared in the `ipps.h` file. This function converts signal samples from 8-bit A-law encoded format in the vector *pSrc* to 8-bit  $\mu$ -law format and stores them in the vector *pDst*.

The conversion of A-law format to  $\mu$ -law format is usually performed using look-up *Table 4/G.711* shown in the CCITT specification *G.711* [[CCITT](#)]. Refer to the G.711 specification for details.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pDst</i> or <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

### Example 9-3 Using Comanding Functions

```
void compand( void ) {
    Ipp32f x[4] = { 0.1f, 0.2f, 0.3f, 0.4f };
    Ipp8u m[4], a[4];
    ippsLinToMuLaw_32f8u( x, m, 4 );
    ippsMuLawToALaw_8u( m, a, 4 );
    ippsALawToLin_8u32f( a, x, 4 );
    // now x must be close to original
    printf_32f("x =", x, 4, ippStsNoErr);
}
```

Output:

```
x = 0.099609 0.207031 0.304688 0.398438
```

## RT Audio Functions

The Intel IPP functions described in this section if properly combined can be used to construct the speech codec compliant to the Microsoft Real-Time (RT) Audio codec. The primitives are primarily designed to implement the well-defined, computationally expensive core operations that comprise the codec portion of the RT Audio system. The RT Audio codec comprises a fixed and variable rate algorithms intended for encoding 16-bit uniform PCM at the sampling rate of 8 and 16 KHz at bit rates 8800 and 18000 bps respectively.

The list of these functions is given in Table 9-22.

**Table 9-22. Intel MSRT Audio Functions**

Function Base Name	Operation
<a href="#">LPCToLSP_RTA</a>	Converts LP coefficients to LSP coefficients
<a href="#">LSPToLPC_RTA</a>	Converts LSP coefficients to LP coefficients
<a href="#">LevinsonDurbin_RTA</a>	Calculates LP coefficients from the autocorrelation coefficients
<a href="#">QMFGetStateSize_RTA</a>	Calculates the size of the QMF filter state memory
<a href="#">QMFInit_RTA</a>	Initializes the QMF filter state memory
<a href="#">QMFDecode_RTA</a>	Performs QMF synthesis
<a href="#">QMFEncode_RTA</a>	Performs QMF analysis
<a href="#">PostFilterGetStateSize_RTA</a>	Calculates the size of the post filter state memory
<a href="#">PostFilterInit_RTA</a>	Initializes the post filter state memory

Function Base Name	Operation
PostFilter_RTA	Restores speech signal from the residual
AdaptiveCodebookSearch_RTA	Searches for the adaptive codebook index and the lag, and computes the adaptive vector
FixedCodebookSearch_RTA, FixedCodebookSearchRandom_RTA	Searches for the fixed codebook vector
LSPQuant_RTA	Performs quantization of LSP coefficients
HighPassFilter_RTA	Performs high-pass filtering
BandPassFilter_RTA	Performs band pass filtering

## LPCToLSP\_RTA

*Converts LP coefficients to LSP coefficients.*

### Syntax

```
IppStatus ippsLPCToLSP_RTA_32f(const Ipp32f* pSrcLPC, Ipp32f* pDstLSP, int order);
```

### Parameters

<i>pSrcLPC</i>	Pointer to the input vector of length <i>order</i> that contains LP coefficients.
<i>pDstLSP</i>	Pointer to the output vector of length <i>order</i> that contains LSP coefficients.
<i>order</i>	Order of the LP filter (number of LP coefficients).

### Description

The function `ippsLPCToLSP_RTA` is declared in the `ippsc.h` file.

This function converts LP coefficients to LSP coefficients.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>IppStsRangeErr</code>	Indicates an error when <i>order</i> is less than or equal to 0, or greater than 16, or is not even number.

## LSPToLPC\_RTA

*Converts LSP coefficients to LP coefficients.*

---

### Syntax

```
IppStatus ippsLSPToLPC_RTA_32f(const Ipp32f* pSrcLSP, Ipp32f* pDstLPC, int order);
```

### Parameters

<i>pSrcLSP</i>	Pointer to the input vector of length <i>order</i> that contains LSP coefficients.
<i>pDstLPC</i>	Pointer to the output vector of length <i>order</i> that contains LP coefficients.
<i>order</i>	Order of the LP filter (number of LP coefficients).

### Description

The function `ippsLSPToLPC_RTA` is declared in the `ippsc.h` file.

This function converts LSP coefficients to LP coefficients.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>order</i> is less than or equal to 0, or greater than 16, or is not even number.

## LevinsonDurbin\_RTA

*Calculates LP coefficients from the autocorrelation coefficients.*

---

### Syntax

```
IppStatus ippsLevinsonDurbin_RTA_32f(const Ipp32f* pSrcAutoCorr, Ipp32f* pDstLPC, Ipp32f* pDstRC, int order);
```

## Parameters

<i>pSrcAutoCorr</i>	Pointer to the autocorrelation vector.
<i>pDstLPC</i>	Pointer to the LPC output vector.
<i>pDstRC</i>	Pointer to the RC output vector.
<i>order</i>	Order of the LP filter.

## Description

The function `ippsLevinsonDurbin_RTA` is declared in the `ippsc.h` file.

This function calculates linear prediction (LP) coefficients for the LP filter with the given *order* using the autocorrelation coefficients and Levinson-Durbin algorithm.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when <i>order</i> is less than or equal to 0.

## QMFFGetStateSize\_RTA

Calculates the size of the QMF filter state memory.

### Syntax

```
IppStatus ippsQMFFGetStateSize_RTA_32f(int* pSize);
```

## Parameters

<i>pSize</i>	Pointer to the size of the QMF filter memory.
--------------	---

## Description

The function `ippsQMFFGetStateSize_RTA` is declared in the `ippsc.h` file.

This function calculates the size of the Quadrature Mirror Filter (QMF) structure that must be initialized by the function `ippsQMFFInit_RTA` and used by the functions `ippsQMFFEncode_RTA` and `ippsQMFFDecode_RTA`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is NULL.

## QMFFinit\_RTA

*Initializes the QMF filter state memory.*

---

### Syntax

```
IppStatus ippQMFFinit_RTA_32f(IppQMFFState_RTA_32f* pQMFMem, int nTaps);
```

### Parameters

<i>pQMFMem</i>	Pointer to the QMF filter state structure.
<i>nTaps</i>	Number of taps, possible values are: 24, 48.

### Description

The function `ippQMFFinit_RTA` is declared in the `ippsc.h` file.

This function sets the number of filter taps *nTaps* and initializes Quadrature Mirror Filter (QMF) state memory *pQMFMem* that is used by the functions [ippQMFEencode\\_RTA](#) and [ippQMFFdecode\\_RTA](#). The size of the memory must be calculated by calling the function [ippQMFFGet-StateSize\\_RTA](#) beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when <i>pQMFMem</i> pointer is NULL.
<code>IppStsRangeErr</code>	Indicates an error when <i>nTaps</i> is not equal to 24 or 48.

## QMFFdecode\_RTA

*Performs QMF synthesis.*

---

### Syntax

```
IppStatus ippQMFFdecode_RTA_32f(const Ipp32f* pSrcLow, const Ipp32f* pSrcHigh,  
int len, Ipp32f* pDst, IppQMFFState_RTA_32f* pQMFMem);
```



## Parameters

<i>pSrcLow</i>	Pointer to the input low-band vector, its length is <i>len</i> .
<i>pSrcHigh</i>	Pointer to the input high-band vector, its length is <i>len</i> .
<i>len</i>	Number of samples in input vectors.
<i>pDst</i>	Pointer to the output restored vector, its length is $2 * len$ .
<i>pQMFMem</i>	Pointer to the QMF filter state memory.

## Description

The function `ippsQMFD.decode_RTA` is declared in the `ippsc.h` file.

This function decodes the input vectors with QMF synthesis filter. The QMF filter state memory must be initialized by calling the function `ippsQMFinitalize_RTA` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>IppStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or is greater than 320.

## QMFE.encode\_RTA

*Performs QMF analysis.*

---

## Syntax

```
ippStatus ippsQMFE.encode_RTA_32f(const Ipp32f* pSrc, int len, Ipp32f* pDstLow,
Ipp32f* pDstHigh, IppQMFEState_RTA_32f* pQMFMem);
```

## Parameters

<i>pSrc</i>	Pointer to the input vector, its length is <i>len</i> .
<i>len</i>	Number of samples in input vectors.
<i>pDstLow</i>	Pointer to the output low-band vector, its length is $len/2$ .

<i>pDstHigh</i>	Pointer to the output high-band vector, its length is <i>len/2</i> .
<i>pQMFMem</i>	Pointer to the QMF filter state memory.

### Description

The function `ippSQMFEncode_RTA` is declared in the `ippsc.h` file.

This function encodes the input vectors with QMF analysis filter. The QMF filter state memory must be initialized by calling the function `ippSQMFInit_RTA` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>IppStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0, or is greater than 320, or is not even number.

## PostFilterGetStateSize\_RTA

*Calculates the size of the short term post filter state memory.*

---

### Syntax

```
IppStatus ippPostFilterGetStateSize_RTA_32f(int* pSize);
```

### Parameters

<i>pSize</i>	Pointer to the size of the filter memory.
--------------	---

### Description

The function `ippPostFilterGetStateSize_RTA` is declared in the `ippsc.h` file.

This function calculates the size of post filter memory that is initialized by the function `ippPostFilterInit_RTA` and used by the function `ippPostFilter_RTA`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when <i>pSize</i> pointer is <code>NULL</code> .

## PostFilterInit\_RTA

*Initializes the post filter state memory.*

---

### Syntax

```
IppStatus ippsPostFilterInit_RTA_32f(IppPostFilterState_RTA_32s pStateMem);
```

### Parameters

*pStateMem*                      Pointer to the post filter state memory.

### Description

The function `ippsPostFilterInit_RTA` is declared in the `ippsc.h` file.

This function initializes the post state memory *pStateMem*. The size of the memory must be calculated by calling the function [ippsPostFilterGetStateSize\\_RTA](#) beforehand.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error when *pStateMem* pointer is NULL.

## PostFilter\_RTA

*Restores speech signal from the residual.*

---

### Syntax

```
IppStatus ippsPostFilter_RTA_32f_I(const Ipp32f* pSrcLPC, int order, Ipp32f*  
pSrcDstSpeech, int len, int pfType, Ipp32f* pMem[33],  
IppPostFilterState_RTA_32f* pStateMem);
```

### Parameters

*pSrcLPC*                          Pointer to the input vector with LPC coefficients.  
*order*                            Order of the LP filter.  
*pSrcDstSpeech*                  Pointer to the input/output speech vector.  
*len*                                Number of samples.  
*pfType*                           Specifies the type of the post filter. Possible values:

	0 - harmonic and full band coding; 1, 2, 3, 4 - subband coding respectively.
<i>pMem</i>	Pointer for the post filter memory vector.
<i>pStateMem</i>	Pointer for the post filter state memory.

## Description

The function `ippsPostFilter_RTA` is declared in the `ippsc.h` file.

This function restores speech signal from the residual. The function updates *pMem* and *pStateMem* memory appropriately. The post filter state memory must be initialized by calling the function `ippsPostFilterInit_RTA` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>IppStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsRangeErr</code>	Indicates an error when <i>order</i> is less than or equal to 1, or greater than 16, or <i>pFType</i> is less than 0 or greater than 7.

## AdaptiveCodebookSearch\_RTA

*Searches for the adaptive codebook index and the lag, and computes the adaptive vector.*

---

### Syntax

```
IppStatus ippsAdaptiveCodebookSearch_RTA_32f(const Ipp32f* pSrcAdptTarget,
const Ipp32f* pSrcImpulseResponse, int subFrameSize, const Ipp32f*
pSrcBoundary[4], Ipp32f* pSrcDstExc[372], Ipp32f* pSrcDstLag, int*
pDstAdptIndex, Ipp32f* pDstAdptContribution, int deltaSearchRange, int
searchFlag);
```

### Parameters

<i>pSrcAdptTarget</i>	Pointer to the input target signal [ <i>subFrameSize</i> ].
<i>pSrcImpulseResponse</i>	Pointer to the input impulse response [ <i>subFrameSize</i> ].
<i>subFrameSize</i>	Size of the subframe.

<i>pSrcBoundary</i>	Pointer to the input adaptive codebook search boundary.
<i>pSrcDstExc</i>	Pointer to the input/output excitation buffer.
<i>pSrcDstLag</i>	Pointer to the input/output lag.
<i>pDstAdptIndex</i>	Pointer to the output adaptive codebook search index.
<i>pDstAdptContribution</i>	Pointer to the output adaptive codebook search contribution [ <i>subFrameSize</i> ].
<i>deltaSearchRange</i>	Delta search range value.
<i>searchFlag</i>	Search flag, possible values: 0 - full search. 1 - search around open-loop pitch.

## Description

The function `ippsAdaptiveCodebookSearch_RTA` is declared in the `ippsc.h` file.

This function searches for the adaptive codebook index and the lag, using the target signal and the past filtered excitation, and computes the adaptive vector. The result is applied to the subframe.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrameSize</i> is less than or equal to 0; or <i>searchFlag</i> is not equal to 0 or 1.

## FixedCodebookSearch\_RTA, FixedCodebookSearchRandom\_RTA

*Searches for the fixed codebook vector.*

---

### Syntax

```

IppStatus ippsFixedCodebookSearch_RTA_32f(const Ipp32f* pSrcFixedTarget,
const Ipp32f* pSrcImpulseResponse, Ipp32f* pDstFixedVector, int subFrameSize,
int* pDstFixedIndex, Ipp32f* pDstGain, int numPulses, int complexity);

IppStatus ippsFixedCodebookSearchRandom_RTA_32f(const Ipp32f* pSrcFixedTarget,
const Ipp32f* pSrcImpulseResponse, Ipp32f* pDstFixedVector, int subFrameSize,
int* pDstFixedIndex, Ipp32f* pDstGain, int codebookSize, int nStage, int
gainLimit);

```

## Parameters

<i>pSrcFixedTarget</i>	Pointer to the input target signal [ <i>subFrameSize</i> ].
<i>pSrcImpulseResponse</i>	Pointer to the input impulse response [ <i>subFrameSize</i> ].
<i>pDstFixedVector</i>	Pointer to the output fixed codebook vector [ <i>subFrameSize</i> ].
<i>subFrameSize</i>	Size of the subframe.
<i>pDstFixedIndex</i>	Pointer to the output fixed codebook search index value.
<i>pDstGain</i>	Pointer to the output optimal gain value.
<i>numPulses</i>	Number of pulses.
<i>complexity</i>	Complexity control parameter, possible values are: 0, 1, 2, 3.
<i>codebookSize</i>	Size of the random codebook, possible values: 64, 160, 256.
<i>nStage</i>	Number of stage, possible values: 1, 2, 2.
<i>gainLimit</i>	Flag that specifies the gain constrain limit, possible values: 0, 1, 2, 3, 4.

## Description

The functions `ippsFixedCodebookSearch_RTA` and `ippsFixedCodebookSearchRandom_RTA` are declared in the `ippsc.h` file.

These functions search for the fixed-codebook vector and corresponding vector index in the fixed and random fixed codebooks respectively.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>subFrameSize</i> is less than or equal to 0, is greater than 80.
<code>ippStsRangeErr</code>	Indicates an error when <i>complexity</i> is less than 0 or greater than 3; or <i>nStage</i> is not equal to 0, 1, or 2; or <i>gainLimit</i> is not equal to 0, 1, 2, 3, 4; or <i>codebookSize</i> is less than 1 or greater than 256; or <i>numPulses</i> is less than 5 or greater than 10.

## LSPQuant\_RTA

*Performs quantization of LSP coefficients.*

---

### Syntax

```
IppStatus ippsLSPQuant_RTA_32f(const Ipp32f* pSrcLSP, const Ipp32f* pSrcLPC,
Ipp32f* pDstLSP, int order, int* pDstLSPIndex, IppPCMFrequency frequency,
int complexity);
```

### Parameters

<i>pSrcLSP</i>	Pointer to the input unquantized LSP vector.
<i>pSrcLPC</i>	Pointer to the input unquantized LPC vector
<i>pDstLSP</i>	Pointer to the output quantized LSP vector.
<i>order</i>	LP coefficients filter order, possible values are: 4, 6, 8, 10.
<i>pDstLSPIndex</i>	Pointer to the output vector of quantized LSP indexes.
<i>frequency</i>	Sampling frequency, possible values are: IPP_PCM_FREQ_8000 8000 Hz, IPP_PCM_FREQ_16000 16000 Hz, IPP_PCM_FREQ_22050 22050 Hz, IPP_PCM_FREQ_32000 32000 Hz.
<i>complexity</i>	Complexity control parameter, possible values are: 0, 1, 2, 3.

### Description

The function `ippsLSPQuant_RTA` is declared in the `ippsc.h` file.

This function performs quantization of LSP coefficients.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>IppStsRangeErr</code>	Indicates an error when <i>complexity</i> is less than 0 or greater than 3; or <i>order</i> is not equal to 4, 6, 8, 10; or <i>frequency</i> is not equal to the possible values.

## HighPassFilter\_RTA

*Performs high-pass filtering.*

---

### Syntax

```
IppStatus ippsHighPassFilter_RTA_32f(Ipp32f* pSrc, Ipp32f* pDst, int len,
int hpfOrder, IppPCMFrequency frequency, Ipp32f* pMem);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector [ <i>len</i> ].
<i>pDst</i>	Pointer to the destination vector [ <i>len</i> ].
<i>len</i>	Number of elements in the source and destination vectors.
<i>hpfOrder</i>	Filter order, possible values: 4, 5.
<i>frequency</i>	Sampling frequency, possible values are: IPP_PCM_FREQ_8000 8000 Hz, IPP_PCM_FREQ_16000 16000 Hz, IPP_PCM_FREQ_22050 22050 Hz, IPP_PCM_FREQ_32000 32000 Hz.
<i>pMem</i>	Pointer to the memory allocated for the filter [ <i>hpfOrder</i> ].

### Description

The function `ippsHighPassFilter_RTA` is declared in the `ippsc.h` file.

This function performs high-pass filtering and may be applied to the input or synthesized audio signal.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>IppStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsRangeErr</code>	Indicates an error when <i>hpfOrder</i> is not equal to 4 or 5; or <i>frequency</i> is not equal to the possible values.



## BandPassFilter\_RTA

*Performs band pass filtering.*

---

### Syntax

```
IppStatus ippsBandPassFilter_RTA_32f_I(Ipp32f* pSrcDst, int len, Ipp32f* pMem);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector [ <i>len</i> ].
<i>len</i>	Number of elements in the source and destination vectors.
<i>pMem</i>	Pointer to the memory allocated for the band pass filter [ <i>len</i> +2].

### Description

The function `ippsBandPassFilter_RTA` is declared in the `ippsc.h` file.

This function performs frequency enhancement around half of the bandwidth by cascading lowpass and highpass first order filters with the unit gain at stop band, and maximum gain ~ 3.56 db at about 8 kHz with 16kHz sampling rate.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>IppStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>IppStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

---

---

# Audio Coding Functions

Intel® IPP for audio coding includes general purpose functions applicable in several codecs and a number of specific functions for MPEG-4 audio encoder and decoder (see [ISO14496]), MP3 encoder and decoder. These functions implement pipeline blocks with large computational complexity.

The current set of functions is sufficient to implement a portable optimized MPEG-4 AAC Main profile decoder and a portable optimized MPEG-1, 2 Layer III encoder (see [ISO/IEC 11172-3] and [ISO13818]).

Also this chapter includes Spectral Band Replication (SBR) functions as an audio coding bandwidth extension technology, which is standardized in ISO/IEC 14496-3:2001/Amd.1:2003 [ISO14496A].

The full list of functions is given in Table 10-1 below.

**Table 10-1 Intel IPP Audio Coding Functions**

Function Base Name	Operation
Interleaved to Multi-Row Format Conversion Functions	
<a href="#">Interleave</a>	Converts signal from non-interleaved to interleaved format.
<a href="#">Deinterleave</a>	Converts signal from interleaved to non-interleaved format.
Spectral Data Prequantization Functions	
<a href="#">Pow34</a>	Raises a vector to the power of 3/4.
<a href="#">Pow43</a>	Raises a vector to the power of 4/3.
<a href="#">Pow43Scale</a>	Raises a vector to the power of 4/3 with scaling.
Scale Factors Calculation Functions	
<a href="#">CalcSF</a>	Restores actual scale factors from the bit stream values.
Mantissa Conversion and Scaling Functions	
<a href="#">Scale</a>	Applies scale factors to spectral bands in accordance with spectral bands boundaries.
<a href="#">MakeFloat</a>	Converts mantissa and exponent arrays to float arrays.
Modified Discrete Cosine Transform Functions	
<a href="#">MDCTFwdInitAlloc</a> , <a href="#">MDCTInvInitAlloc</a>	Create and initialize modified discrete cosine transform specification structure.
<a href="#">MDCTFwdInit</a> , <a href="#">MDCTInvInit</a>	Initialize modified discrete cosine transform specification structure.
<a href="#">MDCTFwdFree</a> , <a href="#">MDCTInvFree</a>	Close modified discrete cosine transform specification structure.
<a href="#">MDCTFwdGetSize</a> , <a href="#">MDCTInvGetSize</a>	Get the size of MDCT specification structure, initialization, and work buffer.
<a href="#">MDCTFwdGetBufSize</a> , <a href="#">MDCTInvGetBufSize</a>	Get the size of MDCT work buffer.
<a href="#">MDCTFwd</a> , <a href="#">MDCTInv</a>	Compute forward or inverse MDCT of a signal.

Function Base Name	Operation
Block Filtering Functions	
<a href="#">FIRBlockInitAlloc</a>	Initializes FIR block filter state.
<a href="#">FIRBlockFree</a>	Closes FIR block filter state.
<a href="#">FIRBlockOne</a>	Filters vector of sample through FIR block filter.
Frequency Domain Prediction Functions	
<a href="#">FDPInitAlloc</a>	Creates and initializes predictor state.
<a href="#">FDPInit</a>	Initializes FDP state.
<a href="#">FDPFree</a>	Closes FDP state.
<a href="#">FDPGetSize</a>	Gets size of FDP state structure.
<a href="#">FDPReset</a>	Resets predictors for all spectral lines.
<a href="#">FDPResetSfb</a>	Resets predictor-specific information in some scale factor bands.
<a href="#">FDPResetGroup</a>	Resets predictors for group of spectral lines.
<a href="#">FDPFwd</a>	Performs frequency domain prediction procedure and calculates prediction error.
<a href="#">FDPInv</a>	Retrieves input signal from prediction error, using frequency domain prediction procedure.
VLC Functions	
<a href="#">VLCDecodeEscBlock_MP3</a>	Parses the bitstream and decodes variable length code for MP3 using signed VLC tables.
<a href="#">VLCDecodeEscBlock_AAC</a>	Parses the bitstream and decodes variable length code for AAC using signed VLC tables.
<a href="#">VLCDecodeUTupleEscBlock_MP3</a>	Parses the bitstream and decodes variable length code for MP3 using unsigned multi-tupled VLC tables.
<a href="#">VLCDecodeUTupleEscBlock_AAC</a>	Parses the bitstream and decodes variable length code for AAC using unsigned multi-tupled VLC tables.
<a href="#">VLCCountEscBits_MP3</a>	Calculates the number of bits necessary for encoding in MP3 format.
<a href="#">VLCCountEscBits_AAC</a>	Calculates the number of bits necessary for encoding in AAC format.
<a href="#">VLCEncodeEscBlock_MP3</a>	Encodes an array of values into destination bitstream in MP3 format and advances bitstream pointer.
<a href="#">VLCEncodeEscBlock_AAC</a>	Encodes an array of values into destination bitstream in AAC format and advances bitstream pointer.
Psychoacoustic Functions	
<a href="#">Spread</a>	Computes spreading function.
Vector Quantization Functions	
<a href="#">VQCodeBookInitAlloc</a>	Allocates memory and initializes the codebook state structure.
<a href="#">VQCodeBookInit</a>	Initializes the codebook state structure.
<a href="#">VQCodeBookFree</a>	Closes the codebook state structure.

Function Base Name	Operation
VQCodeBookGetSize	Gets size of the codebook state structure.
VQPreliminarySelect	Selects candidates for the nearest code vector of codebooks.
VQMainSelect	Finds optimal indexes with minimal distortion.
VQIndexSelect	Finds optimal vector set for specified number of codebooks.
VQReconstruction	Reconstructs vectors from indexes.
MP3 Encoder Functions	
AnalysisPQMF_MP3	Implements stage 1 of MP3 hybrid analysis filterbank.
AnalysisFilterInit_PQMF_MP3	Initializes PQMF MP3 analysis specification structure.
AnalysisFilterInitAlloc_PQMF_MP3	Allocates memory and initializes the PQMF MP3 analysis specification structure.
AnalysisFilterGetSize_PQMF_MP3	Returns size of PQMF MP3 analysis specification structure.
AnalysisFilterFree_PQMF_MP3	Frees memory allocated for PQMF MP3 analysis specification structure.
AnalysisFilter_PQMF_MP3	Transforms PQMF MP3-processed subband signals into time domain samples.
MDCTFwd_MP3	Implements stage 2 of the MP3 hybrid analysis filterbank.
PsychoacousticModelTwo_MP3	Implements the ISO/IEC 11172-3 psych-acoustic model recommendation 2 to estimate the masked threshold and perceptual entropy associated with a block of PCM audio input.
JointStereoEncode_MP3	Transforms independent left and right channel spectral coefficient vectors into combined mid/side (MS) and/or intensity (IS) mode coefficient vectors suitable for quantization.
Quantize_MP3	Quantizes the spectral coefficients generated by the analysis filterbank.
PackScaleFactors_MP3	Applies noiseless coding to the scale factors and then packs the output into the bitstream buffer.
HuffmanEncode_MP3	Applies lossless Huffman encoding to the quantized samples and packs the output into the bitstream buffer.
PackFrameHeader_MP3	Packs the content of the frame header into the bitstream.
PackSideInfo_MP3	Packs the side information into the bitstream buffer.
BitReservoirInit_MP3	Initializes all elements of the bit reservoir state structure.
MP3 Decoder Functions	
UnpackFrameHeader_MP3	Unpacks the audio frame header.

Function Base Name	Operation
<a href="#">UnpackSideInfo_MP3</a>	Unpacks the side information from the input bitstream for use during the decoding of the associated frame.
<a href="#">UnpackScaleFactors_MP3</a>	Unpacks scale factors.
<a href="#">HuffmanDecode_MP3</a> , <a href="#">HuffmanDecodeSfb_MP3</a> , <a href="#">HuffmanDecodeSfbMbp_MP3</a>	Decodes Huffman data.
<a href="#">ReQuantize_MP3</a> , <a href="#">ReQuantizeSfb_MP3</a>	Requantizes the decoded Huffman symbols.
<a href="#">MDCTInv_MP3</a>	Performs the first stage of hybrid synthesis filter bank.
<a href="#">SynthPQMF_MP3</a>	Performs the second stage of hybrid synthesis filter bank.
<a href="#">SynthesisFilterInit_PQMF_MP3</a>	Initializes PQMF MP3 synthesis specification structure.
<a href="#">SynthesisFilterInitAlloc_PQMF_MP3</a>	Allocates memory for PQMF MP3 synthesis specification structure and initializes it.
<a href="#">SynthesisFilterGetSize_PQMF_MP3</a>	Returns size of PQMF MP3 synthesis specification structure.
<a href="#">SynthesisFilterFree_PQMF_MP3</a>	Frees memory allocated for PQMF MP3 synthesis specification structure.
<a href="#">SynthesisFilter_PQMF_MP3</a>	Transforms PQMF MP3-processed subband signals into time domain samples.
AAC Decoder Functions	
<a href="#">UnpackADIFHeader_AAC</a>	Gets the AAC ADIF format header.
<a href="#">UnpackADTSFrameHeader_AAC</a>	Gets ADTS frame header from the input bitstream.
<a href="#">DecodePrgCfgElt_AAC</a>	Gets program configuration element from the input bitstream.
<a href="#">DecodeChanPairElt_AAC</a>	Gets <i>channel_pair_element</i> from the input bitstream.
<a href="#">NoiselessDecoder_LC_AAC</a>	Decodes all the data for one channel.
<a href="#">DecodeDatStrElt_AAC</a>	Gets data stream element from the input bitstream.
<a href="#">DecodeFillElt_AAC</a>	Gets the fill element from the input bitstream.
<a href="#">QuantInv_AAC</a>	Performs inverse quantization of Huffman symbols for current channel in-place.
<a href="#">DecodeMsStereo_AAC</a>	Processes MS stereo for pair channels in-place.
<a href="#">DecodeIsStereo_AAC</a>	Processes intensity stereo for pair channels.
<a href="#">DeinterleaveSpectrum_AAC</a>	Deinterleaves the coefficients for short block.
<a href="#">DecodeTNS_AAC</a>	Decodes for Temporal Noise Shaping in-place.
<a href="#">MDCTInv_AAC_32s16s</a>	Maps the time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples.
<a href="#">MDCTInv_AAC_32s_I</a>	Computes inverse modified discrete cosine transform (MDCT), windowing and overlapping of signals.

Function Base Name	Operation
<a href="#">DecodeMainHeader_AAC</a>	Gets main header information and main layer information from bit stream.
<a href="#">DecodeExtensionHeader_AAC</a>	Get extension header information and extension layer information from bit stream.
<a href="#">DecodePNS_AAC</a>	Implements perceptual noise substitution coding within an ICS.
<a href="#">DecodeMsPNS_AAC</a>	Implements perceptual noise substitution coding in the case of joint coding.
<a href="#">DecodeChanPairElt_MP4_AAC</a>	Gets <i>channel_pair_element</i> from the input bitstream
<a href="#">LongTermReconstruct_AAC</a>	Uses Long Term Reconstruct to reduce the redundancy of a signal between successive coding frames.
<a href="#">MDCTFwd_AAC_32s</a>	Generates spectrum coefficient of PCM samples.
<a href="#">MDCTFwd_AAC_32s_I</a>	Computes forward modified discrete cosine transform (MDCT) of windowed signals.
<a href="#">EncodeTNS_AAC</a>	Performs reversion of TNS in the Long Term Reconstruct loop in-place.
<a href="#">LongTermPredict_AAC</a>	Gets the predicted time domain signals in the Long Term Reconstruct (LTP) loop.
<a href="#">NoiselessDecode_AAC</a>	Performs noiseless decoding.
<a href="#">LtpUpdate_AAC</a>	Performs required buffer update in the Long Term Reconstruct (LTP) loop.
<b>SBR Audio Encoder Functions</b>	
<a href="#">DetectTransient_SBR</a>	Estimates stationarity of signal by calculating deviation of spectrum audio signal.
<a href="#">EstimateTNR_SBR</a>	Measures Tonality-to-Noise Ratio of complex QMF subband samples.
<a href="#">AnalysisFilterEncGetSize_SBR</a>	Returns sizes of analysis SBR specification structures and work buffers.
<a href="#">AnalysisFilterEncInit_SBR</a>	Initializes analysis SBR specification structure.
<a href="#">AnalysisFilterEnc_SBR</a>	Performs subband filtering of the input audio signal.
<a href="#">AnalysisFilterEncInitAlloc_SBR</a>	Allocates memory and and initializes analysis SBR specification structure.
<a href="#">AnalysisFilterFree_SBR</a>	Closes analysis SBR specification structure.
<b>SBR Audio Decoder Functions</b>	
<a href="#">AnalysisFilterInitAlloc_SBR</a>	Allocates memory and initializes analysis SBR specification structure for real and complex signals.
<a href="#">SynthesisFilterInitAlloc_SBR</a>	Allocates memory and initializes synthesis SBR specification structure for real and complex signals.
<a href="#">SynthesisDownFilterInitAlloc_SBR</a>	Allocates memory and initializes downsample synthesis SBR specification structure for real and complex signals.

Function Base Name	Operation
<a href="#">AnalysisFilterFree_SBR</a>	Closes analysis SBR specification structure for real and complex signals.
<a href="#">SynthesisFilterFree_SBR</a>	Closes synthesis SBR specification structure for real and complex signals.
<a href="#">SynthesisDownFilterFree_SBR</a>	Closes downsample synthesis SBR specification structure for real and complex signals.
<a href="#">AnalysisFilterGetSize_SBR</a>	Returns size of analysis FilterSpec_SBR specification structures, init and work buffers.
<a href="#">SynthesisFilterGetSize_SBR</a>	Returns size of synthesis FilterSpec_SBR specification structures, init and work buffers.
<a href="#">SynthesisDownFilterGetSize_SBR</a>	Returns size of downsample synthesis FilterSpec_SBR specification structures, init and work buffers.
<a href="#">AnalysisFilterInit_SBR</a>	Initializes analysis SBR specification structure.
<a href="#">SynthesisFilterInit_SBR</a>	Initializes synthesis SBR specification structure.
<a href="#">SynthesisDownFilterInit_SBR</a>	Initializes downsample synthesis specification structure.
<a href="#">AnalysisFilter_SBR</a>	Transforms time domain signal output from the core decoder into frequency subband signals.
<a href="#">SynthesisFilter_SBR</a>	Transforms SBR-processed subband signals into time domain samples.
<a href="#">SynthesisDownFilter_SBR</a>	Transforms SBR-processed subband signals into time domain samples and performs downsampling at the same time.
<a href="#">PredictCoef_SBR</a>	Obtains prediction filter coefficients using covariance method.
<a href="#">PredictOneCoef_SBR</a>	Obtains one prediction filter coefficient using covariance method
Parametric Stereo Functions	
<a href="#">AnalysisFilter_PS</a>	Increases frequency resolution of the first lower subbands by hybrid filtering.
DTS Audio Coding Functions	
<a href="#">SynthesisFilterInit_DTS</a>	Initializes DTS synthesis filter specification structure.
<a href="#">SynthesisFilterInitAlloc_DTS</a>	Allocates memory for DTS synthesis filter specification structure and initilaizes it.
<a href="#">SynthesisFilterGetSize_DTS</a>	Returns size of DTS synthesis filter specification structure.
<a href="#">SynthesisFilterFree_DTS</a>	Frees memory allocated for DTS synthesis filter specification structure.
<a href="#">SynthesisFilter_DTS</a>	Transforms QMF DTS-processed subband signals into time domain samples.



## Interleaved to Multi-Row Format Conversion Functions

This section describes functions that enable you to perform transformations between interleaved and non-interleaved forms of multichannel signal. A signal in the interleaved form is a vector with interleaved samples for different channels. In the non-interleaved form, samples for each channel are stored in a separate vector.



**NOTE.** Although you may use both aligned and unaligned memory for arrays, you should expect slower performance when memory is not aligned.

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

### Interleave

*Converts signal from non-interleaved to interleaved format.*

#### Syntax

```
IppStatus ippsInterleave_16s(const Ipp16s** pSrc, int ch_num, int len, Ipp16s* pDst);
```

```
IppStatus ippsInterleave_32f(const Ipp32f** pSrc, int ch_num, int len, Ipp32f* pDst);
```

#### Parameters

<i>pSrc</i>	Array of pointers to the vectors [ <i>len</i> ] containing samples for particular channels.
<i>ch_num</i>	Number of channels.
<i>len</i>	Number of samples in each channel.
<i>pDst</i>	Pointer to the destination vector [ <i>ch_num</i> * <i>len</i> ] in interleaved format.

#### Description

This function is declared in the `ippac.h` header file. The function `ippsInterleave` transforms the signal from the non-interleaved to interleaved format according to the formula

$pDst[i] = pSrc[i \text{ div } ch\_num][i \bmod ch\_num], 0 \leq i < ch\_num * len,$

where `div` is the integer part of the quotient and `mod` is the remainder.

Below see code example 10-1 of using `ippsInterleave_16s` function.

The function `ippsInterleave_32f` is used in the float-point version of MP3 encoder included into Intel IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsNumChannelsErr</code>	Indicates an error when <code>ch_num</code> is less than or equal to 0.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.

## Example 10-1 `ippsInterleave_16s` Usage

```
IppStatus interleave(void)

{
    Ipp16s i0[5] = {0,1,2,3,4};
    Ipp16s i1[5] = {10,11,12,13,14};
    Ipp16s pDst[10];
    Ipp16s *pSrc[2];
    IppStatus st;
    int i;
```

```

    pSrc[0] = i0; pSrc[1] = i1;

    st = ippsInterleave_16s(pSrc, 2, 5, pDst);

    printf("\n pDst = ");
    for (i = 0; i < 10; i++)
        printf("%i ", pDst[i]);
    printf("\n");
    return st;
}

//Output:
//  pDst = 0 10 1 11 2 12 3 13 4 14

```

## Deinterleave

*Converts signal from interleaved to non-interleaved format.*

---

### Syntax

```

IppStatus ippsDeinterleave_16s(const Ipp16s* pSrc, int ch_num, int len,
Ipp16s** pDst);

IppStatus ippsDeinterleave_32f(const Ipp32f* pSrc, int ch_num, int len,
Ipp32f** pDst);

```

### Parameters

<i>pSrc</i>	Pointer to vector [ <i>ch_num</i> * <i>len</i> ] of interleaved samples.
<i>ch_num</i>	Number of channels.
<i>len</i>	Number of samples in each channel.
<i>pDst</i>	Array of pointers to the vectors [ <i>len</i> ] to be filled with samples of particular channels.

### Description

This function is declared in the `ippac.h` header file. The function `ippsDeinterleave` transforms the input signal from interleaved to non-interleaved format according to the formula

$pDst[i][j] = pSrc[i + j * ch\_num], 0 \leq i < ch\_num, 0 \leq j < len.$

See below code example 10-2 of using `ippsDeinterleave_16s` function.

The function `ippsInterleave_16s` is used in the AAC and MP3 encoders and the function `ippsInterleave_32f` is used in the AC3 decoder (float-point version), and AAC and MP3 encoders (float-point version) included into Intel IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> or <code>ch_num</code> is less than or equal to 0.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.

## Example 10-2. `ippsDeinterleave_16s` Usage

```
IppStatus deinterleave(void)
{
    Ipp16s i0[5];
    Ipp16s i1[5];
    Ipp16s pSrc[10] = {0,1,2,3,4,5,6,7,8,9};
    Ipp16s *pDst[2];
    IppStatus st;
    int i;
```

```

pDst[0] = i0; pDst[1] = i1;

st = ippsDeinterleave_16s(pSrc, 2, 5, pDst);

printf("\n pDst[0] = ");
for (i = 0; i < 5; i++)
    printf("%i ", pDst[0][i]);
printf("\n pDst[1] = ");
for (i = 0; i < 5; i++)
    printf("%i ", pDst[1][i]);
printf("\n");

return st;
}

//Output:
//  pDst[0] = 0 2 4 6 8

//  pDst[1] = 1 3 5 7 9

```

## Spectral Data Prequantization Functions

MPEG-1, 2 Layer III and MPEG-4 AAC audio encoders raise the spectral data to the power of  $3/4$  before quantization to provide a more consistent signal-to-noise ratio over the range of quantized values. The re-quantizers in the decoders linearize the values by raising their output to the power of  $4/3$ .

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

### Pow34

*Raises a vector to the power of  $3/4$ .*

#### Syntax

```

IppStatus ippsPow34_32f16s(const Ipp32f* pSrc, Ipp16s* pDst, int len);

IppStatus ippsPow34_32f(const Ipp32f* pSrc, Ipp32f* pDst, int len);

```

```
IppStatus ippsPow34_16s_Sfs(const Ipp16s* pSrc, int inscaleFactor, int Ipp16s*
pDst, int scaleFactor, int len);
```

## Parameters

<i>pSrc</i>	Pointer to the input data vector [ <i>len</i> ].
<i>inscaleFactor</i>	Input data scalefactor value.
<i>pDst</i>	Pointer to the output data vector [ <i>len</i> ].
<i>scaleFactor</i>	Scalefactor value.
<i>len</i>	Number of elements in the input and output vectors.

## Description

This function is declared in the `ippac.h` header file. The function `ippsPow34` performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = |pSrc[i]|^{3/4}, 0 \leq i < len,$$

and stores the result in *pDst*.

For example, in MPEG-1 Layer III the quantization of the complete vector of spectral values is done according to the following formula:

$$ix(i) = nint\left(\left(\frac{|xr(i)|}{4\sqrt[4]{2^{qquant+quantanf}}}\right)^{0.75} - 0.0946\right)$$

Here *ix* is the array of quantized values, *i* is the number of values in the array, *nint* is the function used to round non-integer values to the nearest integer value, *xr* is the vector of the magnitudes of the spectral values, *qquant* is the quantizer step size information, and *quantanf* is a constant that depends on the spectral flatness measure.

You can use the function `ippsPow34_32f16s` for this operation. However, if the maximum of all quantized values is outside the table range or the overall bit sum exceeds the available bits, you should repeat this operation several times with a new *qquant* value and the same *xr* array.

Alternatively, you can use the function `ippsPow34_32f` to calculate the power of 3/4 for *xr* only once before the inner iteration loop and use it during every quantization iteration with the constant multiplier and convert the resulting values to `short`.

$$\text{multiplier} = \left( \frac{1}{\sqrt[4]{2^{q_{\text{quant}} + \text{quant\_inf}}}} \right)^{0.75}$$

This operation is supported by the function `ippsMulC_Low_32f16s`.

Below see example 10-3 of using `ippsPow34_32f` and `ippsPow34_16s_Sfs`.

The function `ippsPow34` is used in the float-point versions of AAC and MP3 encoders included into Intel IPP Samples. See [introduction](#) to this section.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.

**Example 10-3 ippsPow34 Usage**

```
IppStatus pow34(void)
{
    Ipp32f pSrc[5] = {-1234.56, 0, 45.89, -3.896, 45328.562};
    Ipp32f pDst[5], pDst32[5];
    Ipp16s pSrc16[5], pDst16[5];
    int inscaleFactor = 1;
    int scaleFactor = -3;
    IppStatus st;
    int i;

    st = ippsPow34_32f(pSrc, pDst, 5);
    if (st < ippStsOk) return st;
    printf("\n pDst = ");
    for (i = 0; i < 5; i++)
        printf("%f ", pDst[i]);
    printf("\n");

    st = ippsConvert_32f16s_Sfs(pSrc, pSrc16, 5, ippRndNear, inscaleFactor);
    if (st < ippStsOk) return st;
    st = ippsPow34_16s_Sfs(pSrc16, inscaleFactor, pDst16, scaleFactor, 5);
    if (st < ippStsOk) return st;
    st = ippsConvert_16s32f_Sfs(pDst16, pDst32, 5, -scaleFactor);
    printf("Before scaling\n pDst16 = ");
    for (i = 0; i < 5; i++)
        printf("%i ", pDst16[i]);
    printf("\n");
    printf("After scaling\n pDst16 = ");
    for (i = 0; i < 5; i++)
        printf("%f ", pDst32[i]);
```



```

    printf("\n");
    return st;
}
//Output:
//   pDst = 208.273575 0.000000 17.631470 2.773092 3106.554443

//   Before scaling
//   pDst16 = 1666 0 141 23 24853

//   After scaling
//   pDst16 = 208.250000 0.000000 17.625000 2.875000 3106.625000

```

## Pow43

*Raises a vector to the power of 4/3.*

---

### Syntax

```

IppStatus ippPow43_16s32f(const Ipp16s* pSrc, Ipp32f* pDst, int len);

```

### Parameters

<i>pSrc</i>	Pointer to the input data vector [ <i>len</i> ].
<i>pDst</i>	Pointer to the output data vector [ <i>len</i> ].
<i>len</i>	Number of elements in the input and output vectors.

### Description

This function is declared in the `ippac.h` header file. The function `ippPow43` performs the calculation for each element of *pSrc* by the formula

$$pDst[i] = \text{sign}(pSrc) * |pSrc[i]|^{4/3}, 0 \leq i < len,$$

and stores the result in *pDst*.

The function `ippPow43` is used in the float-point versions of AAC and MP3 decoders included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.



**NOTE.** The input values in the *pSrc* array can not be greater than 8206. The function does not check if the input values meet this requirement in order not to cause performance slowdown. If the input values in the array exceed the limit, the function may operate incorrectly. No error code is returned in this case.

---

## Pow43Scale

*Raises a vector to the power of 4/3 with scaling.*

---

### Syntax

```
IppStatus ippsPow43Scale_16s32s_Sf(const Ipp16s* pSrc, Ipp32f* pDst, const
Ipp16s *pScaleFactor, const Ipp32s *pBandOffset, int offset, int bandsNumber,
int groupLen, int scalef);
```

### Parameters

<i>pSrc</i>	Pointer to the input data vector.
<i>pDst</i>	Pointer to the output data vector.
<i>pScaleFactor</i>	Pointer to the data array containing scale factors. The size of the array must be not less than <i>bandsNumber</i> .
<i>pBandOffset</i>	Pointer to the vector of band offsets. The size of array must be not less than <i>bandsNumber</i> + 1.
<i>offset</i>	Scale factors offset.
<i>bandsNumber</i>	Number of bands to which scale factors are applied.
<i>groupLen</i>	Number of windows in the current group.
<i>scalef</i>	Scale factor of the output data.

## Description

This function is declared in the `ippac.h` header file. The `ippsPow43Scale` function performs the following operation:

$$pDst[i] = pSrc[i]^{4/3} * 2^{1/4(pScaleFactor[band] - offset)}.$$

This function does not perform any saturation so the user must consider overflow possibility. Absolute values of `pSrc[i]` must be less than  $2^{14}$ .

Below see example 10-4 of using `ippsPow43Scale` function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>groupLen</code> or <code>bandsNumber</code> is less than 0.

## Example 10-4 ippsPow43Scale Usage

```
#undef SF_OFFSET

#define SF_OFFSET 100

IppStatus pow43scale(void)
{
    Ipp16s pSrc[40];
    Ipp32s pDst0[40], pDst1[40];
    Ipp16s pScaleFactor[5] = {65, 128, 163, 58, 100};
    Ipp32s pBandOffset0[6] = {0,1,3,8,13,20};
    Ipp32s pBandOffset1[6];
    int groupLen = 2;
    int bandsNumber = 5;
    int i;
    IppStatus st;
    for (i = 0; i < 40; i++)
        pSrc[i] = (Ipp16s)(100 * i);
    /* The are two ways of using ippsPow43Scale_16s32s_Sf when groupLen */
    /* is not equal to 1 */
    /* First way is to use function as is */
    st = ippsPow43Scale_16s32s_Sf(pSrc, pDst0, pScaleFactor, pBandOffset0,
                                  SF_OFFSET, bandsNumber, groupLen, 0);
    if (st != ippStsOk) return st;
    /* The second way is to recalculate pBandOffset */
    pBandOffset1[0] = pBandOffset0[0];
    for (i = 0; i < bandsNumber; i++)
        pBandOffset1[i+1] = pBandOffset1[i] +
```

---

```

        (pBandOffset0[i+1] - pBandOffset0[i]) * groupLen;
/* and use function ippsPow43Scale_16s32s_Sf with groupLen is equal to 1 */
/* The result will be the same */
st = ippsPow43Scale_16s32s_Sf(pSrc, pDst1, pScaleFactor, pBandOffset1,
                             SF_OFFSET, bandsNumber, 1, 0);

printf("\n pDst0 = ");
for (i = 0; i < 40; i++)
    printf("%i ", pDst0[i]);
printf("\n");
printf("\n pDst1 = ");
for (i = 0; i < 40; i++)
    printf("%i ", pDst1[i]);
printf("\n");
return st;
}

//Output:
//  pDst1 = 0 1 149709 257062 377244 507968 278884317

//          342520430 409269211 478864278 551089873

//          625767026 702744444 781892234 863097398

//          946260634 12 14 15 16 17 18 19 20 22 23

//          35751 37596 39464 41355 43267 45201 47155

//          49130 51125 53140 55174 57227 59298 61388
//
//  pDst1 = 0 1 149709 257062 377244 507968 278884317

//          342520430 409269211 478864278 551089873

```

```
//          625767026 702744444 781892234 863097398

//          946260634 12 14 15 16 17 18 19 20 22 23

//          35751 37596 39464 41355 43267 45201 47155

//          49130 51125 53140 55174 57227 59298 61388
```

## Scale Factors Calculation Functions

In MPEG-2, 4 GA AAC decoder scale factors extracted from the bitstream require an additional restoring procedure. The function `ippsCalcSF` restores actual scale factors from values transmitted in the bit stream.

### CalcSF

*Restores actual scale factors from the bit stream values.*

---

#### Syntax

```
IppStatus ippsCalcSF_16s32f(const Ipp16s* pSrc, int offset, Ipp32s* pDst,
int len);
```

#### Parameters

<i>pSrc</i>	Pointer to the input data array.
<i>offset</i>	Scale factors offset.
<i>pDst</i>	Pointer to the output data array.
<i>len</i>	Number of elements in the vector.

#### Description

This function is declared in the `ippac.h` header file. The function `ippsCalcSF` restores actual scale factors from the values *pSrc* transmitted in the bit stream, using the common scale factor offset. Computation is performed according to the following formula

$$pDst[i] = 2^{1/4(pSrc[i] - offset)}, 0 \leq i < len.$$

Restored scale factors are written into *pDst*.

Below see example 10-5 of using `ippsCalcSF_16s32f` function.

The function `ippsCalcSF` is used in the float-point versions of AAC and MP3 decoders included into Intel IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.



**NOTE.** The input values in the `pSrc` array must be in the range  $-512 \leq (pSrc[i] - offset) < 512$ . Otherwise the function may operate incorrectly. No error message is returned in this case.

### Example 10-5 ippsCalcSF Usage

```
#undef SF_OFFSET
#define SF_OFFSET 100
IppStatus calcSF(void)
{
    Ipp16s pSrc[5] = {65, 128, 163, 58, 100};
    Ipp32f pDst[5];
    IppStatus st = ippsCalcSF_16s32f(pSrc, SF_OFFSET, pDst, 5);
    int i;
    printf("\n pDst = ");
    for (i = 0; i < 5; i++)
        printf("%f ", pDst[i]);
    printf("\n");
}
//Output:
//  pDst = 0.002323 128.000000 55108.988281 0.000691 1.000000
```

## Mantissa Conversion and Scaling Functions

The scale application procedure is necessary for MPEG-2, 4 GA AAC decoder to restore the original spectral values from a set of the inversely quantized spectral coefficients. The whole spectrum is divided into a set of scale factor bands. Since the widths differ from band to band, the band positions are defined by the offset vector.

The mantissa conversion procedure is necessary for the AC3 decoder to restore the original spectral values from a set of the exponents and mantissas in the bitstream.

### Scale

*Applies scale factors to spectral bands in accordance with spectral bands boundaries.*

---

#### Syntax

```
IppStatus ippScale_32f_I(Ipp32f* pSrcDst, const Ipp32f* pSF, const int* pBandOffset, int bandsNumber);
```

#### Parameters

<i>pSrcDst</i>	Pointer to the input and output data array. The size of array must be not less than <i>pBandOffset</i> [ <i>bands_number</i> ].
<i>pSF</i>	Pointer to the data array containing scale factors. The size of the array must be not less than <i>bands_number</i> .
<i>pBandOffset</i>	Pointer to the vector of band offsets. The size of array must be not less than <i>bands_number</i> + 1.
<i>bandsNumber</i>	Number of bands to which scale factors are applied.

#### Description

This function is declared in the `ippac.h` header file. The function `ippScale_32f_I` computes scaled values from the input vector, the vector of scale factors, and the vector of band offsets. Operations are performed in-place.

This function applies the set of scale factors *pSF* with *bandsNumber* elements to the bands constructed from the input vector *pSrc*. Band boundaries are defined by the vector of the band offsets *pBandOffset*. All values in each band are multiplied by the corresponding scale factor.





**NOTE.** The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of *pSrcDst* vector is contained in the *pBandsOffset[bandsNumber]* element.

Below see example 10-6 of using `ippsScale_32f_I`.

The function `ippsScale` is used in the float-point versions of AAC and MP3 decoders included into Intel IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSrcDst</i> or <i>pBandsOffset</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>bandsNumber</i> is less than or equal to 0.

## Example 10-6 ippsScale Usage

```
IppStatus scale(void)
{
    Ipp32f pSrcDst[20];
    Ipp32f pSF[5] = {0.002323, 128, 55108.988281, 0.000691, 1};
    int pBandOffset[6] = {0,1,3,8,13,20};
    IppStatus st;
    int i;
    for (i = 0; i < 20; i++)
        pSrcDst[i] = (Ipp32f)i;
    st = ippsScale_32f_I(pSrcDst, pSF, pBandOffset, 5);
    printf("\n pSrcDst = ");
    for (i = 0; i < 20; i++)
        printf("%f ", pSrcDst[i]);
    printf("\n");
}

//Output:
//   pDst = 0.000000 128.000000 256.000000 165326.968750 220435.953125
//           275544.937500 330653.937500 385762.906250 0.005528 0.006219
//           0.006910 0.007601 0.008292 13.000000 14.000000
//           15.000000 16.000000 17.000000 18.000000 19.000000
```

## MakeFloat

*Converts mantissa and exponent arrays to float arrays.*

---

### Syntax

```
IppStatus ippsMakeFloat_16s32f (Ipp16s* inmant, Ipp16s* inexp, Ipp32s size,
Ipp32f* outfloat);
```

## Parameters

<i>inmant</i>	Array of mantissas.
<i>inexp</i>	Array of exponents.
<i>size</i>	Number of array elements.
<i>outfloat</i>	Array of resulting float arrays.

## Description

The function `ippsMakeFloat` is declared in the `ippac.h` header file. This function converts the mantissa and exponent arrays decoded from the bitstream to the float array of spectral samples by the formula:

$$outfloat[i] = inmant[i] \times 2^{-inexp[i]-15}, \text{ where } i = 0 \dots size.$$

The conversion serves to improve application performance when decoding bitstreams in the AC3 format.

Below see example 10-7 of using `ippsMakeFloat_16s32f`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>inmant</i> , <i>inexp</i> or <i>outfloat</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>size</i> is less than or equal to 0.

## Example 10-7 ippsMakeFloat Usage

```
IppStatus makefloat(void)
{
    Ipp16s inmant[5] = {1234, 56, 25907, 3498, 27854};
    Ipp16s inexp[5] = {-2, -1, 0, 1, 2};
    Ipp32f outfloat[5];
    IppStatus st = ippsMakeFloat_16s32f(inmant, inexp, 5, outfloat);
    int i;
    printf("\n outfloat = ");
    for (i = 0; i < 5; i++)
        printf("%f ", outfloat[i]);
    printf("\n");
}

//Output:
//   outfloat = 0.150635 0.003418 0.790619 0.053375 0.212509
```

## Modified Discrete Cosine Transform Functions

This section describes Intel IPP functions that compute the modified discrete cosine transform (MDCT) of a signal. MDCT is a lapped orthogonal transform widely used in different audio codecs, such as MPEG-1, MPEG-2, AC-3, and AAC.

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

### MDCTFwdInitAlloc, MDCTInvInitAlloc

*Create and initialize modified discrete cosine transform specification structure.*

---

#### Syntax

```
IppStatus ippsMDCTFwdInitAlloc_32f(IppsMDCTFwdSpec_32f** ppMDCTSpec, int
len);
```

```
IppStatus ippMDCTFwdInitAlloc_16s(IppsMDCTFwdSpec_32s** ppMDCTSpec, int len);  
  
IppStatus ippMDCTInvInitAlloc_32f(IppsMDCTInvSpec_32f** ppMDCTSpec, int len);
```

## Parameters

<i>ppMDCTSpec</i>	Pointer to pointer to MDCT specification structure to be created.
<i>len</i>	Number of samples in MDCT. Since this set of functions was designed specially for audio coding, only the following values of length are supported: 12, 36, and $2^k$ , where $k \geq 5$ . These values are the only values that appear in audio coding.

## Description

These functions are declared in the `ippac.h` header file. The functions `ippMDCTFwdInitAlloc` and `ippMDCTInvInitAlloc` create and initialize MDCT specification structure *ppMDCTSpec* with the specified transform length *len*.

The function `ippMDCTFwdInitAlloc` initializes the forward MDCT specification structure. The function `ippMDCTInvInitAlloc` initializes the inverse MDCT specification structure.

`ippMDCTFwdInitAlloc_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders, `ippMDCTFwdInitAlloc_16s` is used in the fixed-point version of AAC encoder, `ippMDCTInvInitAlloc_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into Intel IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppMDCTSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> does not belong to the above set of admissible values.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.



**NOTE.** `ippMDCTFwd_16s_Sfs` does not support the length values of 12 and 36.

## MDCTFwdInit, MDCTInvInit

*Initialize modified discrete cosine transform specification structure.*

---

### Syntax

```
IppStatus ippMDCTFwdInit_32f(IppsMDCTFwdSpec_32f** ppMDCTSpec, int len,
Ipp8u* pMemSpec, Ipp8u* pMemInit);

IppStatus ippMDCTFwdInit_16s(IppsMDCTFwdSpec_32f** ppMDCTSpec, int len,
Ipp8u* pMemSpec, Ipp8u* pMemInit);

IppStatus ippMDCTInvInit_32f(IppsMDCTInvSpec_32f** ppMDCTSpec, int len,
Ipp8u* pMemSpec, Ipp8u* pMemInit);
```

### Parameters

<i>ppMDCTSpec</i>	Pointer to pointer to MDCT specification structure to be created.
<i>len</i>	Number of samples in MDCT. Since this set of functions was designed specially for audio coding, only the following values of length are supported: 12, 36, and $2^k$ , where $k \geq 5$ . These values are the only values that appear in audio coding.
<i>pMemSpec</i>	Pointer to the area for MDCT specification structure.
<i>pMemInit</i>	Pointer to the buffer that is used for initialization.

### Description

These functions are declared in the `ippac.h` header file. The functions `ippMDCTFwdInit` and `ippMDCTInvInit` initialize MDCT specification structure *pMDCTSpec* with the specified transform length *len*.

The function `ippMDCTFwdInit` initializes the forward MDCT specification structure. The function `ippMDCTInvInit` initializes the inverse MDCT specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppMDCTSpec</i> , <i>pMemSpec</i> or <i>pMemInit</i> pointer is <code>NULL</code> .

`ippStsSizeErr`

Indicates an error when *len* does not belong to the above set of admissible values.



**NOTE.** `ippsMDCTFwd_16s_Sfs` does not support the length values of 12 and 36.

## MDCTFwdFree, MDCTInvFree

*Closes modified discrete cosine transform specification structure.*

### Syntax

```
IppStatus ippsMDCTFwdFree_32f(IppsMDCTFwdSpec_32f* pMDCTSpec);
IppStatus ippsMDCTFwdFree_16s(IppsMDCTFwdSpec_16s* pMDCTSpec);
IppStatus ippsMDCTInvFree_32f(IppsMDCTInvSpec_32f* pMDCTSpec);
```

### Parameters

*pMDCTSpec*

Pointer to the MDCT specification structure to be closed.

### Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdFree` and `ippsMDCTInvFree` close the MDCT structure *pMDCTSpec* by freeing all memory associated with the specification created by `ippsMDCTFwdInitAlloc` or `ippsMDCTInvInitAlloc` functions.

Call either `ippsMDCTFwdFree` or `ippsMDCTInvFree` after the transform is completed.

The function `ippsMDCTFwdFree` closes the forward MDCT specification structure.

The function `ippsMDCTInvFree` closes the inverse MDCT specification structure.

`ippsMDCTFwdFree_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders, `ippsMDCTFwdFree_16s` is used in the fixed-point version of AAC encoder, `ippsMDCTInvFree_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into IPP Samples. See [introduction](#) to this section.

### Return Values

`ippStsNoErr`

Indicates no error.

`ippStsNullPtrErr` Indicates an error when the `pMDCTSpec` pointer is NULL.

`ippStsContextMatchErr` Indicates an error when the specification identifier `pMDCTSpec` is incorrect.

## MDCTFwdGetSize, MDCTInvGetSize

*Get the sizes of MDCT specification structure, MDCT initialization, and MDCT work buffer.*

---

### Syntax

```
IppStatus ippMDCTFwdGetSize_32f(int len, int* pSizeSpec, int* pSizeInit,
int* pSizeBuf);

IppStatus ippMDCTFwdGetSize_16s(int len, int* pSizeSpec, int* pSizeInit,
int* pSizeBuf);

IppStatus ippMDCTInvGetSize_32f(int len, int* pSizeSpec, int* pSizeInit,
int* pSizeBuf);
```

### Parameters

<code>len</code>	Number of samples in MDCT. Since this set of functions is designed specially for audio coding, only the following values of length are supported: 12, 36, and $2^k$ , where $k \geq 5$ . These values are the only values that appear in audio coding.
<code>pSizeSpec</code>	Address of the MDCT specification structure size value in bytes.
<code>pSizeInit</code>	Address of the MDCT initialization buffer size value in bytes.
<code>pSizeBuf</code>	Address of the MDCT work buffer size value in bytes.

### Description

These functions are declared in the `ippac.h` header file. The functions `ippMDCTFwdGetSize` and `ippMDCTInvGetSize` get the specification structure, initialization, and work buffer sizes of the MDCT described by the length and store the result in `pSizeSpec`, `pSizeInit`, and `pSizeBuf`.



The function `ippsMDCTFwdGetSize` gets the size of the work buffer for the forward MDCT. The function `ippsMDCTInvGetSize` gets the size of the work buffer for the inverse MDCT.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSizeSpec</code> , <code>pSizeInit</code> , or <code>pSizeBuf</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> does not belong to the above set of admissible values.



**NOTE.** `ippsMDCTFwd_16s_Sfs` does not support the length values of 12 and 36.

## MDCTFwdGetBufSize, MDCTInvGetBufSize

*Gets the size of MDCT work buffer.*

### Syntax

```
IppStatus ippsMDCTFwdGetBufSize_32f(const IppsMDCTFwdSpec_32f* pMDCTSpec,
int* pSize);
```

```
IppStatus ippsMDCTFwdGetBufSize_16s(const IppsMDCTFwdSpec_16s* pMDCTSpec,
int* pSize);
```

```
IppStatus ippsMDCTInvGetBufSize_32f(const IppsMDCTInvSpec_32f* pMDCTSpec,
int* pSize);
```

### Parameters

<code>pMDCTSpec</code>	Pointer to the MDCT specification structure.
<code>pSize</code>	Address of the MDCT work buffer size value in bytes.

### Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwdGetBufSize` and `ippsMDCTInvGetBufSize` get the work buffer size of the MDCT described by the specification structure `pMDCTSpec` and store the result in `pSize`.

The function `ippsMDCTFwdGetBufSize` gets the size of the work buffer for the forward MDCT.

The function `ippsMDCTInvGetBufSize` gets the size of the work buffer for the inverse MDCT.

`ippsMDCTGetBufSize_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders, `ippsMDCTFwdGetBufSize_16s` is used in the fixed-point version of AAC encoder, `ippsMDCTInvGetBufSize_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pMDCTSpec</code> pointer or <code>pSize</code> value is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification structure <code>pMDCTSpec</code> is invalid.

## MDCTFwd, MDCTInv

*Computes forward or inverse modified discrete cosine transform (MDCT) of a signal.*

---

### Syntax

```

IppStatus ippsMDCTFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsMDCTFwdSpec_32f* pMDCTSpec, Ipp8u* pBuffer);

IppStatus ippsMDCTInv_32f(const Ipp32f* pSrc, Ipp32f* pDst, const
IppsMDCTInvSpec_32f* pMDCTSpec, Ipp8u* pBuffer);

IppStatus ippsMDCTFwd_32f_I(Ipp32f* pSrcDst, const IppsMDCTFwdSpec_32f*
pMDCTSpec, Ipp8u* pBuffer);

IppStatus ippsMDCTInv_32f_I(Ipp32f* pSrcDst, const IppsMDCTInvSpec_32f*
pMDCTSpec, Ipp8u* pBuffer);

IppStatus ippsMDCTFwd_16s_Sfs(const Ipp16s* pSrc, Ipp16s* pDst, const
IppsMDCTFwdSpec_32f* pMDCTSpec, int scaleFactor, Ipp8u* pBuffer);

```

### Parameters

<code>pSrc</code>	Pointer to the input data array.
<code>pDst</code>	Pointer to the output data array.
<code>pSrcDst</code>	Pointer to the input and output data array for the in-place operations.

---

<i>pMDCTSpec</i>	Pointer to the MDCT specification structure.
<i>pBuffer</i>	Pointer to the MDCT work buffer.
<i>scaleFactor</i>	Scalefactor value.

### Description

These functions are declared in the `ippac.h` header file. The functions `ippsMDCTFwd` and `ippsMDCTInv` compute the forward and inverse modified discrete cosine transform (MDCT), respectively.

In the following definition of MDCT,  $N$  denotes the length and  $n_0 = (N/2+1)/2$ .

For the forward MDCT,  $x(n)$  is `pSrc[n]` and  $y(k)$  is `pDst[k]`, whereas for the inverse MDCT  $x(n)$  is `pDst[n]` and  $y(k)$  is `pSrc[k]`.

The forward MDCT is defined by the formula:

$$y(k) = 2 \cdot \sum_{n=0}^{N-1} x(n) \cos\left(\frac{\pi}{N}(n+n_0)(2k+1)\right)$$

for  $0 \leq k < N/2$ .

The inverse MDCT is defined as

$$x(n) = \frac{2}{N} \cdot \sum_{k=0}^{\frac{N}{2}-1} y(k) \cos\left(\frac{\pi}{N}(n+n_0)(2k+1)\right),$$

for  $0 \leq k < N$ .

The `pBuffer` argument provides the MDCT functions with the necessary work memory and helps to avoid memory allocation within the functions. The buffer may also increase the performance if the MDCT functions use the result of the previous operation stored in cache as an input array.

See code example 10-8 below of using `ippsMDCTFwd_32f`.

`ippsMDCTFwd_32f` is used in the float-point versions of AAC decoder, AAC and MP3 encoders, `ippsMDCTInv_32f` is used in the float-point versions of AC3, AAC, and MP3 decoders, `ippsMDCTFwd_16s_Sfs` is used in the fixed version of AAC encoder included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the specification structure <i>pMDCTSpec</i> is invalid.
<code>ippStsMisalignedBuf</code>	Indicates misaligned data. Supply aligned data for better performance.

**Example 10-8 ippsMDCTFwd\_32f Usage**

```

IppStatus mdct(void)
{
    Ipp32f pSrc[32];
    Ipp32f pDst[16];
    Ipp8u* pBuffer;
    IppsMDCTFwdSpec_32f* pMDCTSpec;
    IppStatus st;
    int size, i;
    for (i = 0; i < 32; i++)
        pSrc[i] = (float)cos(IPP_2PI * i / 32);
    st = ippsMDCTFwdInitAlloc_32f(&pMDCTSpec, 32);
    if (st != ippStsOk) return st;
    st = ippsMDCTFwdGetBufSize_32f(pMDCTSpec, &size);
    if (st != ippStsOk) return st;
    if (size != 0) pBuffer = ippsMalloc_8u(size);
    st = ippsMDCTFwd_32f(pSrc, pDst, pMDCTSpec, pBuffer);
    printf("\n pDst = ");
    for (i = 0; i < 16; i++)
        printf("%f ", pDst[i]);
    printf("\n");
    return st;
}

//Output:
//   pDst = 11.430438 -16.110317 7.151322 4.418863
//           -3.528530 -2.781422 2.448268 2.108223
//           -1.940894 -1.757418 1.664175 1.560623

```

```
//          -1.509103 -1.455249 1.432051 1.415278
```

## Block Filtering Functions

Intel IPP functions described in this section implement the finite impulse response (FIR) block filter. You can use this group of functions to design transform domain adaptive filters. These filters preprocess the signal by decomposing the input vector into orthogonal components, which are subsequently used as inputs to a parallel bank of adaptive subfilters. You may use this approach for implementing frequency domain linear predictors in audio codecs, for example, CELP and AAC.

The filtering function receives a number of vectors (signals). Every call of a filtering function produces one filtered sample for each input signal. The library functions do not perform any particular adaptation method but you can specify the filter taps at each call of a filtering function.

To use the FIR block filter functions, follow these general steps:

- Call `ippsFIRBlockInitAlloc` to initialize the state structure of a block filter.
- Call `ippsFIRBlockOne` to filter a vector of samples through a block filter.
- Call `ippsFIRBlockFree` to free dynamic memory associated with the FIR block filter.

### FIRBlockInitAlloc

*Initializes FIR block filter state.*

---

#### Syntax

```
IppStatus ippsFIRBlockInitAlloc_32f(IppsFIRBlockState_32f** pState, int
order, int len);
```

#### Parameters

<i>pState</i>	Pointer to the FIR block filter state structure to be created.
<i>order</i>	Number of elements in the array containing the tap values.
<i>len</i>	Number of input signals.

#### Description

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockInitAlloc` creates and initializes a FIR block filter state.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the pointers to data arrays are <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.
<code>ippStsFIRLenErr</code>	Indicates an error when <i>order</i> or <i>len</i> is less than or equal to 0.

## FIRBlockFree

*Closes FIR block filter state.*

---

### Syntax

```
IppStatus ippSFIRBlockFree_32f(IppsFIRBlockState_32f* pState);
```

### Parameters

<i>pState</i>	Pointer to the FIR block filter state structure to be closed.
---------------	---

### Description

This function is declared in the `ippac.h` header file. The function `ippsFIRBlockFree` closes the FIR block filter state by freeing all memory associated with the filter state created by the function `ippsFIRBlockInitAlloc`.

Call `ippsFIRBlockFree` after filtering is completed.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pState</i> is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

## FIRBlockOne

*Filters vector of samples through FIR block filter.*

### Syntax

```
IppStatus ippFIRBlockOne_32f(Ipp32f* pSrc, Ipp32f* pDst,
IppsFIRBlockState_32f* pState, Ipp32f* pTaps);
```

### Parameters

<i>pSrc</i>	Pointer to the input vector of samples to be filtered.
<i>pDst</i>	Pointer to the vector of filtered output samples.
<i>pState</i>	Pointer to the FIR filter state structure.
<i>pTaps</i>	Pointer to the vector of filter taps.

### Description

This function is declared in the `ippac.h` header file. The function `ippFIRBlockOne` filters a vector of samples *pSrc* of the length *len* through a filter and stores the result in *pDst*.

The filter taps are specified in the vector *pTaps* of the length *order*. The values of *len* and *order* parameters are specified in the `ippFIRBlockInitAlloc` call.

In the following definition of the FIR filter, the sample of the input vector *i* to be filtered with the delay *k* is denoted  $x_{n-k}^i$ , and the taps are denoted  $h_k$ . The output value  $y_i$  is defined by the following formula:

$$y_n^i = \sum_{k=0}^{\text{order}-1} h_k x_{n-k}^i, \quad 0 \leq i < \text{len}.$$

Before calling the function `ippFIRBlockOne`, initialize the filter state by calling the function `ippFIRBlockInitAlloc`. Specify the taps values in the argument *pTaps*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------



---

<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.
<code>ippStsMisalignedBuf</code>	Indicates misaligned arrays. Supply aligned data for better performance.
<code>ippStsFIRLenErr</code>	Indicates an error when one of the following conditions is true:  <i>pState-&gt;len</i> is less than or equal to 0; <i>pState-&gt;order</i> is less than or equal to 0; <i>pState-&gt;queue_end</i> is less than 0; <i>pState-&gt;queue_end</i> is greater or equal to <i>pState-&gt;order</i> .

## Example 10-9. Single-Rate Filtering with the ippsFIRBlockOne Function

```
IppStatus fir(void)
{
    #undef NUMITERS
    #define NUMITERS 20
    #undef BLOCKSIZE
    #define BLOCKSIZE 20

    int n;
    int i;

    IppStatus status;
    IppsFIRBlockState_32f *fctx;
    Ipp32f x [BLOCKSIZE], y [BLOCKSIZE];
    const float taps[] = {
        0.0051f, 0.0180f, 0.0591f, 0.1245f, 0.1869f, 0.2127f, 0.1869f,
        0.1245f, 0.0591f, 0.0180f, 0.0051f, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
    };

    ippsFIRBlockInitAlloc_32f( &fctx, 11, BLOCKSIZE );
    for (n =0;n<NUMITERS;++n)
    {
        for (i = 0; i< BLOCKSIZE; i++) x[i]=(Ipp32f)sin(IPP_2PI *
            n *0.2 + i);
        status = ippsFIRBlockOne_32f( x, y, fctx, (Ipp32f*)taps );
        for (i = 0; i< BLOCKSIZE; i++)

            printf("%f", y[i]);
    }
    ippsFIRBlockFree_32f(fctx);
    return status;
}
```

## Frequency Domain Prediction Functions

MPEG-2, 4 AAC encoder uses prediction in frequency domain (FDP) to decrease redundancy in the audio signal and ensure more effective coding. For each spectral line, input signals are filtered through the second order adaptive FIR filter called a predictor. Then, instead of processing the signal, the difference between the original signal and the filter output (that is, the prediction error), is passed for further processing. The decoder derives the original signal from the prediction error using a symmetrical block.

You should regularly reset the predictors to their initial state to reduce accumulated calculation error. You may also need to reset the predictors in special cases discussed and described in the ISO-144963 standard. You can reset predictors for the entire spectrum, several scale factor bands, or a selected group of spectral lines.

For more information on the algorithm of filter coefficient adaptation and FDP usage see ISO-144963, clause 6.5.3.2.

To use the FDP prediction tool functions described in this section, follow these steps:

- Call the function `ippsFDPInitAlloc` to allocate memory and initialize predictor state.
- Call the function `ippsFDPFwd` for each frame to calculate prediction error or call the function `ippsFDPInv` to retrieve the original signal.
- Call the functions `ippsFDPReset`, `ippsFDPResetSfb`, or `ippsFDPResetGroup` to reset predictors in certain spectral lines at any time after creating the state.
- Call the function `ippsFDPFree` to free the memory allocated by `ippsFDPInitAlloc`.

### FDPInitAlloc

*Creates and initializes predictor state.*

#### Syntax

```
IppStatus ippsFDPInitAlloc_32f(IppsFDPState_32f** ppFDPState, int len);
```

#### Parameters

<i>ppFDPState</i>	Pointer to pointer to the FDP state structure to be created.
<i>len</i>	Number of spectral lines to be processed.

## Description

This function is declared in the `ippac.h` header file. The function `ippsFDPInitAlloc` creates and initializes the FDP state.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>ppFDPState</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## FDPInit

*Initializes predictor state.*

---

## Syntax

```
IppStatus ippsFDPInit_32f(IppsFDPState_32f** ppFDPState, int len, Ipp8u* pMemSpec);
```

## Parameters

<code>ppFDPState</code>	Pointer to pointer to the FDP state structure to be initialized.
<code>len</code>	Number of spectral lines to be processed.
<code>pMemSpec</code>	Pointer to the area for FDP state structure.

## Description

This function is declared in the `ippac.h` header file. The function `ippsFDPInit` initializes the FDP state.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>ppFDPState</code> or <code>pMemSpec</code> is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## FDPFree

*Closes FDP state.*

---

### Syntax

```
IppStatus ippsFDPFree_32f(IppsFDPState_32f* pFDPState);
```

### Parameters

*pFDPState*                      Pointer to the FDP state structure to be closed.

### Description

This function is declared in the `ippac.h` header file. The function `ippsFDPFree` closes the FDP state by freeing all memory associated with the FDP state structure created by the function `ippsFDPInitAlloc`.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error when the *pFDPState* pointer is `NULL`.  
`ippStsContextMatchErr`        Indicates an error when the state structure is invalid.

## FDPGetSize

*Gets size of FDP state structure.*

---

### Syntax

```
IppStatus ippsFDPGetSize_32f(int len, int* pSizeState);
```

### Parameters

*len*                                Number of spectral lines to be processed.  
*pSizeState*                      Address of size value in bytes of the frequency domain predictor state structure.

### Description

This function is declared in the `ippac.h` header file. The function `ippsFDPGetSize` gets sizes of the frequency domain predictor state structure and stores the results in *pSizeState*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pSizeState</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## FDPReset

*Resets predictors for all spectral lines.*

---

### Syntax

```
IppStatus ippFDPReset_32f(IppsFDPState_32f* pFDPState);
```

### Parameters

`pFDPState` Pointer to the predictor specific state structure.

### Description

This function is declared in the `ippac.h` header file. The function `ippFDPReset` resets predictors for all spectral lines.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pFDPState</code> pointer is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

## FDPResetSfb

*Resets predictor-specific information in some scale factor bands.*

---

### Syntax

```
IppStatus ippFDPResetSfb_32f (const int* pSfbOffset, int sfbNumber, const Ipp8u* pResetFlag, IppsFDPState_32f* pFDPState);
```

## Parameters

<i>pSfbOffset</i>	Pointer to the band offset vector.
<i>sfbNumber</i>	Number of bands.
<i>pResetFlag</i>	Array of flags showing whether predictors for spectral lines in a certain scale factor band need to be reset.
<i>pFDPState</i>	Pointer to the predictor specific state structure.

## Description

This function is declared in the `ippac.h` header file. The function `ippsFDPResetSfb` resets predictors for all spectral lines in each scale factor band *i*, for which `pResetFlag[i]` is not equal to 0.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>sfbNumber</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

## FDPResetGroup

Resets predictors for group of spectral lines.

### Syntax

```
IppStatus ippsFDPResetGroup_32f (int resetGroupName, int step,
IppsFDPState_32f* pFDPState);
```

### Parameters

<i>resetGroupName</i>	Number of the group to be reset.
<i>step</i>	Distance between two neighboring spectral lines in the group.
<i>pFDPState</i>	Pointer to the predictor specific state structure.

## Description

This function is declared in the `ippac.h` header file. The function `ippsFDPResetGroup` resets predictors for each *step*-th spectral line beginning from the start up to the end of spectrum.

Below see code example 10-10 of using `ippsFDPInv` and `ippsFDPResetGroup_32f` functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFDPState</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>resetGroupNumber</i> or <i>step</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

## Example 10-10 ippsFDPInv and ippsFDPResetGroup Usage

```
/* 1 step - initialization */
IppsFDPState_32f* pFDPState;
ippsFDPInitAlloc_32f(&pFDPState, 1024);

/* 2 step - using inside decoder. Something like this (from AAC)*/

ippsFDPInv_32f_I(p_spectrum, sfb_offset_long_window,
                pred_max_sfb, prediction_used,
                pFDPState);

if ((predictor_reset) &&
    predictor_reset_group_number > 0 &&
    predictor_reset_group_number < 31) {
    ippsFDPResetGroup_32f(30, predictor_reset_group_number - 1, pFDPState);
}

/* 3 step - freeing */
ippsFDPFree_32f(pFDPState);
```



## FDPFwd

*Performs frequency domain prediction procedure and calculates prediction error.*

---

### Syntax

```
IppStatus ippsFDPFwd_32f(const Ipp32f* pSrc, Ipp32f* pDst, IppsFDPState_32f* pFDPState);
```

### Parameters

<i>pSrc</i>	Pointer to the input data array.
<i>pDst</i>	Pointer to the data array to be filled with prediction errors.
<i>pFDPState</i>	Pointer to the predictor specific state structure.

### Description

This function is declared in the `ippac.h` header file. The function `ippsFDPFwd` applies frequency domain prediction procedure to the input signal *pSrc* and stores prediction errors in the *pDst* vector.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFDPState</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.
<code>ippStsMisalignedBuf</code>	Indicates misaligned arrays. Supply aligned data for better performance.

## FDPIInv

*Retrieves input signal from prediction error, using frequency domain prediction procedure.*

---

### Syntax

```
IppStatus ippsFDPIInv_32f_I(Ipp32f* pSrcDst, const int* pBandsOffset, int predictorBandsNumber, Ipp8u* pPredictionUsed, IppsFDPState_32f* pFDPState);
```

## Parameters

<i>pSrcDst</i>	Pointer to the input and output data array for the in-place operation.
<i>pDstpBandsOffset</i>	Pointer to the band offset vector.
<i>predictorBandsNumber</i>	Number of scale factor bands.
<i>pPredictionUsed</i>	Pointer to array of flags showing whether prediction will be used in certain scale factor band.
<i>pFDPState</i>	Pointer to the predictor specific state structure.

## Description

This function is declared in the `ippac.h` header file. The function `ippsFDPInv` applies the procedure of frequency domain prediction to specific bands of the input spectral vector *pSrcDst*. Positions of bands are defined by the parameters *predictorBandsNumber* and *pBandsOffset*.

For each scale factor band *i*, if *pPredictionUsed[i]* is not equal to 0, all values of *pSrcDst* within the band are treated as a prediction error and the original signal is restored. If *pPredictionUsed[i]* is equal to 0, all values of *pSrcDst* within the band are treated as a signal and will be passed without any changes.

Regardless of the *pPredictionUsed* flag, the coefficients of each predictor are updated.



**NOTE.** The function operates on the assumption that the end of the last band coincides with the end of the spectral data vector, that is, the size of *pSrcDst* vector is stored in the *pBandsOffset[predictorBandsNumber]* element.

See code [example 10-10](#) of using `ippsFDPInv` function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>predictorBandsNumber</i> is less than or equal to 0.
<code>ippStsContextMatchErr</code>	Indicates an error when the state structure is invalid.

`ippStsMisalignedBuf` Indicates misaligned arrays. Supply aligned data for better performance.

## VLC Functions

Variable length coding (VLC) is an audio data compression method that uses statistical modeling to define which values occur more frequently than others to build tables for subsequent encoding and decoding operations. Audio data in the bitstream is encoded with VLC tables so that the shortest codes correspond to the most frequent values and the longer codes correspond to the less frequent values. Every standard that uses variable length coding has corresponding tables named `inputTable` that list possible codes and their values.

### inputTable Format

```
static IppsVLCTable_32s inputTable[]=  
{  
    {value0, code0, length0};  
    {value1, code1, length1};  
    ...  
    {valueN, codeN, lengthN};  
}
```

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## VLCDecodeEscBlock\_MP3

*Parses the bitstream and decodes variable length code for MP3 using signed VLC tables.*

### Syntax

```
IppStatus ippVLCDecodeEscBlock_MP3_lul6s(const Ipp8u** ppBitStream, int*  
pBitOffset, int linbits, Ipp16s* pData, int len, const IppsVLCDecodeSpec_32s*  
pVLCSpec);
```

## Parameters

<i>ppBitStream</i>	Double pointer to the current byte in the bitstream buffer, is updated by the function.
<i>pBitOffset</i>	Pointer to the bit position in the byte pointed by <i>ppBitStream</i> . <i>pBitOffset</i> is updated by the function.
<i>linbits</i>	Length of escape sequence.
<i>pData</i>	Pointer to the array of decoded values.
<i>len</i>	Number of values to decode into array <i>pData</i> .
<i>pVLCSpec</i>	Pointer to <code>VLCDecoder</code> specification structure.

## Description

The function `ippsVLCDecodeEscBlock_MP3` is declared in the `ippac.h` header file. The function parses the bitstream and decodes variable length code using the `VLCDecoder` specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function `ippsVLCDecodeEscBlock_MP3`, rearrange *inputTable* before initialization as follows:

```
value = (first_sample_value) << 8) + (second_sample_value + 128).
code = (((code_ini << (first_sample_value != 0)) + (first_sample_value < 0))
<< (second_sample_value != 0)) + (first_sample_value < 0).
length = length_ini + (first_sample_value != 0) + (second_sample_value !=
0).
```

MP3Esc VLC codebook is an unsigned two-dimensional codebook with a large absolute value equal to 15.

From one record of the initial *inputTable* you can obtain from one to four records of rearranged *inputTable*. For example:

```
abs(first_sample_value) = 9;
abs(second_sample_value) = 6;

code_ini = 0xf5
length_ini = 12;

record 1: first_sample_value = -9, second_sample_value = -6;
value = (-9 << 8) + (-6 + 128) = -2182;
```

```

code = (((0xf5 << 1) + 1) << 1) + 1 = 0x3d7;
length = 12 + 1 + 1 = 14;
{-2182, 0x3d7, 14)

record 2: first_sample_value = -9, second_sample_value = 6;
value = (-9 << 8) + (6 + 128) = -2170;
code = (((0xf5 << 1) + 1) << 1) + 0 = 0x3d6;
length = 12 + 1 + 1 = 14;
{-2170, 0x3d6, 14)

record 3: first_sample_value = 9, second_sample_value = -6;
value = (9 << 8) + (-6 + 128) = 2426;
code = (((0xf5 << 1) + 0) << 1) + 1 = 0x3d5;
length = 12 + 1 + 1 = 14;
{2426, 0x3d7, 14)

record 4: first_sample_value = 9,
second_sample_value = 6;
value = (9 << 8) + (6 + 128) = 2438;
code = (((0xf5 << 1) + 0) << 1) + 0 = 0x3d4;
length = 12 + 1 + 1 = 14;
{2438, 0x3d7, 14)

```

The function `ippVLCDecodeEscBlock_MP3_1u16s` is used in the float-point and fixed-point versions of MP3 decoder included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBitOffsetErr</code>	Indicates an error when <code>pBitOffset</code> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <code>pVLCSpec</code> structure does not match the operation.
<code>ippStsVLCInputDataErr</code>	Indicates an error when incorrect input is used. For decode functions it can indicate that bitstream contains code that is not specified inside the used table.

## VLCDecodeEscBlock\_AAC

*Parses the bitstream and decodes variable length code for AAC using signed VLC tables.*

---

### Syntax

```
IppStatus ippsVLCDecodeEscBlock_AAC_lu16s(const Ipp8u** ppBitStream, int*
pBitOffset, Ipp16s* pData, int len, const IppsVLCDecodeSpec_32s* pVLCSpec);
```

### Parameters

<i>ppBitStream</i>	Double pointer to the current byte in the bitstream buffer. <i>ppBitStream</i> is updated by the function.
<i>pBitOffset</i>	Pointer to the bit position in the byte pointed by <i>ppBitStream</i> . <i>pBitOffset</i> is updated by the function.
<i>pData</i>	Pointer to the array of decoded values.
<i>len</i>	Number of values to decode into array <i>pData</i> .
<i>pVLCSpec</i>	Pointer to <code>VLCDecoder</code> specification structure.

### Description

The function `ippsVLCDecodeEscBlock_AAC` is declared in the `ippac.h` header file. The function parses the bitstream and decodes variable length code using the `VLCDecoder` specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function `ippsVLCDecodeEscBlock_AAC`, rearrange *inputTable* before initialization as follows:

```
value = (first_sample_value << 8) + (second_sample_value + 128).
code = (((code_ini << (first_sample_value != 0)) + (first_sample_value < 0))
<< (second_sample_value != 0)) + (first_sample_value < 0).
length = length_ini + (first_sample_value != 0) + (second_sample_value !=
0).
```

AACEsc VLC codebook is an unsigned two-dimensional codebook with a large absolute value is equal to 16.

From one record of initial *inputTable* you can obtain from one to four records of rearranged *inputTable*. For example,

```
Initial input table record - {159, 0xlad, 9}
abs(first_sample_value) = 159/17 = 9;
abs(second_sample_value) = 159 - 17*9 = 6;

code_ini = 0xlad
length_ini = 9;

record 1: first_sample_value = -9, second_sample_value = -6;
value = (-9 << 8) + (-6 + 128) = -2182;
code = (((0xlad << 1) + 1) << 1) + 1 = 0x6b7;
length = 9 + 1 + 1 = 11;
{-2182, 0x6b7, 11)

record 2: first_sample_value = -9, second_sample_value = 6;
value = (-9 << 8) + (6 + 128) = -2170;
code = (((0xlad << 1) + 1) << 1) + 0 = 0x6b6;
length = 9 + 1 + 1 = 11;
{-2170, 0x6b6, 11)

record 3: first_sample_value = 9, second_sample_value = -6;
value = (9 << 8) + (-6 + 128) = 2426;
code = (((0xlad << 1) + 0) << 1) + 1 = 0x6b5;
length = 9 + 1 + 1 = 11;
{2426, 0x6b7, 11)

record 4: first_sample_value = 9, second_sample_value = 6;
value = (9 << 8) + (6 + 128) = 2438;
code = (((0xlad << 1) + 0) << 1) + 0 = 0x6b4;
length = 9 + 1 + 1 = 11;
{2438, 0x6b7, 11)
```

The function `ippsVLCDecodeEscBlock_AAC` is used in the float-point and fixed-point versions of AAC decoder included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBitOffsetErr</code>	Indicates an error when <i>pBitOffset</i> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <i>pVLCSpec</i> structure does not match the operation.

ippStsVLCInputDataErr

Indicates an error when incorrect input is used. For decode functions it can indicate that bitstream contains code that is not specified inside the used table.

ippStsVLCACEscCodeLengthErr

Indicates an error when bitstream contains AAC-Esc code with the length more than 21.

## VLCDecodeUTupleEscBlock\_MP3

Parses the bitstream and decodes variable length code for MP3 using unsigned multi-tupled VLC tables.

---

### Syntax

```
IppStatus ippSVLCDecodeUTupleEscBlock_MP3_1u16s(Ipp8u **ppBitStream, int
*pBitOffset, int linbits, Ipp16s *pData, int len, const
IppsVLCDecodeUTupleSpec_32s * pVLCSpec);
```

### Parameters

<i>ppBitStream</i>	Double pointer to the current byte in the bitstream buffer. <i>ppBitStream</i> is updated by the function.
<i>pBitOffset</i>	Pointer to the bit position in the byte pointed by <i>ppBitStream</i> . Valid within the range 0 to 7. <i>pBitOffset</i> is updated by the function.
<i>linbits</i>	Length of escape sequence.
<i>pData</i>	Pointer to the array of decoded values.
<i>len</i>	Number of samples to decode into <i>pData</i> array.
<i>pVLCSpec</i>	Pointer to <code>VLCDecoder</code> specification structure.

### Description

The function `ippSVLCDecodeUTupleEscBlock_MP3` is declared in the `ippac.h` header file. The function parses the bitstream and decodes variable length code using the `VLCDecoder` specification structure built by the function `ippSVLCDecodeUTupleInit_32s` or `ippSVLCDecodeUTupleInitAlloc_32s` from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function `ippSVLCDecodeUTupleEscBlock_MP3`, rearrange *inputTable* before initialization as a 2-tuple table.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBitOffsetErr</code>	Indicates an error when <code>pBitOffset</code> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <code>pVLCSpec</code> structure does not match the operation.
<code>ippStsVLCInputDataErr</code>	Indicates an error when incorrect input is used. For decode functions, it can indicate that the bitstream contains code that is not specified inside the table used.

## VLCDecodeUTupleEscBlock\_AAC

*Parses the bitstream and decodes variable length code for AAC using unsigned multi-tupled VLC tables.*

---

### Syntax

```
IppStatus ippsvLCDecodeUTupleEscBlock_AAC_lu16s(Ipp8u **ppBitStream, int
*pBitOffset, Ipp16s *pData, int len, const IppsVLCDecodeUTupleSpec_32s
*pVLCSpec);
```

### Parameters

<code>ppBitStream</code>	Pointer to pointer to the current byte in the bitstream buffer. <code>ppBitStream</code> is updated by the function.
<code>pBitOffset</code>	Pointer to the bit position in the byte pointed by <code>ppBitStream</code> . Valid within the range 0 to 7. <code>pBitOffset</code> is updated by the function.
<code>pData</code>	Pointer to the array of decoded values.
<code>len</code>	Number of samples to decode into <code>pData</code> array.
<code>pVLCSpec</code>	Pointer to <code>VLCDecoder</code> specification structure.

### Description

The function `ippsVLCDecodeUTupleEscBlock_AAC` is declared in the `ippac.h` header file. The function parses the bitstream and decodes variable length code using the `VLCDecoder` specification structure built by the function `ippsVLCDecodeUTupleInit_32s` or `ippsVLCDecodeUTupleInitAlloc_32s` from data compression domain, and resets the pointers to new positions. The pointer *pBitStream* points to the 8-bit value and the bit offset may vary from one to 32. After processing, the pointer and bit offset are changed and their new values are returned. To use the function `ippsVLCDecodeUTupleEscBlock_AAC`, rearrange *inputTable* before initialization as a 2-tuple table.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBitOffsetErr</code>	Indicates an error when <i>pBitOffset</i> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <i>pVLCSpec</i> structure does not match the operation.
<code>ippStsVLCInputDataErr</code>	Indicates an error when incorrect input is used. For decode functions, it can indicate that the bitstream contains code that is not specified inside the table used.
<code>ippStsVLCAACEscCodeLengthErr</code>	Indicates an error when the bitstream contains AAC-Esc code with the length exceeding 21.

## VLCCountEscBits\_MP3

*Calculates the number of bits necessary for encoding in MP3 format.*

---

### Syntax

```
IppStatus ippsVLCCountEscBits_MP3_16s32s(const Ipp16s* pInputData, int len,
int linbits, Ipp32s* pCountBits, const IppsVLCEncodeSpec_32s* pVLCSpec);
```

### Parameters

*pInputData*                      Pointer to the array of source values.

<i>len</i>	Size of values array <i>pInputData</i> .
<i>linbits</i>	Length of escape sequence.
<i>pCountBits</i>	Pointer to calculated length in bits to encode <i>pInputData</i> .
<i>pVLCSpec</i>	Pointer to <code>VLCDecoder</code> specification structure.

## Description

The function `ippsVLCCountEscBits_MP3` is declared in the `ippac.h` header file. The function calculates the number of bits necessary for encoding source data in *pInputData*, using the `VLCDecoder` specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain. To use the function `ippsVLCCountEscBits_MP3`, rearrange *inputTable* before initialization in the same way as described in the description of `ippsVLCDecodeEscBlock_MP3` with one exception (because the large absolute value is equal to 15):

```
value= (first_sample_value << 5) + (second_sample_value + 15).
```

The function `ippsVLCCountEscBits_MP3_16s32s` is used in the float-point and fixed-point versions of MP3 encoder included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when <i>pVLCSpec</i> structure does not match the operation.

## VLCCountEscBits\_AAC

*Calculates the number of bits necessary for encoding in AAC format.*

---

### Syntax

```
IppStatus ippsVLCCountEscBits_AAC_16s32s(const Ipp16s* pInputData, int len,
Ipp32s* pCountBits, const IppsVLCDecodeSpec_32s* pVLCSpec);
```

### Parameters

<i>pInputData</i>	Pointer to the array of source values.
-------------------	--

<i>len</i>	Size of values array <i>pInputData</i> .
<i>pCountBits</i>	Pointer to calculated length in bits to encode <i>pInputData</i> .
<i>pVLCSpec</i>	Pointer to <code>VLCDecoder</code> specification structure.

## Description

The function `ippsVLCCountEscBits_AAC` is declared in the `ippac.h` header file. The function calculates the number of bits necessary for encoding source data in *pInputData*, using the `VLCDecoder` specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain. To use the function `ippsVLCCountEscBits_AAC`, rearrange *inputTable* before initialization in the same way as described in the description of `ippsVLCDecodeEscBlock_AAC` with one exception (because the large absolute value is equal to 16):

```
value = (first_sample_value << 6) + (second_sample_value + 16).
```

The function `ippsVLCCountEscBits_AAC_16s32s` is used in the float-point and fixed-point versions of AAC encoder included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when <i>pVLCSpec</i> structure does not match the operation.

## VLCDecodeEscBlock\_MP3

*Encodes an array of values into destination bitstream in MP3 format and advances bitstream pointer.*

---

### Syntax

```
IppStatus ippsVLCDecodeEscBlock_MP3_16s1u(const Ipp16s* pInputData, int len,
int linbits, Ipp8u** ppBitStream, int* pBitOffset, const
IppVLCDecodeSpec_32s* pVLCSpec);
```

## Parameters

<i>pInputData</i>	Pointer to the array of source values.
<i>len</i>	Size of values array <i>pInputData</i> .
<i>linbits</i>	Length of escape sequence.
<i>ppBitStream</i>	Pointer to pointer to the current byte in the bitstream buffer. <i>ppBitStream</i> is updated by the function.
<i>pBitOffset</i>	Ppointer to the bit position in the byte pointed by <i>ppBitStream</i> . <i>pBitOffset</i> is updated by the function.
<i>pVLCSpec</i>	Pointer to <code>VLCDecoder</code> specification structure.

## Description

The function `ippsVLCDecodeEscBlock_MP3` is declared in the `ippac.h` header file. The function encodes an array of values from *pInputData* into destination *Bitstream* and advances the bitstream pointer, using the `VLCDecoder` specification structure built by the function `ippsVLCDecodeInit_32s` or `ippsVLCDecodeInitAlloc_32s` from data compression domain. To use the function `ippsVLCDecodeEscBlock_MP3`, rearrange *inputTable* before initialization in the same way as described in the description of [ippsVLCDecodeEscBlock\\_MP3](#) with one exception (because the large absolute value is equal to 15):

```
value = (first_sample_value << 5) + (second_sample_value + 15).
```

The function `ippsVLCDecodeEscBlock_MP3_16s1u` is used in the float-point and fixed-point versions of MP3 encoder included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsBitOffsetErr</code>	Indicates an error when <i>pBitOffset</i> is less than 0 or more than 7.
<code>ippStsContextMatchErr</code>	Indicates an error when <i>pVLCSpec</i> structure does not match the operation.

## VLC Encode EscBlock\_AAC

*Encodes an array of values into destination bitstream in AAC format and advances bitstream pointer.*

---

### Syntax

```
IppStatus ippsVLC EncodeEscBlock_AAC_16s1u(const Ipp16s* pInputData, int len,
Ipp8u** ppBitStream, int* pBitOffset, const IppsVLC EncodeSpec_32s* pVLCSpec);
```

### Parameters

<i>pInputData</i>	Pointer to the array of source values.
<i>len</i>	Size of values array <i>pInputData</i> .
<i>ppBitStream</i>	Double pointer to the current byte in the bitstream buffer. <i>ppBitStream</i> is updated by the function.
<i>pBitOffset</i>	Pointer to the bit position in the byte pointed by <i>ppBitStream</i> . <i>pBitOffset</i> is updated by the function.
<i>pVLCSpec</i>	Pointer to VLC Encoder specification structure.

### Description

The function `ippsVLC EncodeEscBlock_AAC` is declared in the `ippac.h` header file. The function encodes an array of values from *pInputData* into destination *ppBitstream* and advance bitstream pointer, using the VLC Encoder specification structure built by the function `ippsVLC EncodeInit_32s` or `ippsVLC EncodeInitAlloc_32s` from data compression domain. To use the function `ippsVLC EncodeEscBlock_AAC`, rearrange *inputTable* before initialization in the same way as described in the description of `ippsVLC DecodeEscBlock_AAC` with one exception (because the large absolute value is equal to 16):

```
value = (first_sample_value << 6) + (second_sample_value + 16).
```

The function `ippsVLC EncodeEscBlock_AAC_16s1u` is used in the float-point and fixed-point versions of AAC encoder included into IPP Samples. See [introduction](#) to this section.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

`ippStsBitOffsetErr` Indicates an error when `pBitOffset` is less than 0 or more than 7.

`ippStsContextMatchErr` Indicates an error when `pVLCSpec` structure does not match the operation.

## Psychoacoustic Functions

This section describes psychoacoustic functions.

### Spread

*Computes spreading function.*

---

#### Syntax

```
IppStatus ippsSpread_16s_Sfs(Ipp16s* Src1, Ipp16s Src2, int inScaleFactor, Ipp16s* pDst);
```

#### Parameters

<code>Src1</code>	Input data 1.
<code>Src2</code>	Input data 2.
<code>inScaleFactor</code>	Input value scalefactor value.
<code>pDst</code>	Pointer to the output data vector, output data is in Q15 format.

#### Description

The function `ippsSpread` is declared in the `ippac.h` file. This function gives a masking threshold produced by a single tone masker frequency `Src2` (in Bark) for neighboring frequency `Src1` (in Bark). This function is widely used in psychoacoustics.

Below see code example 10-12 of using `ippsSpread_16s_Sfs` function.

The function `ippsSpread_16s_Sfs` is used in the fixed-point version of AAC encoder included into IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when `pDst` pointer is NULL.

## Example 10-12 ippsSpread Usage

```
// Function ippsSpread_16s_Sfs calculates spread(b1,b2) value

// using the following formula:
// tmpx = (b2 >= b1) ? 3*(b2-b1) : 1.5f*(b2-b1);
// tmpz = 8 * ((tmpx-0.5f)*(tmpx-0.5f) - 2*(tmpx-0.5f));
// if (tmpz > 0) tmpz = 0;
// tmpy = 15.811389f + 7.5f*(tmpx + 0.474f)-
// 17.5f*(float)sqrt(1 + (tmpx+0.474f)*(tmpx+0.474f));
//
// spread = (tmpy < -100 ? 0 : (float)pow(10,(float)(tmpz + tmpy)/10));
// The output of ippsSpread_16s_Sfs in Q15 format.
IppStatus spread(void)
{
    Ipp16s b1 = 18668; /* 9.115234 in Q11 */
    Ipp16s b2 = 20255; /* 9.890137 in Q11 */
    Ipp16s dst;
    IppStatus st = ippsSpread_16s_Sfs(b1, b2, -11, &dst);
    printf ("dst = %i\n", dst);
    return st;
}

//Output:
//    dst = 547
```



## Vector Quantization Functions

This sections describes functions used for vector quantization operations.

### VQCodeBookInitAlloc

*Creates and initializes the codebook structure.*

#### Syntax

```
IppStatus ippsVQCodeBookInitAlloc_32f(const Ipp32f* pInputTable,
IppsVQCodeBookState_32f** ppCodeBook, int step, int height);
```

#### Parameters

<i>pInputTable</i>	Pointer to the codebook table of the size <i>step</i> * <i>height</i> containing <i>height</i> quantization vectors of the length <i>step</i> .
<i>ppCodeBook</i>	Doubleo pointer to the codebook state structure.
<i>step</i>	Step to the next line in the table <i>InputTable</i> , quantization vector length.
<i>height</i>	Table height, number of quantization vectors.

#### Description

The function `ippsVQCodeBookInitAlloc` is declared in the `ippac.h` header file. This function allocates memory and initializes the state structure that contains the codebook and additional information needed for the search operation. This structure is used `ippsVQPreliminarySelect`, `ippsVQMainSelect`, `ippsVQIndexSelect`, and `ippsVQReconstruction`.

To free the memory allocated by this function, use the function `ippsVQCodeBookFree`.

#### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>ppCodeBook</i> or <i>pInputTable</i> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.
----------------------------	--

## VQCodeBookInit

*Initializes the codebook structure.*

---

### Syntax

```
IppStatus ippSVQCodeBookInit_32f(const Ipp32f* pInputTable,
IppsVQCodeBookState_32f** ppCodeBook, int step, int height, Ipp8u* pMemSpec);
```

### Parameters

<i>pInputTable</i>	Pointer to the codebook table of the size <i>step</i> * <i>height</i> containing <i>height</i> quantization vectors of the length <i>step</i> .
<i>ppCodeBook</i>	Double pointer to the codebook state structure.
<i>step</i>	Step to the next line in the table <i>InputTable</i> , quantization vector length.
<i>height</i>	Table height, number of quantization vectors.
<i>pMemSpec</i>	Pointer to the area for the codebook structure.

### Description

The function `ippSVQCodeBookInit` is declared in the `ippac.h` header file. This function initializes the structure that contains the codebook and additional information needed for the search operation. This structure is used by the functions [ippSVQPreliminarySelect](#), [ippSVQMainSelect](#), [ippSVQIndexSelect](#), and [ippSVQReconstruction](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pInputTable</i> , <i>ppCodeBook</i> or <i>pMemSpec</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>step</i> or <i>height</i> is less than or equal to 0.

## VQCodeBookFree

*Closes the `IppsVQCodeBookState_32f` structure created by the function `ippsVQCodeBookInitAlloc`.*

---

### Syntax

```
IppStatus ippsVQCodeBookFree_32f(IppsVQCodeBookState_32f* pCodeBook);
```

### Parameters

*pCodeBook*                      Pointer to the codebook state structure.

### Description

This function is declared in the `ippac.h` header file. The function closes the codebook state structure created by the function `ippsVQCodeBookInitAlloc`.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error when *pCodeBook* pointer is NULL.

## VQCodeBookGetSize

*Gets the size of the codebook structure.*

---

### Syntax

```
IppStatus ippsVQCodeBookGetSize_32f(int step, int height, int* pSizeState);
```

### Parameters

*step*                              Step to the next line in the codebook table, quantization vector length.  
*height*                            Table height, number of quantization vectors.  
*pSizeState*                        Address of the codebook structure size value in bytes.

## Description

This function is declared in the `ippac.h` header file. This function calculates the size of the codebook state structure and stores the results in `pSizeState`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSizeState</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when the step or height is less than or equal to 0.

## VQPreliminarySelect

*Selects candidates for the nearest code vector of codebooks.*

---

### Syntax

```
IppStatus ippVQPreliminarySelect_32f(const Ipp32f* pSrc, const Ipp32f*
pWeights, int nDiv, const Ipp32s* pLengths, Ipp32s* pIndx, Ipps32s* pSign,
int nCand, int* pPolbits, IppsVQCodeBookState_32f* pCodeBook);
```

### Parameters

<code>pSrc</code>	Pointer to the source vector.
<code>pWeights</code>	Pointer to the vector of weights.
<code>nDiv</code>	Number of fragmentations of the <code>pSrc</code> and <code>pWeights</code> vectors.
<code>pLengths</code>	Pointer to an array of lengths of fragmentations.
<code>pIndx</code>	Pointer to the output vector of indexes of the <code>nCand</code> minimum candidates.
<code>pSign</code>	Pointer to the output vector of signs of <code>nCand</code> minimum candidates. The value of 1 indicates that the minimal distortion appears when the norm is negative. The value of 0 indicates that the minimal distortion appears when the norm is positive.
<code>nCand</code>	Number of output candidates.
<code>pPolbits</code>	Pointer to the flag vector.

*pCodeBook*

Pointer to the initialized codebook state structure.

### Description

The function `ippsVQPreliminarySelect` is declared in the `ippac.h` header file. This function computes indexes and finds *nCand* vectors with the closest values from the codebook.

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

If the *polbits[idiv]* is greater than `MAXBIT_SHAPE`, the following distortion are also calculated

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

for *idiv* = 0 to *nDiv* - 1, *icb* is the number of the codebook line.

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCodeBook* initialized in the function [ippsVQCodeBookInitAlloc](#).

The function then selects *nCand* candidates of *dist* and *dist<sub>n</sub>* with the minimum distortion measure.

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when <i>pCodeBook</i> or <i>pSrc</i> pointer is NULL.

## VQMainSelect

*Finds optimal indexes with minimal distortion.*

---

### Syntax

```
ippStatus ippsVQMainSelect_32f(const Ipp32f* pSrc, const Ipp32f* pWeights,
int nDiv, const Ipp32s* pLengths, int nCand, Ipp32s** ppIndexCand, Ipps32s**
ppSignCand, Ipp32s** ppIndx, Ipp32s** ppSign, IppsVQCodeBookState_32f**
ppCodeBooks, int nCodeBooks);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pWeights</i>	Pointer to the vector of weights.
<i>nDiv</i>	Number of fragmentations of the <i>pSrc</i> and <i>pWeights</i> vectors.
<i>pLengths</i>	Pointer to an array of lengths of fragmentations.
<i>nCand</i>	Number of input candidates.
<i>ppIndexCand</i>	Double pointer to the input vectors of indexes of <i>nCand</i> minimum candidates.
<i>ppSignCand</i>	Double pointer to the input vectors of signs of <i>nCand</i> minimum candidates.
<i>ppIndx</i>	Double pointer to the output vectors of indexes.
<i>ppSign</i>	Double pointer to the output vectors of signs.
<i>ppCodeBooks</i>	Double pointer to the initialized codebook state structures.

*nCodeBooks*                      Number of codebooks.

### Description

This function is declared in the `ippac.h` header file. The function restores vectors for all possible combinations of indexes across the specified number of codebooks then computes the distortion against the source vector. The function then returns the combination that provides for the minimal distortion.

The following formula serves for computing of the distortion for the given fragmentation *idiv* with the specified quantization vectors *icb[i]*, where *i* is within the range of  $[0, nCodeBooks-1]$ .

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp] - pSrcDiv[ismp])^2$$

Here

$$rec[ismp] = \frac{\sum_{i=0}^{nCodeBooks-1} pSignCand[idiv*nCand+icb[i]]*ppTable[pIndexCand[idiv*nCand+icb[i]]][ismp]}{nCodeBooks}$$

In these formulas

- *pSrcDiv* is a pointer to the beginning of *idiv* fragmentation in the stream computed by the following formula:

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *pWeightDiv* is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- *ppTable* is a double pointer to the table with a set of vectors for quantization, where *icb* is the vector number, *ismp* is the number of element in the given vector. This table is part of the structure *pCodeBook* initialized by the functions [ippsVQCodeBookInitAlloc](#) or [ippsVQCodeBookInit](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## VQIndexSelect

*Finds optimal vector set for specified number of codebooks.*

---

### Syntax

```
IppStatus ippsVQIndexSelect_32f(const Ipp32f* pSrc, const Ipp32f* pWeights,
int nDiv, const Ipp32s* pLengths, int nCand, int** ppPolbits, Ipp32s** ppIndx,
Ipp32s** ppSign, IppsVQCodeBookState_32f** ppCodeBooks, int nCodeBooks);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pWeights</i>	Pointer to the vector of weights.
<i>nDiv</i>	Number of fragmentations of the <i>pSrc</i> and <i>pWeights</i> vectors.
<i>pLengths</i>	Pointer to an array of lengths of fragmentations.
<i>nCand</i>	Number of input candidates.
<i>ppPolbits</i>	Double pointer to the value that indicates whether one or two norms must be used to compute the optimal vector set for the specified number of codebooks.



<i>ppIndx</i>	Double pointer to the output vectors of indexes.
<i>ppSign</i>	Double pointer to the output vectors of signs. The value of 1 indicates that the minimal distortion appears when the norm is negative. The value of 0 indicates that the minimal distortion appears when the norm is positive.
<i>ppCodeBooks</i>	Double pointer to the initialized codebook state structures.
<i>nCodeBooks</i>	Number of codebooks.

### Description

This function is declared in the `ippac.h` header file. The function computes *nCand* vectors for each codebook by the following formula:

$$dist_p[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] - pSrcDiv[ismp])^2$$

If *polbits[ idiv]* is 1, the following distortion is also calculated

$$dist_n[idiv][icb] = \sum_{ismp=0}^{pLengths[idv]-1} pWeightDiv[ismp] \cdot (ppTable[icb][ismp] + pSrcDiv[ismp])^2$$

for *idiv*=0 to *nDiv* -1, *icb* - number of codebook line.

The function subsequently restores vectors for all possible combinations of indexes across the specified number of codebooks then computes the distortion against the source vector. The function then returns the combination that provides for the minimal distortion. The following formula serves for computing of the distortion for the given fragmentation *idiv* with the specified quantization vectors *icb[i]*, where *i* is within the range of  $[0, nCodeBooks - 1]$ .

$$dist_{cross}[idiv] = \sum_{ismp=0}^{pLengths[idiv]-1} pWeightDiv[ismp](rec[ismp] - pSrcDiv[ismp])^2$$

$$rec[ismp] = \frac{\sum_{i=0}^{nCodeBooks-1} pSignCand[idiv*nCand+icb[i]]*ppTable[pIndexCand[idiv*nCand+icb[i]]][ismp]}{nCodeBooks}$$

for  $idiv=0$  to  $nDiv-1$ ,  $icb$  – number of codebook line.

In these formulas

- $pSrcDiv$  is a pointer to the beginning of  $idiv$  fragmentation in the stream computed by the following formula:

$$pSrcDiv = pSrc + \sum_{s=0}^{idiv-1} pLengths[s].$$

- $pWeightDiv$  is a pointer to the weight array for a given fragmentation.

$$pWeightDiv = pWeight + \sum_{s=0}^{idiv-1} pLengths[s].$$

- $ppTable$  is a double pointer to the table with a set of vectors for quantization, where  $icb$  is the vector number,  $ismp$  is the number of element in the given vector. This table is part of the structure  $pCodeBook$  initialized by the functions [ippsVQCodeBookInitAlloc](#) or [ippsVQCodeBookInit](#).

## Return Values

`ippStsNoErr`                      Indicates no error.

`ippStsNullPtrErr` Indicates an error when one of the specified pointers is `NULL`.



**NOTE.** You can use this function instead of calling the functions `ippsVQPreliminarySelect` and `ippsVQMainSelect` to save time and memory cost.

## VQReconstruction

*Reconstructs vectors from indexes.*

### Syntax

```
IppStatus ippsVQReconstruction_32f(const Ipp32s** ppIndx, const Ipp32s**
ppSign, const Ipp32s* pLengths, int nDiv, Ipp32f* pDst,
IppsVQCodeBookState_32f** ppCodeBooks, int nCodeBooks);
```

### Parameters

<i>ppIndx</i>	Double pointer to an array of input vectors of indexes for each codebook.
<i>ppSign</i>	Double pointer to an array of input vectors of signs for each codebook containing either 1 or -1.
<i>pLengths</i>	Pointer to an array of lengths of partitions of output vector.
<i>pDst</i>	Pointer to the reconstructed vector of spectrum values.
<i>nDiv</i>	Number of partitions of output vector.
<i>ppCodeBooks</i>	Double pointer to an array of pointers to the initialized codebook state structures.
<i>nCodeBooks</i>	Number of codebooks.

### Description

This function is declared in the `ippac.h` header file. The function reconstructs a vector divided on several partitions. A partition is defined as set of indexes of vectors from the specified number of codebooks, that is, one index for one partition from each codebook. To reconstruct the values of the output vector, the function calculates the arithmetic mean of vector values from the specified number of codebooks. The arrays *ppIndx* and *ppSign* contain the number of vectors from the codebook and the sign.

$$pDst \left[ \sum_{i=0}^{iDiv-1} pLength[i] + j \right] = \frac{1}{nCodeBooks} \sum_{k=0}^{nCodeBooks-1} pSign[k][i] \cdot pCodeBooks[k] \rightarrow table[pIndx[k][i]][j]$$

for  $i=0$  to  $nDiv-1$ , and for  $j = 0$  to  $pLength[i] - 1$ .

Length of the array  $pDst$  must be equal to

$$\sum_{k=0}^{nCodeBooks-1} pLengths[k]$$

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .

## MP3 Audio Coding Functions

The [ISO/IEC 11172-3](#) MPEG-1, Layer III (also referred to as “MP3”) audio coding algorithms are widely used to compress stereophonic and dual-channel music signals. Well-suited for both transmission and storage applications, the MP3 algorithm delivers high-fidelity audio playback quality with bitrates as low as one-tenth of the original. As a result, the MP3 algorithm has become the *de facto* standard compression methodology for portable and handheld storage media, as well as for transmission of high-fidelity compressed audio over the Internet. MP3 encoder/decoder is widely used in music storage and audio recording.

## Macros and Constants

The MP3 macro and constant definitions are listed in Table 10-2 below.

**Table 10-2. MP3 Macro and Constant Definitions**

Global Macro Name	Definition	Notes
IPP_MP3_GRANULE_LEN	576	The number of samples in one granule
IPP_MP3_V_BUF_LEN	512	V data buffers length (32-bit words)
IPP_MP3_SF_BUF_LEN	40	Scale factor buffer length (8-bit words)
IPP_MP3_SFB_TABLE_LONG_LEN	138	Scale factor band table for long block length (16-bit words)
IPP_MP3_SFB_TABLE_SHORT_LEN	84	Scale factor band table for short block length (16-bit words)
IPP_MP3_ID_MPEG2	0	MPEG-2 frame identifier
IPP_MP3_ID_MPEG1	1	MPEG-1 frame identifier

## Data Structures

The MP3 coding API includes several data structures.

The structure `IppMP3FrameHeader` contains the complete set of header information associated with one frame.

The structure `IppMP3SideInfo` contains the complete set of side information associated with one granule of one channel.

The structure `IppMP3HuffmanTable` contains the complete set of information associated with Huffman table.

The structure `IppMP3PsychoacousticModelTwoAnalysis` contains the outputs generated by the Intel IPP implementation of [ISO/IEC 11172-3](#) psychoacoustic analysis model 2, including estimates of the masked thresholds and perceptual entropy associated with the current frame. Masked thresholds are represented in terms of Mask-to-Signal Ratios (MSRs).

The structure `IppMP3PsychoacousticModelTwoState` contains the state information associated with the Intel IPP implementation of [ISO/IEC 11172-3](#) psychoacoustic analysis model 2 to facilitate coherent block processing.

The structure `IppMP3BitReservoir` contains the state information associated with the quantization bit reservoir.

## Frame Header

```
typedef struct {
    int id;          /* ID: 1 - MPEG-1, 0 - MPEG-2 */
    int layer;       /* layer index: 0x3 - Layer I
                     0x2 - Layer II
                     0x1 - Layer III */
    int protectionBit; /* CRC flag: 0 - CRC on, 1 - CRC off */
    int bitRate;      /* bit rate index */
    int samplingFreq; /* sampling frequency index */
    int paddingBit;   /* padding flag: 0 - no padding, 1 -
                     padding */
    int privateBit;   /* private_bit, not used */
    int mode;         /* mono/stereo selection */
    int modeExt;      /* extension to mode */
    int copyright;    /* copyright or not: 0 - no, 1 - yes */
    int originalCopy; /* original or copied: 0 - copy,
```

```

        1 - original */
    int emphasis;          /* flag indicating the type of de-emphasis */
    int CRCWord;          /* CR-Ccheck word */
} IppMP3FrameHeader;

```

### Side Information

```

typedef struct {
    int part23Len;          /* number of main_data bits */
    int bigVals;           /* half the number of Huffman code words whose maximum amplitudes may be
                           greater than 1 */
    int globGain;          /* quantizes step size information */
    int sfCompress;        /* number of bits used for scale factors */
    int winSwitch;         /* window switch flag */
    int blockType;         /* block type flag */
    int mixedBlock;        /* flag: 0 - non-mixed block, 1 - mixed
                           block */
    int pTableSelect[3];
    /* Huffman table index for the 3 rectangle in <big_values> field */
    int pSubBlkGain[3];
    /* gain offset from the global gain for one subblock */
    int reg0Cnt;           /*the number of scale factor bands in the first region of <big_values>
                           less one */
    int reg1Cnt;           /*the number of scale factor bands in the second region of <big_values>
                           less one */
    int preFlag;           /* flag indicating high frequency boost */
    int sfScale;           /* scale factor scaling */
    int cnt1TabSel;
    /* Huffman table index for the <count1> field of quadruples */
} IppMP3SideInfo;

```

## MP3 Huffman Table Structure

```
typedef struct {
    int tableSize;          /* number of rows in table */
    int linbits;            /* variable used for encode if the magnitude of encoded value is
greater or equal to 15 */
    int maxBitsValue;       /* maximum bit length of codewords */
    Ipp16u *pHcod;          /* pointer to Huffman code table */
    Ipp8u *pSlen;           /* pointer to Huffman length table */
} IppMP3HuffmanTable;
```

## MP3 Psychoacoustic Model Two Analysis

```
typedef struct{
    Ipp32s pMSR[36]; /* MSRs for one granule/channel.

    For long blocks, elements0- 20 represent the thresholds
    associated with the 21 SFBs. For short blocks, elements
    0,3,6,...,33, elements 1,4,...,34, and elements 2,5,...,35,
    respectively, represent the thresholds associated with the
    12 SFBs for each of the 3 consecutive short blocks in one
    granule/channel. That is, the block thresholds are
    interleaved such that the thresholds are grouped by SFB.*/

    Ipp32s PE; /* Estimated perceptual entropy, one
granule/channel */
} IppMP3PsychoacousticModelTwoAnalysis;
```

## Psychoacoustic Model Two State

```
typedef struct {
    Ipp64s pPrevMaskedThesholdLong[2][63];
    /* long block masked
threshold history buffer; contains masked
threshold estimates for the threshold
calculation partitions associated with the
two most recent long blocks */

    Ipp64s pPrevMaskedThesholdShort[IPP_MP3_PSY_BAND_SHORT_NUM];
    /* short block masked threshold history
buffer; contains masked threshold
```



```

    estimates for the threshold calculation
    partitions associated with the two most
    recent short blocks */

Ipp32sc pPrevFFT[2][FIRST_6_CW];    /*FFT history buffer; contains
    real and imaginary FFT components
    associated with the two most recent
    long blocks */

Ipp32s pPrevFFTMag[2][FIRST_6_CW];  /*FFT magnitude history buffer;
    contains FFT component magnitudes
    associated with the two most
    recent long blocks */

int nextPerceptualEntropy;    /*PE estimate for next granule; one
    granule delay provided for synchronization
    with analysis filterbank */

int nextBlockType;    /* Expected block type for next granule; either
    long (normal), short, or stop. Depending upon
    analysis results for the granule following the
    next, a long block could change to a start block,
    and a stop block could change to a short block.
    This buffer provides one granule of delay for
    synchronization with the analysis filterbank */

Ipp32s pNextMSRLong[21];        /*long block MSR estimates for next
    granule. One granule delay provided for
    synchronization with analysis filterbank */

Ipp32s pNextMSRShort[36];      /*short block MSR estimates for next
    granule. One granule delay provided for
    synchronization with analysis filterbank */

} IppMP3PsychoacousticModelTwoState;

```

## MP3 Bit Reservoir

```

typedef struct {

    int BitsRemaining; /*bits currently remaining in the
        reservoir */

    int MaxBits;        /*maximum possible reservoir size, in
        bits, determined as follows:
        min(7680-avg_frame_len, 2^9*8),
        where: avg_frame_len is the average frame
        length (in bits), including padding bits
        and excluding side information bits */

```

```
} IppMP3BitReservoir;
```

## MP3 Codec Enumerated Types

The Intel® IPP MP3 encoder and decoder APIs define enumerated data types that facilitate the synchronization and data transfer between the encoder and decoder components. As shown in Table 10-3 below, the MP3 codec API includes several enumerated types that provide semantic interpretations for frequently used constants.

**Table 10-3. MP3 Enumerated Data Types**

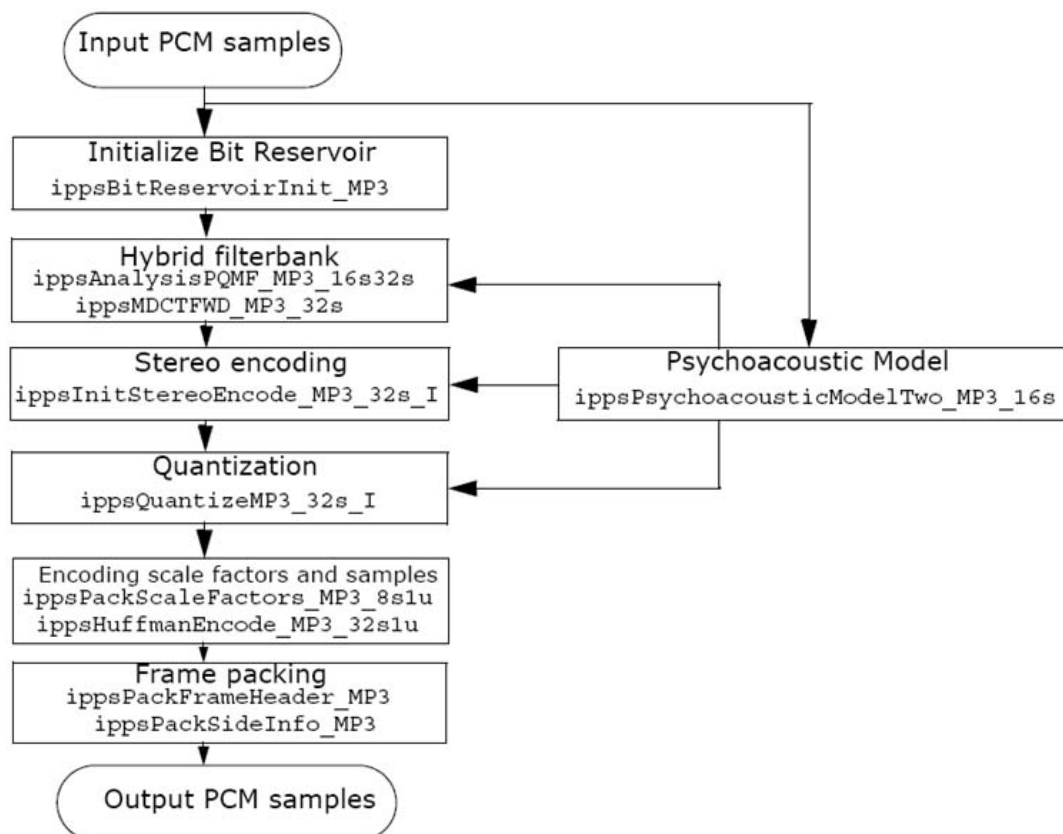
Enumerated Type Name	Symbolic Values	Constant Value
IppMP3BitRate	ippMP3BitRateFree	0
	ippMP3BitRate32	1
	ippMP3BitRate40	2
	ippMP3BitRate48	3
	ippMP3BitRate56	4
	ippMP3BitRate64	5
	ippMP3BitRate80	6
	ippMP3BitRate96	7
	ippMP3BitRate112	8
	ippMP3BitRate128	9
	ippMP3BitRate160	10
	ippMP3BitRate192	11
	ippMP3BitRate224	12
	ippMP3BitRate256	13
	ippMP3BitRate320	14
IppMP3SampleRate	ippMP3SampleRate32000	2
	ippMP3SampleRate44100	0
	ippMP3SampleRate48000	1
IppMP3PcmMode	ippMP3NonInterleavedPCM	1
	ippMP3InterleavedPCM	2
IppMP3ERmpphasis	IppMP3EmphasisNone	0

Enumerated Type Name	Symbolic Values	Constant Value
	IppMP3Emphasis5015	1
	IppMP3EmphasisReserved	2
	IppMP3CCITTJ17	3

MP3 Audio Encoder

The MP3 encoder Application Programming Interface (API) provides a variety of capabilities, including bitstream packing functions and MP3 core encoding functions, See [ISO/IEC 11172-3]. This chapter provides a reference guide to the Intel® Integrated Performnace Primitives (Intel® IPP) MP3 audio encoder API. As shown in Figure 10-1 below, this API includes several functions as well as predefined macros and constants.

Figure 10-1. Intel® IPP MP3 Encoder API Flowchart



## AnalysisPQMF\_MP3

Implements stage 1 of MP3 hybrid analysis filterbank.

### Syntax

```
IppStatus ippsAnalysisPQMF_MP3_16s32s(const Ipp16s* pSrcPcm, Ipp32s* pDstS,
int pcmMode);
```

## Parameters

<i>pSrcPcm</i>	Pointer to the start of the buffer containing the input PCM audio vector. The samples conform to the following guidelines: must be in 16-bit, signed, little-endian, Q15 format most recent 480 (512-32) samples must be contained in the vector $pSrcPcm[pcmMode*i]$ , where $i = 0, 1, \dots, 479$ ; samples associated with the current granule should be contained in the vector $pSrcPcm[pcmMode*j]$ , where $j = 480, 481, \dots, 1055$ .
<i>pcmMode</i>	PCM mode flag. Communicates to PQMF filterbank the type of input PCM vector organization to expect: $pcmMode = 1$ denotes non-interleaved PCM input samples; $pcmMode = 2$ denotes interleaved PCM input samples.
<i>pDstS</i>	Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples under the following index: $pDstXs[32*i + sb]$ , where $i = 0, 1, \dots, 17$ is time series index $sb = 0, 1, \dots, 31$ is the subband index.

## Description

The function is declared in the `ippac.h` file. This function implements the first stage of MP3 hybrid analysis filterbank. The function applies the critically sampled block PQMF analysis bank characterized by the 512-sample prototype window to a PCM input audio vector.

Call the function `ippsAnalysisPQMF_MP3` 18 times per granule on each channel, that is, 36 times per channel on each frame.



**NOTE.** All coefficients are represented using the Q7.24 format

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsErr</code>	Indicates an error when <i>pcmMode</i> exceeds [1,2].

## AnalysisFilterInit\_PQMF\_MP3

*Initializes PQMF MP3 analysis specification structure.*

---

### Syntax

```
IppStatus ippAnalysisFilterInit_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3 **
ppFilterSpec, Ipp8u *pMemSpec);
```

### Parameters

<i>ppFilterSpec</i>	Double pointer to the specification structure.
<i>pMemSpec</i>	Pointer to the area for the specification structure.

### Description

The function is declared in the `ippac.h` file. This function creates and initializes a polyphase quadrature mirror MP3 analysis specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .

## AnalysisFilterInitAlloc\_PQMF\_MP3

*Allocates memory and initializes the PQMF MP3 analysis specification structure.*

---

### Syntax

```
IppStatus ippAnalysisFilterInitAlloc_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3
** ppFilterSpec);
```

### Parameters

<i>ppFilterSpec</i>	Double pointer to the specification structure.
---------------------	--

### Description

The function is declared in the `ippac.h` file. This function allocates memory for polyphase quadrature mirror MP3 analysis specification structure, creates, and initializes this structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>ppFilterSpec</i> pointer is NULL.

## AnalysisFilterGetSize\_PQMF\_MP3

*Returns size of PQMF MP3 analysis specification structure.*

---

### Syntax

```
IppStatus ippAnalysisFilterGetSize_PQMF_MP3_32f(int *pSizeSpec);
```

### Parameters

<i>pSizeSpec</i>	Pointer to the size in bytes of the specification structure.
------------------	--

### Description

The function is declared in the `ippac.h` file. This function calculates the size of the polyphase quadrature mirror MP3 analysis specification structure and stores the result in *pSizeSpec*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSizeSpec</i> is NULL.

## AnalysisFilterFree\_PQMF\_MP3

*Frees memory allocated for PQMF MP3 analysis specification structure.*

---

### Syntax

```
IppStatus ippAnalysisFilterFree_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3 *  
pFilterSpec);
```

### Parameters

<i>pFilterSpec</i>	Pointer to the specification structure.
--------------------	---

## Description

The function is declared in the `ippac.h` file. This function frees memory allocated for the polyphase quadrature mirror MP3 analysis specification structure.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when `pFilterSpec` is NULL.  
`ippStsContextMatchErr` Indicates an error when `pFilterSpec` identifier is incorrect.

## AnalysisFilter\_PQMF\_MP3

*Transforms PQMF MP3-processed subband signals into time domain samples.*

---

## Syntax

```
IppStatus ippAnalysisFilter_PQMF_MP3_32f(const Ipp32f *pSrc, Ipp32f *pDst,
const IppsFilterSpec_PQMF_MP3 *pFilterSpec, int mode);
```

## Parameters

<code>pSrc</code>	Pointer to the source vector, holds time domain input samples.
<code>pDst</code>	Array of pointers, holds PQMF MP3-processed subband signals.
<code>pFilterSpec</code>	Pointer to the initialized specification structure.
<code>mode</code>	Flag that indicates whether or not MP3 audio input channels should be interleaved: 1 - indicates no interleaving, 2 - indicates interleaving.

## Description

The function is declared in the `ippac.h` file. This function performs the polyphase quadrature mirror filter bank, that is, a critically-sampled 32-channel PQMF analysis bank that generates 32-sample output block of IMDCT outputs for each 32 time-domain input sample. For each input block, the PQMF generates an output 32-sample block in the vector pointed to by `pDst`. If `mode` is set to 2, the left and right channel input samples are interleaved, that is, LRLRLR, so that the left channel data is organized as follows: `pDst [2*i]`,  $i = 0$  to 31. If `mode` is set to 1, then the left and right channel inputs are not interleaved.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>mode</i> exceeds [1,2].
<code>ippStsContextMatchErr</code>	Indicates an error when <i>pFilterSpec</i> identifier is incorrect.

## MDCTFwd\_MP3

Implements stage 2 of the MP3 hybrid analysis filterbank.

### Syntax

```
IppStatus ippsMDCTFwd_MP3_32s(const Ipp32s* pSrc, Ipp32s* pDst, int blockType,
int mixedBlock, IppMP3FrameHeader* pFrameHeader, Ipp32s* pOverlapBuf);
```

### Parameters

<i>pSrc</i>	Pointer to the start of the 576-element block PQMF analysis output vector containing 18 consecutive blocks of 32 subband samples that are indexed as follows: <code>pDstS[32* i+ sb]</code> , where <i>i</i> = 0,1,...,17 is time series index, <i>sb</i> = 0,1,...,31 is the subband index. All coefficients are represented using the Q7.24 format.
<i>blockType</i>	Block type indicator: 0 0 stands for normal block 1 stands for start block 2 stands for short block 3 stands for stop block.
<i>mixedBlock</i>	Mixed block indicator:0 stands for not mixed 1 stands for mixed.
<i>pFrameHeader</i>	Pointer to the <code>IppMP3FrameHeader</code> structure that contains the header associated with the current frame. Only MPEG-1 ( <i>id</i> = 1) is supported.
<i>pOverlapBuf</i>	Pointer to the MDCT overlap buffer that contains a copy of the most recent 576-element block of PQMF bank outputs. Prior to processing a new audio stream with the analysis filterbank, all elements of this buffer must be initialized to the constant value 0.

*pDst*

Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank.

## Description

The function is declared in the `ippac.h` file. This function implements the second stage of MP3 hybrid analysis filterbank by performing the following operations:

- **Forward MDCT.** An appropriately arranged set of 12-point and/or 36-point forward Modified Discrete Cosine Transforms (MDCTs) is applied to the 18-sample spectral coefficient blocks generated on each of the 32 PQMF subbands during the first stage analysis.
- **Aliasing reduction butterflies.** The butterflies specified in [ISO/IEC 11172-3](#) are applied to the MDCT outputs in order to mitigate the aliasing artifacts introduced by cascading two critically sampled analysis filterbanks. Each of these introduces some non-negligible amount of interband aliasing.

The function `ippsMDCTFwd_MP3_32s` updates the 576-element MDCT overlap buffer `pMDCTOverlap[ ]`, the contents of which must be preserved between calls to facilitate coherent block processing. The function must be applied once per granule on each channel (that is, applied twice per channel on each frame).




---

**NOTE.** Input coefficients are represented in the Q7.24 format. Output coefficients are represented in the Q5.26 format.

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .

## PsychoacousticModelTwo\_MP3

*Implements ISO/IEC 11172-3 psychoacoustic model recommendation 2 to estimate masked threshold and perceptual entropy associated with a block of PCM audio input.*

### Syntax

```
IppStatus ippsPsychoacousticModelTwo_MP3_16s(const Ipp16s* pSrcPcm,
IppMP3PsychoacousticModelTwoAnalysis* pDstPsyInfo, int* pDstIsSfbBound,
IppMP3SideInfo* pDstSideInfo, IppMP3FrameHeader* pFrameHeader,
IppMP3PsychoacousticModelTwoState* pFramePsyState, Ipp32s* pWorkBuffer, int
pcmMode);
```

### Parameters

*pSrcPcm*

Pointer to the start of the buffer containing the input PCM audio vector, the samples of which must conform to the following format specification: 16-bits per sample, signed, little-endian, Q15. The buffer must contain 1152 samples, that is, two granules of 576 samples each, if the parameter *pFrameHeader -> mode* has the value 1 (mono), or 2304 samples, that is, two granules of 576 samples each, if the parameter *pFrameHeader -> mode* has the value of 2 (stereo, dual mono). In the stereophonic case, the PCM samples associated with the left and right channels should be organized according to the *pcmMode* flag. Failure to satisfy any of the above PCM format and/or buffer requirements results in undefined model outputs.

*pDstPsychoInfo*

Pointer to the first element in a set of PsychoacousticModelTwoAnalysis structures. Each set member contains the MSR and PE estimates for one granule. The number of elements in the set is equal to the number of channels, with the outputs arranged as follows: (Analysis[0] = granule 1, channel 1), (...Analysis[1] = granule 1, channel 2), (...Analysis[2] = granule 2, channel 1), (...Analysis[3] = granule 2, channel 2).

*pDstIsSfbBound*

If intensity coding has been enabled, *pDstIsSfbBound* points to the list of SFB lower bounds above which all spectral coefficients should be processed by the joint stereo intensity coding module. Since the intensity coding SFB lower bound is block-specific, the number of valid elements pointed to by *pDstIsSfbBound* varies depending upon the individual block types associated with each granule. In particular, the list of SFB bounds is indexed as follows: *pIsSfbBound*[3\* *gr*] for long block granules *pIsSfbBound*[3\* *gr* + *w*] for short block granules, where *gr* is the granule index (0 indicates granule 1 and 1 indicates granule 2), and *w* is the block index (0 indicates block 1, 1 indicates block 2, 2 indicates block 3).

For example, given short-block analysis in granule 1 followed by long block analysis in granule 2, the list of SFB bounds would be generated in the following order: *pIsSfbBound*[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}.

Only one SFB lower bound decision is generated for long block granules, whereas three are generated for short block granules. If both MS and intensity coding are enabled, then the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding has been enabled, then the SFB bound represents the lowest non-MS SFB.

*pDstSideInfo*

Pointer to the updated set of *IppMP3SideInfo* structures associated with all granules and channels. The model updates the following fields in all set elements: *blockType*, *winSwitch*, and *mixedBlock*. The number of elements in the set is equal to 2 times the number of channels. Ordering of the set elements is the same as *pDstPsychoInfo*.

---

<i>pFrameHeader</i>	<p>Pointer to the updated <code>IppMP3FrameHeader</code> structure that contains the header associated with the current frame. The model updates the element <code>modeExt</code> to reflect the joint stereo coding mode decision. No other frame header fields are modified by this function.</p>
<i>pFramePsyState</i>	<p>Pointer to the first element in a set of <code>IppMP3PsychoacousticModelTwoState</code> structures that contains the updated psychoacoustic model state information associated with both the current frame and next frame. The number of elements in the set is equal to the number of channels contained in the input audio. That is, a separate analysis is carried for each channel.</p> <p>Prior to encoding a new audio stream, all elements of the psychoacoustic model state structure <code>pPsychoacousticModelState</code> should be initialized to contain the value 0.</p> <p>In the signal processing domain, this could be accomplished using the function <code>ippsZero_16s</code> as follows:</p> <pre>ippsZero_16s ((Ipp16s *) pPsychoacousticModelState, sizeof(IppMP3PsychoacousticModelTwoState)/sizeof(Ipp16s)).</pre>
<i>pWorkBuffer</i>	<p>Pointer to the workspace buffer internally used by the psychoacoustic model for storage of intermediate results and other temporary data. The buffer length must be at least 25,200 bytes, that is, 6300 elements of type <code>Ipp32s</code>.</p>
<i>pcmMode</i>	<p>PCM mode flag. Communicates the psychoacoustic model which type of PCM vector organization to expect: <code>pcmMode = 1</code> denotes non-interleaved PCM input samples, that is, <code>pSrcPcm[0..1151]</code> contains the input samples associated with the left channel, and <code>pSrcPcm[1152..2303]</code> contains the input samples associated with the right channel. <code>pcmMode = 2</code> denotes interleaved PCM input samples, that is, <code>pSrcPcm[2*i]</code> and <code>pSrcPcm[2*i+1]</code> contain the samples associated with the left and right channels, respectively, where <math>i = 0, 1, \dots, 1151</math>.</p>

You can also use appropriately typecast elements `ippMP3NonInterleavedPCM` and `ippMP3InterleavedPCM` of the enumerated type `IppMP3PcmMode` as an alternative to the constants 1 and 2 for `pcmMode`.

## Description

The function is declared in the `ippac.h` file. This function implements the [ISO/IEC 11172-3](#) psychoacoustic model recommendation 2 to estimate the masked threshold and perceptual entropy associated with a block of PCM audio input. Quantization process uses model outputs to estimate a perceptually optimal bit allocation for the spectral coefficients generated by the analysis filterbank. The psychoacoustic model also controls stereophonic MS/intensity mode selection and processing as well as analysis filterbank block size switching. Given one frame of PCM input audio of 1152 samples per channel, that is, two granules of 576 samples each, the psychoacoustic model generates the following outputs:

- **Estimated SFB (scale factor band) Mask-to-Signal ratios (MSRs).** The model generates a vector of estimated MSRs for the 21 SFBs in long block mode and 12 SFBs for each of three consecutive blocks in short block mode. The MSR is derived from the masked threshold, which quantifies the simultaneous masking power associated with one granule/channel (576 samples) of input audio. Given the properties of the audio stimulus presented to the listener, this threshold essentially quantifies the granule-instantaneous modified threshold of hearing. Ideally, the threshold estimate should provide a frequency-dependent intensity (dB SPL) profile beneath which an average listener cannot perceive quantization noise or, for that matter, any other spectral energy. To estimate the masked threshold from a block of input audio, the function `ippsPsych_MP3_16s` implements the procedure recommended in Annex D.2 of [ISO/IEC 11172-3](#). First, the output of a classical FFT-based spectral analysis is grouped into threshold calculation partitions that are organized to achieve analysis with sub-critical bandwidth resolution. On each threshold calculation partition, the model employs a weighted estimate of tone-like or noise-like signal behavior determined by an assessment of a spectral unpredictability across time to estimate masking power in each partition. Second, a spreading function is applied to model the spectral selectivity of the auditory system. Finally, the estimated threshold is compared against the absolute threshold of hearing in quiet and the maximum of the two is assigned to the threshold calculation partition. Ultimately, in order to match its output to the bit allocation scheme of the quantization module, the model converts from the threshold calculation partition scale to a scale factor band (SFB) scale. One set of 21 SFB thresholds is generated for long blocks (576 samples), or three consecutive blocks of 12 SFB thresholds are generated for short blocks (192 samples). To facilitate efficient quantization, the SFB thresholds are inverted and normalized by the signal energy and returned in a vector of SFB Mask-to-Signal ratios (MSRs). The estimated MSRs are returned in the `PsychoacousticModelTwoAnalysis` structure.

- **Estimated perceptual entropy.** The model generates a perceptual entropy (PE) estimate for each granule. The PE quantifies the minimum number of bits required to represent the PCM samples of the granule with “perceptual transparency”. That is, without audible loss of quality for an average listener in comparison to the original, uncoded version. The estimated PE is derived from the masked threshold, in combination with classical assumptions about the minimum number of bits required to achieve a particular signal-to-noise ratio (SNR) target in each SFB, incremental per bit SNR improvement = +6 dB, where the minimum required SNR and hence minimum required number of bits for each SFB is derived from the signal-to-mask ratio (SMR). Perceptual entropy is used to control analysis filterbank block size switching, since sudden large PE increases are often associated with transient audio events that are prone to pre-echo distortion. The PE estimate is returned in the `PsychoacousticModelTwoAnalysis` structure.
- **Analysis filterbank block size decision.** Using perceptual entropy and other indicators, the model determines whether or not the current granule is susceptible to pre-echo distortion. You should prefer using the short block mode when pre-echoes are likely and use long blocks in all other cases. In order to ensure selection of the appropriate block type, the decision incorporates single block look ahead switching logic. For example, if the current block type is long and the next block type is short, the current block type is changed from block type “long/normal” to block type “long/start” in order to guarantee seamless block processing upon mode switch. Similarly, if the current block type has been designated as “long/stop” and the next block type is determined to be “short”, the block switching logic changes the current block from “long/stop” back to “short” in order to avoid unnecessary mode switching. The block type decision is returned in the frame/granule `IppMP3SideInfo` structure.
- **Joint stereophonic processing mode decision.** For 2-channel audio sources, the model evaluates interchannel correlations and other indicators in order to generate joint stereo LR/MS and/or intensity processing mode decisions. The joint stereo mode decision is returned in the `modeExt` field of the `IppMP3FrameHeader` structure.
- **Intensity stereo coding SFB bound decision.** If intensity coding has been activated (see the preceding item joint stereophonic processing mode decision), the psychoacoustic model determines an appropriate lower SFB bound above which all spectral coefficients should be encoded using intensity mode stereophonic processing.

The psychoacoustic model performs analysis on a frame basis (1152 samples per channel), including two granules and up to two channels for either stereophonic or dual mono inputs. Valid lengths for both input and output vectors depend upon which mono or stereo channel modes have been enabled.

## Return Values

`ippStsNoErr`                      Indicates no error.

`ippStsNullPtrErr` Indicates an error when at least one of the pointers `pSrcPcm`, `pDstPsyInfo`, `pDstSideInfo`, `pDstIsSfbBound`, `pFrameHeader`, `pDstPsyState`, or `pWorkBuffer` is NULL.

## JointStereoEncode\_MP3

*Transforms independent left and right channel spectral coefficient vectors into combined mid/side and/or intensity mode coefficient vectors suitable for quantization.*

---

### Syntax

```
IppStatus ippsJointStereoEncode_MP3_32s_I(Ipp32s* pSrcDstXrL, Ipp32s*
pSrcDstXrR, Ipp8s* pDstScaleFactorR, IppMP3FrameHeader* pFrameHeader,
IppMP3SideInfo* pSideInfo, int* pIsSfbBound);
```

### Parameters

<code>pSrcDstXrL</code>	Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank for the left channel of input audio. All coefficients are represented using the Q5.26 format.
<code>pSrcDstXrR</code>	Pointer to the 576-element spectral coefficient output vector generated by the analysis filterbank for the right channel of input audio. All coefficients are represented using the Q5.26 format.
<code>pFrameHeader</code>	Pointer to the <code>IppMP3FrameHeader</code> structure that contains the header information associated with the current frame.
<code>pSideInfo</code>	Pointer to the pair of <code>IppMP3SideInfo</code> structures associated with the channel pair to be jointly encoded. The number of elements in the set is 2, and ordering of the set elements is as follows: <code>pSideInfo[0]</code> describes channel 1, and <code>pSideInfo[1]</code> describes channel 2.



<i>pIsSfbBound</i>	Pointer to the list of intensity coding SFB lower bounds for both channels of the current granule above which all L/R channel spectral coefficients are combined into an intensity-coded representation.
<i>pDstScaleFactorR</i>	Pointer to the vector of scalefactors associated with one granule of the right/S channel.

## Description

The function is declared in the `ippac.h` file. This function transforms the independent left and right channel spectral coefficient vectors into combined mid/side (MS) and/or intensity (IS) mode coefficient vectors suitable for quantization.

Upon function entry, the `IppMP3FrameHeader` structure fields `samplingFreq`, `id`, `mode`, and `modeExt` must contain, respectively, the sample rate associated with the current input audio, the algorithm `id` (MPEG-1 or MPEG-2), and the joint stereo coding commands generated by the psychoacoustic model. All other `pFrameHeader` fields are ignored. Only MPEG-1 (`id = 1`) is supported.

Upon the function entry, the `blockType` side information fields of the structure `IppMP3SideInfo` for both channels must reflect the analysis modes (short or long block) selected by the psychoacoustic model on each channel. All other fields in the `pSideInfo[0]` and `pSideInfo[1]` structures are ignored.

The number of elements of the `pIsSfbBound` depends on the block type associated with the current granule. For short blocks, the SFB bounds are represented in the following order: `pIsSfbBound[0]` describes block 1, `pIsSfbBound[1]` describes block 2, and `pIsSfbBound[2]` describes block 3. For long blocks, only a single SFB lower bound decision is required and is represented in `pIsSfbBound[0]`. If both MS and intensity coding are enabled, the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding is enabled, the SFB bound represents the lowest non-MS SFB.

If intensity coding has been enabled by the psychoacoustic model above a certain SFB lower bound, as indicated by the frame header and the vector pointed to by `pIsSfbBound`, the function `StereoEncode_MP3_32s_I` updates with the appropriate scalefactors those elements of `pDstScaleFactorR[]` that are associated with intensity coded scale factor bands. Other SFB entries in the scale factor vector remain unmodified. The length of the vector referenced by `pDstScaleFactorR` varies as a function of block size. The vector contains 21 elements for long block granules, or 36 elements for short block granules

If MS coding is enabled (`pFrameHeader->modeExt & 0x10 == 1`), the left and right channels are converted to Mid and Side channels as follows:

$$M = \frac{L+R}{\sqrt{2}} \quad S = \frac{L-R}{\sqrt{2}}$$

This function is called on dual granule basis. You must call it for every granule/2 channels.

If intensity coding is enabled (`pFrameHeader->modeExt & 0x01 = 1`), the left channel carries the intensity data for SFBs above the SFB intensity lower bound and the right channel above the SFB lower bound is cleared. All coefficients are set to 0.

$L = L + R, R = 0$

In order to facilitate energy-proportional recovery of the left and right spectral coefficients at the decoder, an intensity energy scale factor, `is_pos`, is transmitted in place of the right channel scale factor since the right channel spectral coefficients above the SFB bound are eliminated. The energy normalization constant is derived from the L/R SFB energy ratios and then transformed to improve its quantization properties. That is,

$$is\_pos = n \cdot \text{int} \left( \frac{12}{\pi} \arctan \left( \sqrt{\frac{L\_energy}{R\_energy}} \right) \right)$$

where `L_energy` and `R_energy` are, respectively, the SFB energies associated with the spectral coefficients of the left and right channels. At the decoder, the `is_pos` scale factor is used to apportion jointly coded signal energy between the left and right channels in a manner consistent with the distribution of signal energy prior to joint coding. The `is_pos` intensity scalefactors are returned in the vector `pDstScalefactorR`.

On each granule, joint stereo coding is applied once per channel pair (576 samples per granule on each channel). The function `JointStereoEncode_MP3_32s_I` must therefore be called once per granule, or twice per frame.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <code>pSrcDstXrL</code> , <code>pSrcDstXrR</code> , <code>pDstScaleFactorR</code> , <code>pFrameHeader</code> , <code>pSideInfo</code> , or <code>pIsSfbBound</code> is NULL.
<code>ippStsMP3SideInfoErr</code>	Indicates an error if <code>pSideInfo[0].blockType != pSideInfo[1].blockType</code> when IS or MS is used.

## Quantize\_MP3

*Quantizes spectral coefficients generated by analysis filterbank.*

---

### Syntax

```
IppStatus ippsQuantize_MP3_32s_I(Ipp32s* pSrcDstXrIx, Ipp8s* pDstScalefactor,
int* pDstScfsi, int* pDstCount1Len, int* pDstHufSize, IppMP3FrameHeader*
pFrameHeader, IppMP3SideInfo* pSideInfo, IppMP3PsychoacousticModelTwoAnalysis*
pPsychoInfo, IppMP3PsychoacousticModelTwoState* pFramePsyState,
IppMP3BitReservoir* pResv, int meanBits, int* pIsSfbBound, Ipp32s*
pWorkBuffer);
```

### Parameters

<i>pSrcDstXrIx</i>	Pointer to the set of unquantized spectral coefficient vectors generated by the analysis filterbank and optionally processed by the joint stereo coding module for one frame.
<i>pDstScaleFactor</i>	Pointer to the output set of scalefactors generated during the quantization process.
<i>pDstScfsi</i>	Pointer to the output vector of scale factor selection information.
<i>pDstCount1Len</i>	Pointer to an output vector of <i>count1</i> region length specifiers.
<i>pDstHufSize</i>	Pointer to an output vector of Huffman coding bit allocation specifiers.
<i>pFrameHeader</i>	Pointer to the <code>IppMP3FrameHeader</code> structure that contains the header information associated with the current frame.
<i>pSideInfo</i>	Pointer to the set of <code>IppMP3SideInfo</code> structures associated with all granules and channels.
<i>pPsychoInfo</i>	Pointer to the first element in a set of <code>PsychoacousticModelTwoAnalysis</code> structures associated with the current frame.

<i>pFramePsyState</i>	Pointer to the first element in a set of <code>IppMP3PsychoacousticModelTwoState</code> structures that contains the psychoacoustic model state information associated with both the current frame and next frame.
<i>pResv</i>	Pointer to the <code>IppMP3BitReservoir</code> structure that contains the bit reservoir state information. Upon the function entry, all structure fields must contain valid data.
<i>meanBits</i>	The number of bits allocated on an average basis for each frame of spectral coefficients and scalefactors given the target bit rate (kilobits per second) specified in the frame header. This number excludes the bits allocated for the frame header and side information.
<i>pIsSfbBound</i>	Pointer to the list of SFB lower bounds above which all L/R channel spectral coefficients have been combined into an intensity-coded representation.
<i>pWorkBuffer</i>	Pointer to the workspace buffer internally used by the quantizer for storage of intermediate results and other temporary data. The buffer length must be at least 2880 bytes, that is, 720 words of 32-bits each.

## Description

The function is declared in the `ippac.h` file. This function quantizes the spectral coefficients generated by the analysis filterbank such that the resulting distortion, that is, quantization noise, is shaped to match a profile derived from the masked thresholds estimated by the psychoacoustic model.

The set of unquantized coefficients *pSrcDstXrIx* must be indexed as follows:

*pSrcDstXrIx*[*gr* \* 1152 + *ch* \* 576 + *i*] for stereophonic and dual-mono input sources;

*pSrcDstXrIx*[*gr* \* 576 + *i*] for monaural, that is, input sources,

where

*i* = 0,1,...,575 is the spectral coefficient index

*gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2

*ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

Depending on which type of joint coding has been applied, if any, the coefficients for each channel could be associated with L/R, M/S, and/or intensity representations of the input audio. All coefficients should be represented using the Q5.26 format.

The scalefactors *pDstScaleFactor* determine the quantizer granularity. Scale factor vector lengths depend on the block mode associated with each granule. The order of the elements is:

1. (granule 1, channel 1)
2. (granule 1, channel 2)
3. (granule 2, channel 1)
4. (granule 2, channel 2).

Given this general organization, the side information for each granule/channel in conjunction with the flags contained in the vector *pDstScfsi* can be used to determine the precise scale factor vector indices and lengths.

The vector *pDstScfsi* contains a set of binary flags that indicate whether or not scalefactors are shared across granules of a frame within predefined scale factor selection groups. For example, bands 0,1,2,3,4,5 form one group; bands 6,7,8,9,10 form a second group, as defined in ISO/IEC 11172-3 . The vector is indexed as follows: *pDstScfsi[ch][scfsi\_band]*, where *ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

*scfsi\_band* is the scale factor selection group number:

group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, group 3 includes SFBs 16-20.

For the purposes of Huffman coding spectral coefficients of a higher frequency than the *bigvals* region, the *count1* parameter indicates the size of the region in which spectral samples can be combined into quadruples for which all elements are of magnitude less than or equal to 1. The vector contains  $2 * nchan$ , elements and is indexed as follows: *pDstCount1Len[gr \* nchan + ch]*, where

*gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.

*nchan* is the number of channels.

*ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

For each granule/channel, the specifiers indicate the total number of Huffman bits required to represent the quantized spectral coefficients in the *bigvals* and *count1* regions. Whenever necessary, each *HufSize* bit count is augmented to include the number of bits required to manage the bit reservoir. For frames in which the reservoir has reached the maximum capacity, the quantizer expends the surplus bits by padding with additional bits the Huffman representation

of the spectral samples. The *HufSize* result returned by the quantizer reflects these padding requirements. That is, *HufSize* [*i*] is equal to the sum of the number of bits required for Huffman symbols and the number of padding bits. The vector contains  $2*nchan$ , elements and is indexed as follows: *pDstHufSize*[*gr*\**nchan*+*ch*], where

*gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.

*nchan* is the number of channels.

*ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

Upon the function entry, the structure *IppMP3FrameHeader* fields *samplingFreq*, *id*, *mode*, and *modeExt* must contain, respectively, the sample rate associated with the current input audio, the algorithm *id*, that is, MPEG-1 or MPEG-2, and the joint stereo coding commands generated by the psychoacoustic model. All other *pFrameHeader* fields are ignored. Only MPEG-1 (*id* = 1) is supported.

The set of *IppMP3SideInfo* structures must contain  $2*nchan$ , elements and must be indexed as follows: *pSideInfo*[*gr*\**nchan* + *ch*], where

*gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2.

*nchan* is the number of channels.

*ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

Upon the function entry, in all set elements the structure fields *blockType*, *mixedBlock*, and *winSwitch* must contain, respectively, the block type indicator (*start*, *short*, or *stop*), filter bank mixed block analysis mode specifier, and window switching flags (*normal* or *blockType*) associated with the current input audio. All other *pSideInfo* fields are ignored upon the function entry and updated upon function exit, as described below under the description of output arguments.

Each member of the set of *PsychoacousticModelTwoAnalysis* structures contains the MSR and PE estimates for one channel of one granule. The set should contain  $2*nchan$ , elements and is indexed as: *pPsychoaInfo*[ *gr*\* *nchan* +*ch*], where

*gr* is the granule index. 0 stands for granule 1, 1 stands for granule 2

*nchan* is the number of channels

*ch* is the channel index. 0 stands for channel 1, 1 stands for channel 2.

The number of elements in the set of `IppMP3PsychoacousticModelTwoState` structures is equal to the number of channels contained in the input audio. That is, a separate analysis is carried for each channel. The quantizer uses the frame type look ahead information `nextBlockType` to manage the bit reservoir. All other structure elements are ignored by the quantizer.

The quantizer uses `meanBits` as a target allocation for the current frame. Given perceptual bit allocation requirements greater than this target, the quantizer makes use of the surplus bits held in the bit reservoir to satisfy frame-instantaneous demands. Similarly, given perceptual bit allocation requirements below this target, the quantizer will store surplus bits in the bit reservoir for use by future frames.

The number of valid elements pointed to by `pIsSfbBound` depends upon the block types associated with the granules of the current frame. In particular, the list of SFB bounds pointed to by `pIsSfbBound` is indexed as follows: `pIsSfbBound[3* gr]` for long block granules, and `pIsSfbBound[3* gr + w]` for short block granules, where

`gr` is the granule index. 0 stands for granule 1, 1 stands for granule 2

`w` is the block index. 0 stands for block 1, 1 stands for block 2, 2 stands for block 3.

For example, given short-block analysis in granule 1 followed by long block analysis in granule 2, the list of SFB bounds would be expected in the following order: `pIsSfbBound[] = {granule 1/block 1, granule 1/block 2, granule 1/block 2, granule 2/long block}`.

If a granule is configured for long block analysis, then only a single SFB lower bound decision is expected, whereas three are expected for short block granules. If both MS and intensity coding have been enabled, then the SFB intensity coding lower bound simultaneously represents the upper bound SFB for MS coding. If only MS coding has been enabled, then the SFB bound represents the lowest non-MS SFB.

Along with meeting perceptual distortion criteria, the quantizer simultaneously adjusts the overall bit allocation to achieve a fixed bit rate target. In accordance with the [ISO/IEC 11172-3](#) recommendation, a bit reservoir is maintained in order to meet instantaneous peak rate demands without violating on an average basis the fixed rate constraint. Surplus bits are deposited in the reservoir during frames with lower than average perceptual bit rate requirements. Supplementary bits are withdrawn from the reservoir to satisfy frames with higher-than-average perceptual bit allocation requirements.

The quantizer manages the reservoir and overall bit allocation such that a constant average rate constraint is satisfied. The quantizer operates on a complete frame of data, that is, two granules and either one or two channels, and it should therefore be called once per frame.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsMP3SideInfo</code>	Indicates an error when <code>pSideInfo-&gt;winSwitch</code> and <code>pSideInfo-&gt;mixedBlock</code> are both defined.
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <code>pFrameHeader-&gt;samplingFreq</code> or <code>pFrameHeader-&gt;id</code> is not correct.

## PackScaleFactors\_MP3

*Applies noiseless coding to scalefactors and packs output into bitstream buffer.*

---

### Syntax

```
IppStatus ippPackScaleFactors_MP3_8slu(const Ipp8s* pSrcScalefactor, Ipp8u**
ppBitStream, int* pOffset, IppMP3FrameHeader* pFrameHeader, IppMP3SideInfo*
pSideInfo, int* pScfsi, int granule, int channel);
```

### Parameters

<code>pSrcScaleFactor</code>	Pointer to a vector of scalefactors generated during the quantization process for one channel of one granule.
<code>ppBitStream</code>	Updated bitstream byte pointer. This parameter points to the first available bitstream buffer byte immediately following the bits generated by the scale factor Huffman encoder and sequentially written into the stream buffer. The scale factor bits are formatted according to the bitstream syntax given in <a href="#">ISO/IEC 11172-3</a> .
<code>pOffset</code>	Updated bitstream bit pointer. The <code>pOffset</code> parameter indexes the next available bit in the next available byte referenced by the updated bitstream buffer byte pointer <code>ppBitStream</code> . This parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit.



<i>pFrameHeader</i>	Pointer to the <code>IppMP3FrameHeader</code> structure for this frame.
<i>pSideInfo</i>	Pointer to the <code>IppMP3SideInfo</code> structure for the current granule and channel.
<i>pScfsi</i>	Pointer to the scale factor selection information table.
<i>granule</i>	Index of the current granule. 0 stands for granule 1, 1 stands for granule 2.
<i>channel</i>	Index of the current channel. 0 stands for channel 1, 1 stands for channel 2.

## Description

The function is declared in the `ippac.h` file. This function applies noiseless coding to the scalefactors and then packs the output into the bitstream buffer. This function operates on one channel of one granule at a time. Therefore, this function must be called for each channel of each granule.

Length of the scale factor vector *pSrcScaleFactor* depends on the block mode. Short block granule scale factor vectors contain 36 elements, or 12 elements for each subblock. Long block granule scale factor vectors contain 21 elements.

Thus short block scale factor vectors are indexed as follows: *pSrcScaleFactor*[*sb* \* 12 + *sfb*], where

- *sb* is the subblock index. 0 stands for subblock 1, 1 stands for subblock 2, 2 stands for subblock 3.
- *sfb* is the scale factor band index (0-11).

Long block scale factor vectors are indexed as follows: *pSrcScaleFactor* [*sfb*], where *sfb* is the scale factor band index (0-20). The associated side information for an individual granule/channel can be used to select the appropriate indexing scheme.

Upon the function entry, the fields *id* and *modeExt* of the *pFrameHeader* must contain, respectively, the algorithm *id* (MPEG-1 or MPEG-2) and the joint stereo coding commands generated by the psychoacoustic model. All other fields of the *pFrameHeader* are ignored. Only MPEG-1 (*id* = 1) is supported.

Upon function entry, the fields *blockType*, *mixedBlock*, and *sfCompress* of the *pSideInfo* must contain, respectively, the block type indicator *start*, *short*, or *stop*, filter bank mixed block analysis mode specifier, and scale factor bit allocation. All other *pSideInfo* fields are ignored by the scale factor encoder.

A scale factor selection information table *pScfsi* contains the set of binary flags that indicate whether scalefactors are shared across granules of a frame within the predefined scale factor selection groups.

For example, bands 0,1,2,3,4,5 form one group and bands 6,7,8,9,10 form a second group (as defined in [ISO/IEC 11172-3](#)). The vector is indexed as follows: *pScfsi[ ch][scfsi\_band ]*, where *ch* is the channel index (0 stands for channel 1, 1 stands for channel 2), and *scfsi\_band* is the scale factor selection group number:

group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, group 3 includes SFBs 16-20.

The resulting bitstream is fully compliant with the syntax specified in [ISO/IEC 11172-3](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsMP3SideInfoErr</code>	Indicates an error when <i>pFrameHeader-&gt;id == IPP_MP3_ID_MPEG1</i> and <i>pSideInfo-&gt;sfCompress</i> exceeds [0..15]; <i>pFrameHeader-&gt;id == IPP_MP3_ID_MPEG2</i> and <i>pSideInfo-&gt;sfCompress</i> exceeds [0..511].
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <i>pFrameHeader-&gt;id == IPP_MP3_ID_MPEG2</i> and <i>pFrameHeader-&gt;modeExt</i> exceeds [0..3].

## HuffmanEncode\_MP3

*Applies lossless Huffman encoding to quantized samples and packs output into bitstream buffer.*

---

### Syntax

```
ippStatus ippHuffmanEncode_MP3_32slu(Ipp32s* pSrcIx, Ipp8u** ppBitStream,
int* pOffset, IppMP3FrameHeader* pFrameHeader, IppMP3SideInfo* pSideInfo,
int count1Len, int hufSize);
```

## Parameters

<i>pSrcIx</i>	Pointer to the quantized samples of a granule. The buffer length is 576. Depending on which type of joint coding has been applied, if any, the coefficient vector might be associated with either the L, R, M, S, and/or intensity channel of the quantized spectral data.
<i>ppBitStream</i>	Updated double pointer to the bitstream byte. It points to the first available bitstream buffer byte immediately following the bits generated by the spectral coefficient Huffman encoder and sequentially written into the stream buffer. The Huffman symbol bits are formatted according to the bitstream syntax given in <a href="#">ISO/IEC 11172-3</a> .
<i>pOffset</i>	Updated bitstream bit pointer. This parameter indexes the next available bit in the next available byte referenced by the updated bitstream buffer byte pointer <i>ppBitStream</i> . This parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit.
<i>pFrameHeader</i>	Pointer to the <code>IppMP3FrameHeader</code> structure for this frame. The Huffman encoder uses the frame header <code>id</code> field in connection with the side information (as described below) to compute the Huffman table region boundaries for the big value spectral region. The Huffman encoder ignores all other frame header fields. Only MPEG-1 ( <code>id = 1</code> ) is supported.
<i>pSideInfo</i>	Pointer to the <code>IppMP3SideInfo</code> structure for the current granule and channel. The structure elements <code>bigVals</code> , <code>pTableSelect[0]-[2]</code> , <code>reg0Cnt</code> , and <code>reg1Cnt</code> are used to control coding of spectral coefficients in the big value region. The structure element <code>cnt1TabSel</code> is used to select the appropriate Huffman table for the (-1,0,+1)-valued 4-tuples in the <code>count1</code> region. For detailed descriptions of all side information elements, see the structure definition header file.

<i>count1Len</i>	The <i>count1</i> region length specifier. Indicates the number of spectral samples for the current granule/channel above the big value region that can be combined into 4-tuples in which all elements are of magnitude less than or equal to 1.
<i>hufSize</i>	<p>Huffman coding bit allocation specifier. Indicates the total number of bits that are required to represent the Huffman-encoded quantized spectral coefficients for the current granule/channel in both the <i>bigVals</i> and <i>count1</i> regions.</p> <p>When necessary, this bit count must be augmented to include the number of bits required to manage the bit reservoir. For frames in which the reservoir has reached maximum capacity, the surplus bits are expended by padding with additional bits the Huffman representation of the spectral samples.</p> <p>The <i>pDstHufSize</i> result returned by the function <i>Quantize_MP3_32s_I</i> reflects these padding requirements. That is, <i>pDstHufSize[i]</i> is equal to the total of the number of bits required for Huffman symbols and the number of padding bits.</p>

## Description

The function is declared in the *ippac.h* file. This function applies lossless Huffman encoding to the quantized samples and packs the output into the bitstream buffer.

The function encodes one granule at a time, and therefore must be called once for each granule of each channel.

The resulting bitstream is fully compliant with [ISO/IEC 11172-3](#).

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when any of the specified pointers is <i>NULL</i> .
<i>ippStsBadArgErr</i>	Indicates an error when <i>pOffset</i> exceeds <i>[0,7]</i> .

`ippStsMP3SideInfoErr` Indicates an error when `pSideInfo->bigVals*2> IPP_MP3_GRANULE_LEN (pSideInfo->reg0Cnt + pSideInfo->reg1Cnt + 2) >= 23`, `pSideInfo->cnt1TabSel` exceeds `[0,1]`, `pSideInfo->pTableSelect[i]` exceeds `[0..31]`.

`ippStsMP3FrameHeader` Indicates an error when `pFrameHeader->id != IPP_MP3_ID_MPEG1`, `pFrameHeader->layer != 1`, `pFrameHeader->samplingFreq` exceeds `[0..2]`.

## PackFrameHeader\_MP3

*Packs the content of the frame header into the bitstream.*

---

### Syntax

```
IppStatus ippsPackFrameHeader_MP3 (IppMP3FrameHeader* pSrcFrameHeader, Ipp8u** ppBitStream);
```

### Parameters

<code>pSrcFrameHeader</code>	Pointer to the <code>IppMP3FrameHeader</code> structure. This structure contains all the header information associated with the current frame. All structure fields must contain valid data upon function entry.
<code>ppBitStream</code>	Updated bitstream byte pointer. This parameter points to the first available bitstream buffer byte immediately following the packed frame header bits. The frame header bits are formatted according to the bitstream syntax given in <a href="#">ISO/IEC 11172-3</a> .

### Description

The function is declared in the `ippac.h` file. This function packs the content of the frame header into the bitstream.

The resulting bitstream is fully compliant with the syntax specified in [ISO/IEC 11172-3](#).

The function should be called once per frame.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when *pSrcFrameHeader* or *ppBitStream* pointer is NULL.

## PackSideInfo\_MP3

*Packs the side information into the bitstream buffer.*

---

### Syntax

```
IppStatus ippsPackSideInfo_MP3(IppMP3SideInfo* pSrcSideInfo, Ipp8u**
ppBitStream, int mainDataBegin, int privateBits, int* pSrcScfsi,
IppMP3FrameHeader* pFrameHeader);
```

### Parameters

<i>pSrcSideInfo</i>	Pointer to the <code>IppMP3SideInfo</code> structures. It must contain twice the channel number of elements. The order is the following: granule 1, channel 1; granule 1, channel 2; granule 2, channel 1; granule 2, channel 2. All fields of all set elements must contain valid data upon the function entry.
<i>ppBitStream</i>	Updated bitstream byte pointer. The parameter <code>ppBitStream</code> points to the first available bitstream buffer byte immediately following the packed side information bits. The frame header bits are formatted according to the bitstream syntax given in ISO/IEC 11172-3 :1993.
<i>mainDataBegin</i>	Negative bitstream offset, in bytes. The parameter value is typically the number of bytes remaining in the bit reservoir before the start of quantization for the current frame. When computing <i>mainDataBegin</i> , you must exclude the header and side information bytes.
<i>privateBits</i>	Channel-dependent number of application-specific (private) bits in the layer III bitstream audio data section.
<i>pSrcScfsi</i>	Pointer to the scale factor selection information table.

*pFrameHeader* Pointer to the `IppMP3FrameHeader` structure. Only MPEG-1 (`id = 1`) is supported.

## Description

The function is declared in the `ippac.h` file. This function packs the side information into the bitstream buffer.

The side information formatter packs the 9-bit value of *mainDataBegin* into the `main_data_begin` field of the output bitstream.

The scale factor selection information table *pSrcScfsi* contains a set of binary flags that indicate whether scalefactors are shared across granules of a frame within predefined scale factor selection groups. For example, bands 0,1,2,3,4,5 form one group and bands 6,7,8,9,10 form the second group, as defined in [ISO/IEC 11172-3](#).

The vector is indexed as *pDstScfsi[ch][scfsi\_band]*, where *ch* is the channel index (0 stands for channel 1, 1 stands for channel 2), *scfsi\_band* is the scale factor selection group number:

group 0 includes SFBs 0-5, group 1 includes SFBs 6-10, group 2 includes SFBs 11-15, group 3 includes SFBs 16-20.

Depending on the number of channels, the function extracts the appropriate number of least significant bits from the parameter *privateBits* and packs them into the `private_bits` field of the output bitstream. The ISO/IEC 11172-3 bitstream syntax reserves a channel-dependent number of application-specific (private) bits in the layer III bitstream audio data section immediately following the parameter `main_data_begin`. For dual- and single-channel streams, respectively, three and five bits are reserved.

Upon the function entry, the structure fields `id`, `mode`, and `layer` of the *pFrameHeader* must contain, respectively, the algorithm `id` (MPEG-1 or MPEG-2), the mono or stereo mode, and the MPEG layer specifier.

All other *pFrameHeader* fields are ignored

The resulting bitstream is fully compliant with the syntax specified in [ISO/IEC 11172-3](#).

This function should be called once per frame.

## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when *pSrcSideInfo*, *ppBitStream*, *pSrcScfsi*, or *pFrameHeader* is NULL.

`ippStatusMP3FrameHeaderErr` Indicates an error when `pFrameHeader->id != IPP_MP3_ID_MPEG1`  
`pFrameHeader->id != IPP_MP3_ID_MPEG2` `pFrameHeader->layer`  
`!= 1` `pFrameHeader->mode` exceeds [0..3].

## BitReservoirInit\_MP3

*Initializes all elements of the bit reservoir state structure.*

---

### Syntax

```
IppStatus ippBitReservoirInit_MP3(IppMP3BitReservoir* pDstBitResv,
IppMP3FrameHeader* pFrameHeader);
```

### Parameters

<i>pFrameHeader</i>	<p>Pointer to the <code>IppMP3FrameHeader</code> structure that contains the header information associated with the current frame. The frame header fields <code>bitRate</code> and <code>id</code>, bit rate index and algorithm identification, respectively, must contain valid data prior to calling the function <code>BitReservoirInit_MP3</code> since both are used to generate the bit reservoir initialization parameters.</p> <p>All other frame header parameters are ignored by the bit reservoir initialization function. Only MPEG-1 (<code>id = 1</code>) is supported.</p>
<i>pDstBitResv</i>	<p>Pointer to the initialized <code>IppMP3BitReservoir</code> state structure. The structure element <code>BitsRemaining</code> is initialized as 0. The structure element <code>MaxBits</code> is initialized to reflect the maximum number of bits that can be contained in the reservoir at the start of any given frame. The appropriate value of <code>MaxBits</code> is directly determined by the selected algorithm (MPEG-1 or MPEG-2) and the stream bit rate indicated by the field <code>pFrameHeader.bitRate</code>.</p>



### Description

The function is declared in the `ippac.h` file. This function initializes all elements of the bit reservoir state structure based on the coding algorithm (MPEG-1 or MPEG-2) and the average per frame bit allocation specified in the frame header.

### Return Values

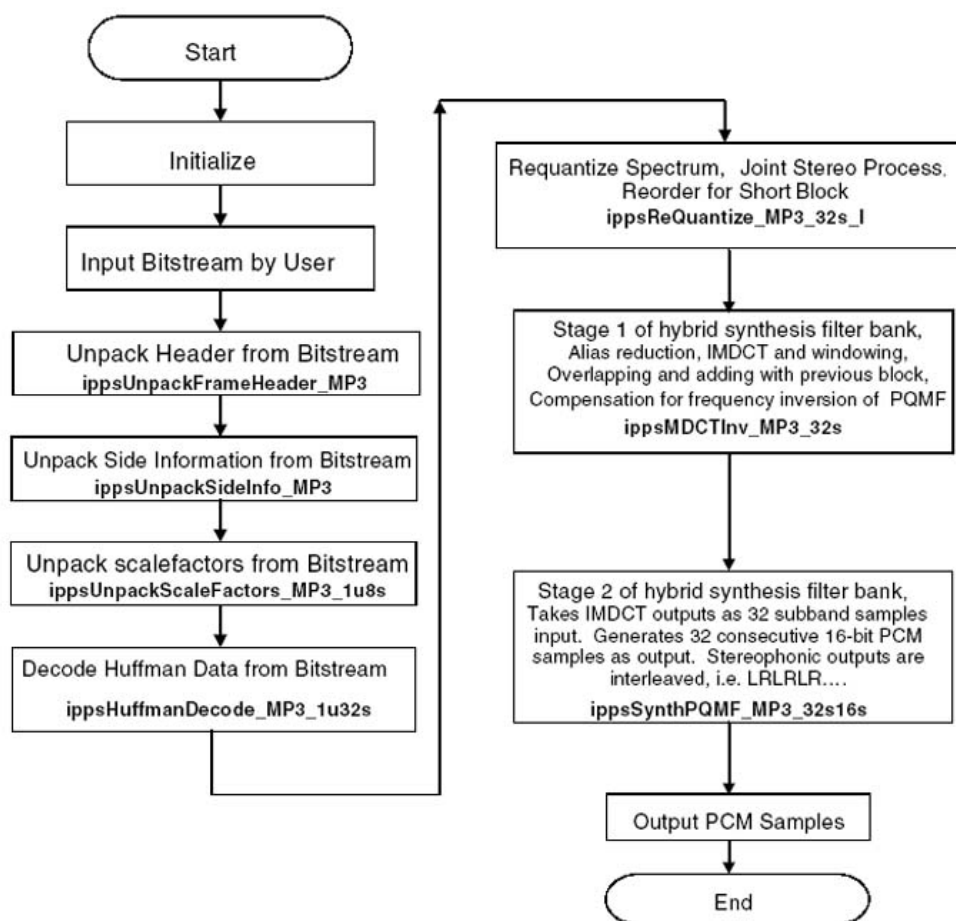
<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pDstBitResv</code> or <code>pFrameHeader</code> is <code>NULL</code> .
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <code>pFrameHeader-&gt;id != IPP_MP3_ID_MPEG1</code> .

## MP3 Audio Decoder

This section describes Intel IPP functions for assembling audio decoders compliant with the layer III portions of the [ISO/IEC 11172-3](#) MPEG-1 and ISO/IEC [ISO/IEC 13818](#) MPEG-2 audio specifications (see these specifications and provides a reference guide to the Intel IPP MP3 Application Programming Interface (API). As shown in Figure 10-2 below, the MP3 API consists of seven functions, two data structures, and several pre-defined constants and macros.

### Figure 10-2 MP3 Decoder API

---



## UnpackFrameHeader MP3

Unpacks audio frame header.

### Syntax

```
IppStatus ippsUnpackFrameHeader_MP3(Ipp8u** ppBitStream, IppMP3FrameHeader*
pFrameHeader);
```

## Parameters

<i>ppBitStream</i>	Pointer to the pointer to the first byte of the MP3 frame header; it is updated in the function.
<i>pFrameHeader</i>	Pointer to the MP3 frame header structure.

## Description

The function `ippsUnpackFrameHeader_MP3` is declared in the `ippac.h` file. This function unpacks the audio frame header. If cyclic redundancy check (CRC) is enabled, this function also unpacks the CRC word.

Before calling `ippsUnpackFrameHeader_MP3`, you must locate the bit stream synchronization word and ensure that *ppBitStream* points to the first byte of the 32-bit frame header.

If CRC is enabled, the assumption is that the 16-bit CRC word is adjacent to the 32-bit frame header, as defined in the MP3 standard. Before returning to the caller, the function updates the pointer *ppBitStream* so that it references the next byte after the frame header or the CRC word.

The first byte of the 16-bit CRC word is stored in *pFrameHeader* ->CRCWord(15:8), and the second byte is stored in *pFrameHeader* ->CRCWord(7:0).

The function does not detect corrupted frame headers.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>ppBitStream</i> or <i>FrameHeader</i> is NULL.

## UnpackSideInfo\_MP3

*Unpacks side information from input bitstream for use during decoding of associated frame.*

---

## Syntax

```
IppStatus ippsUnpackSideInfo_MP3(Ipp8u** ppBitStream, IppMP3SideInfo*
pDstSideInfo, int* pDstMainDataBegin, int* pDstPrivateBits, int* pDstScfsi,
IppMP3FrameHeader* pFrameHeader);
```

## Parameters

<i>ppBitStream</i>	Pointer to the pointer to the first byte of the side information associated with the current frame in the bit stream buffer. The function updates this parameter.
<i>pFrameHeader</i>	Pointer to the structure that contains the unpacked MP3 frame header. The header structure provides format information about the input bitstream. Both single- and dual-channel MPEG-1 and MPEG-2 modes are supported.
<i>pDstSideInfo</i>	Pointer to the MP3 side information structure. The structure contains side information that applies to all granules and all channels for the current frame. One or more of the structures are placed contiguously in the buffer pointed to by <i>pDstSideInfo</i> in the following order: {granule 0 (channel 0, channel 1), granule 1 (channel 0, channel 1)}.
<i>pDstMainDataBegin</i>	Pointer to the <code>main_data_begin</code> field.
<i>pDstPrivateBits</i>	Pointer to the <code>private bits</code> field.
<i>pDstScfsi</i>	Pointer to the buffer containing the scale factor selection information associated with the current frame. The data is organized contiguously in the buffer in the following order: {channel 0 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3), channel 1 (scfsi_band 0, scfsi_band 1, ..., scfsi_band 3)}.

## Description

The function `ippsUnpackSideInfo_MP3` is declared in the `ippac.h` file. This function unpacks the side information from the input bitstream. Before calling the function `ippsUnpackSideInfo_MP3`, make sure that the pointer *ppBitStream* points to the first byte of the bit stream that contains the side information associated with the current frame. Before returning to the caller, the function updates the pointer *ppBitStream* so that it references the next byte after the side information.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

- `ippStsNullPtrErr` Indicates an error when at least one of the pointers `ppBitStream`, `pDstSideInfo`, `pDstMainDataBegin`, `pDstPrivateBits`, `pDstScfsi`, `pFrameHeader`, or `ppBitStream` is NULL.
- `ippStsMP3FrameHeaderErr` Indicates an error when some elements in the MP3 frame header structure are invalid: `pFrameHeader->id != IPP_MP3_ID_MPEG1`, `pFrameHeader->id != IPP_MP3_ID_MPEG2`, `pFrameHeader->layer != 1`, `pFrameHeader->mode` exceeds [0..3].
- `ippStsMP3SideInfoErr` Indicates an error when the value of `block_type` is zero when `window_switching_flag` is set.

## UnpackScaleFactors\_MP3

*Unpacks scalefactors.*

---

### Syntax

```
IppStatus ippUnpackScaleFactors_MP3_lu8s(Ipp8u** ppBitStream, int* pOffset,
Ipp8s* pDstScaleFactor, IppMP3SideInfo* pSideInfo, int* pScfsi,
IppMP3FrameHeader* pFrameHeader, int granule, int channel);
```

### Parameters

- |                          |   |
|--------------------------|---|
| <code>ppBitStream</code> | Pointer to the pointer to the first bitstream buffer byte associated with the scale factors for the current frame, granule, and channel. The function updates this parameter.   |
| <code>pOffset</code>     | Pointer to the next bit in the byte referenced by <code>ppBitStream</code> . Valid within the range of 0 to 7, where 0 corresponds to the most significant bit and 7 corresponds to the least significant bit. The function updates this parameter. |
| <code>pSideInfo</code>   | Pointer to the MP3 side information structure associated with the current granule and channel.  |
| <code>pScfsi</code>      | Pointer to scale factor selection information for the current channel.  |
| <code>channel</code>     | Channel index. Can take the values of either 0 or 1.  |
| <code>granule</code>     | Granule index. Can take the values of either 0 or 1.  |

<i>pFrameHeader</i>	Pointer to MP3 frame header structure for the current frame.
<i>pDstScaleFactor</i>	Pointer to the scalefactor vector for long and/or short blocks.

## Description

The function `ippsUnpackScaleFactors_MP3` is declared in the `ippac.h` file. This function unpacks short and/or long block scalefactors for one granule of one channel and places the results in the vector *pDstScaleFactor*. Before returning to the caller, the function updates *ppBitStream* and *pOffset* so that they point to the next available bit in the input bitstream.



**NOTE.** MPEG-2 intensity mode: if the intensity position is equal to the maximum value, or illegal position, the illegal position sets to negative. Consequently, in the requantization module, negative positions indicate illegal positions. Scalefactors that are not treated as intensity positions must be set to positive before using them.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>pDstScaleFactor</i> , <i>pSideInfo</i> , <i>pScfsi</i> , <i>ppBitStream</i> , or <i>pFrameHeader</i> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>pOffset</i> > 7 , or <i>pOffset</i> < 0 , or granule and/or channel exceeds [0, 1].
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <i>pFrameHeader</i> ->id != IPP_MP3_ID_MPEG1 <i>pFrameHeader</i> ->id != IPP_MP3_ID_MPEG2 <i>pFrameHeader</i> ->mode exceeds [0..3].
<code>ippStsMP3SideInfoErr</code>	Indicates an error when <i>pSideInfo</i> ->blockType exceeds [0, 3] , or <i>pSideInfo</i> ->mixedBlock exceeds [0, 1], or <i>pSideInfo</i> ->sfCompress exceeds [0, 15]; or <i>pScfsi</i> [0..3] exceeds [0, 1] when <i>pFrameHeader</i> indicates the bitstream is MPEG-1, or <i>pSideInfo</i> -> sfCompress exceeds [0, 511] when <i>pFrameHeader</i> indicates the bitstream is MPEG-2.

## HuffmanDecode\_MP3, HuffmanDecodeSfb\_MP3, HuffmanDecodeSfbMbp\_MP3

*Decodes Huffman data.*

### Syntax

```
IppStatus ippsHuffmanDecode_MP3_lu32s(Ipp8u** ppBitStream, int* pOffset,
Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,
IppMP3FrameHeader* pFrameHeader, int hufSize);

IppStatus ippsHuffmanDecodeSfb_MP3_lu32s(Ipp8u** ppBitStream, int* pOffset,
Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,
IppMP3FrameHeader* pFrameHeader, int hufSize, IppMP3ScaleFactorBandTableLong*
pSfbTableLong);

IppStatus, ippsHuffmanDecodeSfbMbp_MP3_lu32s(Ipp8u** ppBitStream, int*
pOffset, Ipp32s* pDstIs, int* pDstNonZeroBound, IppMP3SideInfo* pSideInfo,
IppMP3FrameHeader* pFrameHeader, int hufSize, IppMP3ScaleFactorBandTableLong*
pSfbTableLong, IppMP3ScaleFactorBandTableShort pSfbTableShort,
IppMP3MixedBlockPartitionTable pMbpTable);
```

### Parameters

<i>ppBitStream</i>	Pointer to the pointer to the first bit stream byte that contains the Huffman code words associated with the current granule and channel. The function updates this parameter.
<i>pOffset</i>	Pointer to the starting bit position in the bit stream byte pointed to by <i>ppBitStream</i> . The parameter is valid within the range of 0 to 7, where 0 corresponds to the most significant bit, and 7 corresponds to the least significant bit. The function updates this parameter.
<i>pSideInfo</i>	Pointer to MP3 structure containing side information associated with the current granule and channel.
<i>pFrameHeader</i>	Pointer to MP3 structure containing the header associated with the current frame.

<i>pDstIs</i>	Pointer to the vector of decoded Huffman symbols used to compute the quantized values of the 576 spectral coefficients associated with the current granule and channel.
<i>pDstNonZeroBound</i>	Pointer to the spectral region above which all coefficients are set to zero.
<i>hufSize</i>	The number of Huffman code bits associated with the current granule and channel.
<i>pSfbTableLong</i>	Pointer to the scale factor bands table for a long block.
<i>pSfbTableShort</i>	Pointer to scale factor band table for short block. You can use the default table from MPEG-1, MPEG-2 standards.
<i>pMbpTable</i>	Pointer to the mixed block partition table.

## Description

The functions `ippsHuffmanDecode_MP3`, `ippsHuffmanDecodeSfb_MP3`, and `ippsHuffmanDecodeSfbMbp_MP3` are declared in the `ippac.h` file. These functions decode Huffman symbols for the 576 spectral coefficients associated with one granule of one channel. Before returning to the caller, the function updates *ppBitStream* so that it points to the byte in the bitstream that contains the first new bit following the decoded block of Huffman codes.

The function `ippsHuffmanDecodeSfb_MP3` makes the use of a predefined scale factor bands table possible when working with a long block. Alternatively, you can use the default table from MPEG-1, or MPEG-2 standards.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>pDstIs</i> , <i>pDstNonZeroBound</i> , <i>pSideInfo</i> , <i>pFrameHeader</i> , or <i>ppBitStream</i> is NULL or when <i>pOffset</i> < 0 or <i>pOffset</i> > 7.
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when some elements in the MP3 frame header structure are invalid.
<code>ippStsMP3SideInfoErr</code>	Indicates an error when some elements in the MP3 side information structure are invalid, or when <i>hufSize</i> < 0 or <i>hufSize</i> > <i>pSideInfo</i> -> <i>part23Len</i> .
<code>ippStsErr</code>	Indicates unknown error.



## ReQuantize\_MP3, ReQuantizeSfb\_MP3

*Requantizes the decoded Huffman symbols.*

### Syntax

```
IppStatus ippsReQuantize_MP3_32s_I(Ipp32s* pSrcDstIsXr, int* pNonZeroBound,
Ipp8s* pScaleFactor, IppMP3SideInfo* pSideInfo, IppMP3FrameHeader*
pFrameHeader, Ipp32s* pBuffer);

IppStatus ippsReQuantizeSfb_MP3_32s_I(Ipp32s* pSrcDstIsXr, int* pNonZeroBound,
Ipp8s* pScaleFactor, IppMP3SideInfo* pSideInfo, IppMP3FrameHeader*
pFrameHeader, Ipp32s* pBuffer, IppMP3ScaleFactorBandTableLong pSfbTableLong,
IppMP3ScaleFactorBandTableShort pSfbTableShort);
```

### Parameters

<i>pSrcDstIsXr</i>	Pointer to the vector of the decoded Huffman symbols. For stereo and dual channel modes, right channel data begins at the address <code>&amp;(pSrcDstIsXr [576])</code> . The function updates this vector.
<i>pNonZeroBound</i>	Pointer to the spectral bound above which all coefficients are set to zero. For stereo and dual channel modes, the left channel bound is <code>pNonZeroBound [0]</code> , and the right channel bound is <code>pNonZeroBound [1]</code> .
<i>pScaleFactor</i>	Pointer to the scalefactor buffer. For stereo and dual channel modes, the right channel scalefactors begin at <code>&amp;(pScaleFactor [IPP_MP3_SF_BUF_LEN])</code> .
<i>pSideInfo</i>	Pointer to the side information for the current granule.
<i>pFrameHeader</i>	Pointer to the frame header for the current frame.
<i>pBuffer</i>	Pointer to the workspace buffer. The buffer length must be 576 samples.
<i>pSfbTableLong</i>	Pointer to the scale factor bands table for a long block.
<i>pSfbTableShort</i>	Pointer to the scale factor bands table for a short block.

## Description

The functions `ippsReQuantize_MP3` and `ippsReQuantizeSfb_MP3` are declared in the `ippac.h` file. These functions requantize the decoded Huffman symbols. Spectral samples for the synthesis filter bank are derived from the decoded symbols using the requantization equations given in the ISO standard.

If necessary, you can apply stereophonic mid/side (M/S) and/or intensity decoding. The function returns requantized spectral samples in the vector `pSrcDstIsXr`.

For short blocks you can perform reordering operation. You should preallocate a workspace buffer pointed to by `pBuffer` prior to calling the requantization function. For short blocks, the value pointed to by `pNonZeroBound` is recalculated according to the reordered sequence.

You can use predefined scale factor bands table for a long or short block with the function `ippsReQuantizeSfb_MP3`. Alternatively, you can use the default table from MPEG-1, or MPEG-2 standards.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <code>pNonZeroBound</code> exceeds <code>[0, 576]</code> .
<code>ippStsMP3FrameHeaderErr</code>	Indicates an error when <code>pFrameHeader-&gt;id != IPP_MP3_ID_MPEG1</code> <code>pFrameHeader-&gt;id != IPP_MP3_ID_MPEG2</code> <code>pFrameHeader-&gt;samplingFreq</code> exceeds <code>[0, 2]</code> , <code>pFrameHeader-&gt;mode</code> exceeds <code>[0, 3]</code> , <code>pFrameHeader-&gt;modeExt</code> exceeds <code>[0, 3]</code> .
<code>ippStsMP3SideInfoErr</code>	Indicates an error when the bitstream is in the stereo mode, but the block type of left is different from that of right, <code>pSideInfo[ch].blockType</code> exceeds <code>[0, 3]</code> , <code>pSideInfo[ch].mixedBlock</code> exceeds <code>[0, 1]</code> , <code>pSideInfo[ch].globGain</code> exceeds <code>[0, 255]</code> , <code>pSideInfo[ch].sfScale</code> exceeds <code>[0, 1]</code> , <code>pSideInfo[ch].preFlag</code> exceeds <code>[0, 1]</code> , <code>pSideInfo [ch].pSubBlkGain[w]</code> exceeds <code>[0,</code> <code>7]</code> , where <code>ch</code> is within the range of 0 to 1, and <code>w</code> is within the range of 0 to 2.
<code>ippStsErr</code>	Indicates an unknown error.

## MDCTInv\_MP3

*Performs the first stage of hybrid synthesis filter bank.*

---

### Syntax

```
IppStatus ippMDCTInv_MP3_32s(Ipp32s* pSrcXr, Ipp32s* pDstY, Ipp32s*
pSrcDstOverlapAdd, int nonZeroBound, int* pPrevNumOfImdct, int blockType,
int mixedBlock);
```

### Parameters

<i>pSrcXr</i>	Pointer to the vector of requantized spectral samples for the current channel and granule, represented in Q5.26 format.
<i>pDstY</i>	Pointer to the vector of IMDCT outputs in Q7.24 format for input to PQMF bank.
<i>pSrcDstOverlapAdd</i>	Pointer to the overlap-add buffer. Contains the overlapped portion of the previous granule IMDCT output in Q7.24 format. The function updates this buffer.
<i>nonZeroBound</i>	Limiting bound for spectral coefficients. All spectral coefficients exceeding this boundary become zero for the current granule and channel.
<i>pPrevNumOfImdct</i>	Pointer to the number of IMDCTs computed for the current channel of the previous granule. The function updates this parameter so that it references the number of IMDCTs for the current granule.
<i>blockType</i>	Block type indicator.
<i>mixedBlock</i>	Mixed block indicator.

### Description

The function `ippMDCTInv_MP3` is declared in the `ippac.h` file. This function performs the first stage of the hybrid synthesis filter bank. The following operations are performed:

- alias reduction
- inverse modified discrete cosine transform (IMDCT) according to block size specifiers and mixed block modes

- overlap add of IMDCT outputs
- frequency inversion prior to pseudo-quadrature mirror synthesis filter (PQMF) bank.

Since IMDCT is a lapped transform, you should preallocate a buffer referenced by *pSrcDstOverlapAdd* to maintain the IMDCT overlap-add state.

The buffer must contain 576 elements. Prior to the first call to the synthesis filter bank, all elements of the overlap-add buffer should be set to zero. In between all subsequent calls, the MP3 application must preserve the contents of the overlap-add buffer.

Upon entry to *ippsMDCTInv\_MP3\_32s*, the overlap-add buffer must contain the IMDCT output generated by operating on the previous granule. Upon exit from *ippsMDCTInv\_MP3\_32s*, the overlap-add buffer contains the overlapped portion of the output generated by operating on the current granule.

Upon return from the function, the IMDCT sub-band output samples are organized as follows:

*pDstY[j\*32+subband]*, for *j* = 0 to 17, *subband* = 0 to 31.

Q5.26 designates that 5 bits before and 26 bits after fixed point position are used to present a 32-bit value in the fixed point format.

Q7.24 designates that 7 bits before and 24 bits after fixed point position are used to present a 32-bit value in the fixed point format.




---

**NOTE.** Note that the pointers *pSrcXr* and *pDstY* must reference different buffers.

---

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsBadArgErr</i>	Indicates an error condition if any of the specified pointers is NULL.
<i>ippStsErr</i>	Indicates an error when one or more of the following input data errors are detected: either <i>blockType</i> exceeds [0,3], or <i>mixedBlock</i> exceeds [0,1], or <i>nonZeroBound</i> exceeds [0,576], or <i>pPrevNumOfImdct</i> exceeds [0,32].

## SynthPQMF\_MP3

*Performs the second stage of hybrid synthesis filter bank.*

---

### Syntax

```
IppStatus ippsSynthPQMF_MP3_32s16s(Ipp32s* pSrcY, Ipp16s* pDstAudioOut,
Ipp32s* pVBuffer, int* pVPosition, int mode);
```

### Parameters

<i>pSrcY</i>	Pointer to the block of 32 IMDCT sub-band input samples in Q7.24 format.
<i>pDstAudioOut</i>	Pointer to the block of 32 reconstructed PCM output samples in 16-bit signed little-endian format. Left and right channels are interleaved according to the <i>mode</i> flag.
<i>pVBuffer</i>	Pointer to the input workspace buffer containing Q7.24 data. The function updates this parameter.
<i>pVPosition</i>	Pointer to the internal workspace index. The function updates this parameter.
<i>mode</i>	Flag that indicates whether or not PCM audio output channels should be interleaved. 1 indicates not interleaved, 2 indicates interleaved.

### Description

The function `ippsSynthPQMF_MP3` is declared in the `ippac.h` file. This function performs the second stage of the hybrid synthesis filter bank, that is, a critically-sampled 32-channel PQMF synthesis bank that generates 32 time-domain output samples for each 32-sample input block of IMDCT outputs.

For each input block, the PQMF generates an output sequence of 16-bit signed little-endian PCM samples in the vector pointed to by *pDstAudioOut*.

If *mode* equals to 2, the left and right channel output samples are interleaved, that is, LRLRLR, so that the left channel data is organized as follows:

```
pDstAudioOut[2*i ], i = 0 to 31.
```

If *mode* equals 1, then the left and right channel outputs are not interleaved.

Since PQMF bank contains memory, the MP3 application must maintain two state variables in between calls to the function.

The application must preallocate a workspace buffer of size 512 x (Number of Channels) for the PQMF computation. This buffer is referenced by the pointer *pVBuffer* and its elements must be initialized to zero prior to the first call. During subsequent calls, the *pVBuffer* input for the current call must contain the output generated by the previous call.

The MP3 application must also initialize to zero and thereafter preserve the value of the state variable *pVPosition*. The MP3 application must modify the values contained in *pVBuffer* or *pVPosition* only during decoder reset, and the reset values mustd always be zero.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when least one of the pointers <i>pSrcY</i> , <i>pDstAudioOut</i> , <i>pVBuffer</i> , or <i>pVPosition</i> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when at least one of the specified pointers is NULL, or the value of <i>mode</i> is less than 1 or more than 2, or when the value of <i>pVPosition</i> exceeds [0, 15].
<code>ippStsErr</code>	Indicates an unknown error.

## SynthesisFilterInit\_PQMF\_MP3

*Initializes PQMF MP3 synthesis specification structure.*

---

### Syntax

```
IppStatus ippsSynthesisFilterInit_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3**
ppFilterSpec, Ipp8u* pMemSpec);
```

### Parameters

<i>ppFilterSpec</i>	Double pointer to the PQMF MP3 synthesis specification structure.
<i>pMemSpec</i>	Pointer to the area for the PQMF MP3 synthesis specification structure.

### Description

The function `ippsSynthesisFilterInit_PQMF_MP3` is declared in the `ippac.h` file. This function creates and initializes a polyphase quadrature mirror MP3 synthesis specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .

## SynthesisFilterInitAlloc\_PQMF\_MP3

*Allocates memory and initialized the PQMF MP3 synthesis specification structure.*

---

### Syntax

```
IpStatus ippsSynthesisFilterInitAlloc_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3**  
ppFilterSpec);
```

### Parameters

<code>ppFilterSpec</code>	Double pointer to the PQMF MP3 synthesis specification structure.
---------------------------	---

### Description

The function `ippsSynthesisFilterInitAlloc_PQMF_MP3` is declared in the `ippac.h` file. This function allocates memory for polyphase quadrature mirror MP3 synthesis specification structure, creates, and initializes this structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .

## SynthesisFilterGetSize\_PQMF\_MP3

Returns size of PQMF MP3 synthesis specification structure.

---

### Syntax

```
IppStatus ippsSynthesisFilterGetSize_PQMF_MP3_32f(int* pSizeSpec);
```

### Parameters

<i>pSizeSpec</i>	Pointer to the size (in bytes) of the PQMF MP3 synthesis specification structure.
------------------	---

### Description

The function `ippsSynthesisFilterGetSize_PQMF_MP3` is declared in the `ippac.h` file. This function gets size of the polyphase quadrature mirror MP3 synthesis specification structure and stores the result in *pSizeSpec*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSizeSpec</i> is NULL.

## SynthesisFilterFree\_PQMF\_MP3

Frees memory allocated for PQMF MP3 synthesis specification structure.

---

### Syntax

```
IppStatus ippsSynthesisFilterFree_PQMF_MP3_32f(IppsFilterSpec_PQMF_MP3* pFilterSpec);
```

### Parameters

<i>pFilterSpec</i>	Pointer to the initialized PQMF MP3 synthesis specification structure.
--------------------	--



## Description

The function `ippsSynthesisFilterFree_PQMF_MP3` is declared in the `ippac.h` file. This function frees memory allocated for the polyphase quadrature mirror MP3 synthesis specification structure.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the specified pointers is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the `pFilterSpec` identifier is incorrect.

## SynthesisFilter\_PQMF\_MP3

*Transforms PQMF MP3-processed subband signals into time domain samples.*

---

## Syntax

```
IppStatus ippsSynthesisFilter_PQMF_MP3_32f(const Ipp32f* pSrc, Ipp32f* pDst,
const IppsFilterSpec_PQMF_MP3* pFilterSpec, int mode);
```

## Parameters

*pSrc* Array of pointers, holds PQMF MP3-processed subband signals.

*pDst* Pointer to the output vector, holds time domain output samples.

*pFilterSpec* Pointer to the initialized PQMF MP3 synthesis specification structure.

*mode* Flag that indicates whether or not MP3 audio output channels should be interleaved:  
 1 - indicates no interleaving, 2 - indicates interleaving.

## Description

The function `ippsSynthesisFilter_PQMF_MP3` is declared in the `ippac.h` file. This function performs the polyphase quadrature mirror filter bank, that is, a critically-sampled 32-channel PQMF synthesis bank that generates 32 time-domain output samples for each 32-sample input block of IMDCT outputs. For each input block, the PQMF generates an output sequence of PCM samples in the vector pointed to by *pDst*. If *mode* equals 2, the left and right channel output

samples are interleaved, that is, LRLRLR, so that the left channel data is organized as follows: *pDst* [2\**i*], *i* = 0 to 31. If *mode* equals 1, then the left and right channel outputs are not interleaved.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>mode</i> exceeds [1,2].
<code>ippStsContextMatchErr</code>	Indicates an error when the <i>pFilterSpec</i> identifier is incorrect.

## Advanced Audio Coding Functions

The ISO/IEC 13818-7 MPEG-2 AAC (Advanced Audio Coding) algorithm is an efficient coding method for surround signals like 5-channel signals (left, right, center, left surround, right surround).

MPEG-2 AAC supports up to 48 main audio channels with the sampling frequency between 8kHz and 96kHz. MPEG formal tests have shown that for 5-channel audio signals, AAC satisfies the ITU-R quality requirements and provides slightly better audio quality at 320 kilobits per second (kbps) than MPEG-2 BC (Backwards Compatibility) provides at 640 kbps.

Due to its high coding efficiency, AAC is a prime candidate for any digital broadcasting system and has been selected by the DRM (Digital Radio Mondiale) system. AAC also plays a major role for the delivery of high quality music via the Internet. Moreover, with some modifications, AAC is the only high quality audio coding scheme adopted within the MPEG-4 standard, the future “global multimedia language”.

### Global Macros

Table 10-4 below shows the definitions of global macros.

**Table 10-4. Global Macro Definitions**

Global Macro Name	Definition	Notes
<code>IPP_AAC_FRAME_LEN</code>	1024	size of data in one frame
<code>IPP_AAC_SF_LEN</code>	120	scale factor buffer length
<code>IPP_AAC_GROUP_NUM_MAX</code>	8	maximum group number for one frame

Global Macro Name	Definition	Notes
IPP_AAC_TNS_COEF_LEN	60	TNS coefficients buffer length
IPP_AAC_TNS_FILT_MAX	8	maximum TNS filter number for one frame
IPP_AAC_PRED_SFB_MAX	41	maximum prediction scale factor bands number for one frame
IPP_AAC_ELT_NUM	16	maximum number of elements for one program
IPP_AAC_LFE_ELT_NUM	4	maximum Low Frequency Enhance elements number for one program
IPP_AAC_DATA_ELT_NUM	8	maximum data elements number for one program
IPP_AAC_COMMENTS_LEN	256	maximum length of the comment field in bytes
IPP_AAC_PULSE_NUM_MAX	5	maximum number of the pulse escapes

## Data Types and Structures

This section describes the data types and structures of the Intel® IPP advanced audio coding.

## ADIF Header

```
typedef struct {
    Ipp32u  ADIFId; /* 32-bit, "ADIF" ASCII code */
    int     copyIdPres; /* copy id flag: 0 - off, 1 - on */
    int     originalCopy; /* copyright bit:
        0 - no copyright on the coded bitstream,
        1 - copyright protected */
    int     home; /* original/home bit:
        0 - the bitstream is a copy,
        1 - the bitstream is the original
    int     bitstreamType; /* bitstream flag:
        0 - constant rate bitstream,
        1 - variable rate bitstream */
    int     bitRate; /* bit rate: if 0, unknown bit rate */
    int     numPrgCfgElt; /* number of program configure elements */
    int     pADIFBufFullness[IPP_AAC_ELT_NUM]; /*buffer fullness */
    Ipp8u    pCopyId[9]; /* 72-bit copy id */
} IppAACADIFHeader;
```

## ADTS Frame Header

```
typedef struct {
    /* ADTS fixed header */
    int id; /* ID 1*/
    int layer; /* layer index:0x3 - Layer I
        0x2 - LayerII
        0x1 - LayerIII */
    int protectionBit; /* CRCflag 0: CRC on, 1: CRC off */
    int profile; /* profile: 0:MP,1:LC, 2:SSR */
    int samplingRateIndex; /* sampling rate index */
}
```

---

```
int privateBit;      /* private_bit, no use */
int chConfig;        /* channel configure*/
int originalCopy;    /* copyright bit:
    0 - no copyright on the coded bitstream, 1 - copyright protected */
int home;            /* original/home bit:
    0 - the bitstream is a copy,
    1 - the bitstream is the original
int emphasis;        /* not used by ISO/IEC 13818-7, but used
    by 14496-3 */
```

```

/* ADTS variable header */

int cpRightIdBit;      /* copyright id bit */
int cpRightIdStart;    /* copyright id start */
int frameLen;         /* frame length in bytes */
int ADTSBufFullness;   /* buffer fullness */
int numRawBlock;       /* number of raw data blocks in the frame */
/* ADTS CRC error check, 16bits      */
int CRCWord;          /* CRC-check word */
} IppAACADTSFrameHeader;

```

## Individual Channel Side Information

```

typedef struct {
    /* unpacked from the bitstream */

    int icsReservedBit;    /* reserved bit */
    int winSequence;       /* window sequence flag */
    int winShape;          /* window shape flag:
        0 - sine window,
        1 - KBD window */
    int maxSfb;            /* maximum effective scale factor bands */
    int sfGrouping;        /* scale factor grouping information */
    int predDataPres;      /* prediction data present flag for one
        frame:
        0 - prediction off,
        1 - prediction on */
    int predReset;         /* prediction reset flag:
        0 - reset off,
        1 - reset on */
    int predResetGroupNum; /* prediction reset group number */
    Ipp8u pPredUsed[IPP_AAC_PRED_SFB_MAX+3]; /* prediction flag buffer
        for each scale factor band:

```

```

    0 - off,
    1 - on buffer length 44 bytes, 4-byte align      */
/* decoded from the above info */
int  numWinGrp;      /* group number */
int  pWinGrpLen[IPP_AAC_GROUP_NUM_MAX];             /* buffer for number of windows in each
group */
} IppAACIcsInfo;

```

### AAC Scalable Main Element Header

```

typedef struct{
    int windowSequence; /* windows is short or long type */
    int windowShape;    /* what window is used for the right hand
    part of this analysis window */
    int maxSfb; /* number of scale factor band transmitted */
    int sfGrouping; /* grouping of short spectral data */
    int numWinGrp; /* window group number */
    int pWinGrpLen[IPP_AAC_GROUP_NUM_MAX]; /* length of every group */
    int msMode; /* MS stereo flag:

    0 - none,

    1 - different for every sfb,

    2 - all */
    Ipp8u (*ppMsMask)[IPP_AAC_SF_MAX]; /* if MS's used in one sfb, when msMode ==1 */
    IppAACTnsInfo pTnsInfo[IPP_AAC_CHAN_NUM]; /* TNS structure for
    channels */
    IppAACLtpInfo pLtpInfo[IPP_AAC_CHAN_NUM]; /* LTP structure for
    channels */
}IppAACMainHeader;

```

## AAC Scalable Extension Element Header

```
typedef struct{
    int msMode;    /* 0 - non; 1 - part; 2 - all */
    int maxSfb;    /* number of scale factor band for extension layer */
    Ipp8u (*ppMsMask)[IPP_AAC_SF_MAX]; /* if ms is used */
    IppAACTnsInfo pTnsInfo[IPP_AAC_CHAN_NUM]; /* TNS structure for
                                                Stereo */
    int pDiffControlLr[IPP_AAC_CHAN_NUM][IPP_AAC_SFB_MAX]; /* FSS
                                                            information for stereo */
}IppAACExtHeader;
```



## AAC Program Config Element

```
typedef struct {
    int eltInstTag;           /* element instance tag */
    int profile;              /* profile index:
                               0 - main,
                               1 - Low Complexity,
                               2 - Scaleable Sampling Rate,
                               3 - reserved */
    int samplingRateIndex;    /* sampling rate index */
    int numFrontElt;          /* number of front elements */
    int numSideElt;           /* number of side elements */
    int numBackElt;           /* number of back elements */
    int numLfeElt;            /* number of LFE elements */
    int numDataElt;           /* number of data elements */
    int numValidCcElt;         /* number of coupling channel elements */
    monoMixdownPres;          /* mono mixdown flag:
                               0 - off,
                               1 - on */
    int monoMixdownEltNum;     /* number of mono mixdown elements */
    int stereoMixdownPres;     /* stereo mixdown flag:
                               0 - off,
                               1 - on */
    int stereoMixdownEltNum;   /* number of stereo mixdown elements */
    int matrixMixdownIdxPres; /* matrix mixdown flag:
                               0 - off,
                               1 - on */
    int matrixMixdownIdx;      /* identifier of the surround mixdown coefficient */
    int pseudoSurroundEnable; /* pseudo surround:
                               0 - off,
```

```

1 - on */

int pFrontIsCpe[IPP_AAC_ELT_NUM];    /* channel pair flag for
                                       front elements */

int pFrontTagSel[IPP_AAC_ELT_NUM];    /* instance tag for front
                                       elements */

int pSideIsCpe[IPP_AAC_ELT_NUM];     /* channel pair flag for
                                       side elements */

int pSideTagSel[IPP_AAC_ELT_NUM];     /* instance tag for side
                                       elements */

int pBackIsCpe[IPP_AAC_ELT_NUM];     /* channel pair flag for
                                       back elements */

int pBackTagSel[IPP_AAC_ELT_NUM];     /* instance tag for back
                                       elements */

int pLfeTagSel[IPP_AAC_LFE_ELT_NUM];  /* channel pair flag for
                                       LFE elements */

int pDataTagSel[IPP_AAC_DATA_ELT_NUM]; /* instance tag for data
                                       elements */

int pCceIsIndSw[IPP_AAC_ELT_NUM];     /* independent flag for
                                       coupling */

    /* channel elements */

int pCceTagSel[IPP_AAC_ELT_NUM];     /* instance tag for coupling
                                       channel elements */

int numComBytes;                     /* number of comment field
                                       bytes */

Ipp8s pComFieldData[IPP_AAC_COMMENTS_LEN]; /* comment buffer field */
}IppAACPrgCfgElt;

```

## Temporal Noise Shaping (TNS) Structure for One Layer

```
typedef struct{
    int tnsDataPresent; /* Indicator of TNS data present:
        0 - not present,
        1 - present */
    int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX]; /*TNS number filter buffer*/
    int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX]; /* TNS coef resolution
        flag */
    int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX]; /* TNS filter length */
    int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX]; /* TNS filter order */
    int pTnsDirection[IPP_AAC_TNS_FILT_MAX]; /* TNS filter direction
        flag */
    int pTnsCoefCompress[IPP_AAC_GROUP_NUM_MAX]; /* The most
        significant bit of the coefficientsof the noise
        shaping filter in window w isomitted or not */
    Ipp8s pTnsFiltCoef[IPP_AAC_TNS_COEF_LEN];
    /* Coefficients of one
        noise shaping filter applied to window w */
}IppAACTnsInfo;
```

## Long Term Prediction (LTP) Structure

```
typedef struct{
    int ltpDataPresent; /* Indicator of LTP data present:
        0 - not present,
        1 - present */
```

```
int ltpLag;                /* the optimal delay from 0 to 2047 */
Ipp16s ltpCoef;           /* indicate the LTP coefficient */
int pLtpLongUsed[IPP_AAC_SF_MAX+1]; /* if long block use ltp */
int pLtpShortUsed[IPP_AAC_WIN_MAX]; /* if short block use ltp */

int pLtpShortLagPresent[IPP_AAC_WIN_MAX]; /* if short lag is
                                           transmitted */
```

```

int pLtpShortLag[IPP_AAC_WIN_MAX]; /* relative delay for short
                                     window */
}IppAACLtpInfo;

```

## Channel Pair Element

```

typedef struct {
    int commonWin;          /* common window flag:
                             0 - off,
                             1 - on */
    int msMaskPres;         /* MS stereo mask present flag */
    Ipp8u pMsMask[IPP_AAC_GROUP_NUM_MAX][IPP_AAC_SF_MAX]; /* MS stereo
                                                             flag buffer for each scalefactor
                                                             band */
} IppAACChanPairElt;

```

## Channel Information

```

typedef struct {
    int tag; /* channel tag */
    int id; /* element id */
    int samplingRateIndex; /* sampling rate index */
    int predSfbMax; /* maximum prediction scale factor bands*/
    int preWinShape; /* previous block window shape */
    int winLen; /* 128 - short window, 1024 - others */
    int numWin; /* 1 - long block, 8 - short block */
    int numSwb; /* decided by sampling frequency and block
                type */

    /* unpacking from the bitstream */
    int globGain; /* global gain */

```

```

int pulseDataPres; /* pulse data present flag: 0 - off, 1 - on */
int tnsDataPres; /* TNS data present flag: 0 - off, 1 - on */
int gainContrDataPres; /* gain control data present flag: 0 - off,
                        1 - on */

/* icsInfo pointer */
IppAACIcsInfo *pIcsInfo; /* pointer to IppAACIcsInfo structure */
/* channel pair pointer */
IppAACChanPairElt *pChanPairElt; /* pointer to IppAACChanPairElt
                                structure */

/* section data */
Ipp8u pSectCb[IPP_AAC_SF_LEN]; /* section code book buffer */
Ipp8u pSectEnd[IPP_AAC_SF_LEN]; /* end of scalefactor offset
                                in each section */

int pMaxSect[IPP_AAC_GROUP_NUM_MAX]; /* maximum section number
                                for each group */

int pTnsNumFilt[IPP_AAC_GROUP_NUM_MAX]; /*TNS number filter
                                number buffer*/

/* TNS data */

int pTnsFiltCoefRes[IPP_AAC_GROUP_NUM_MAX]; /* TNS coefficients
                                resolution flag */

int pTnsRegionLen[IPP_AAC_TNS_FILT_MAX]; /* TNS filter length */
int pTnsFiltOrder[IPP_AAC_TNS_FILT_MAX]; /* TNS filter order */
int pTnsDirection[IPP_AAC_TNS_FILT_MAX]; /* TNS filter direction
                                flag */

}IppAACChanInfo;

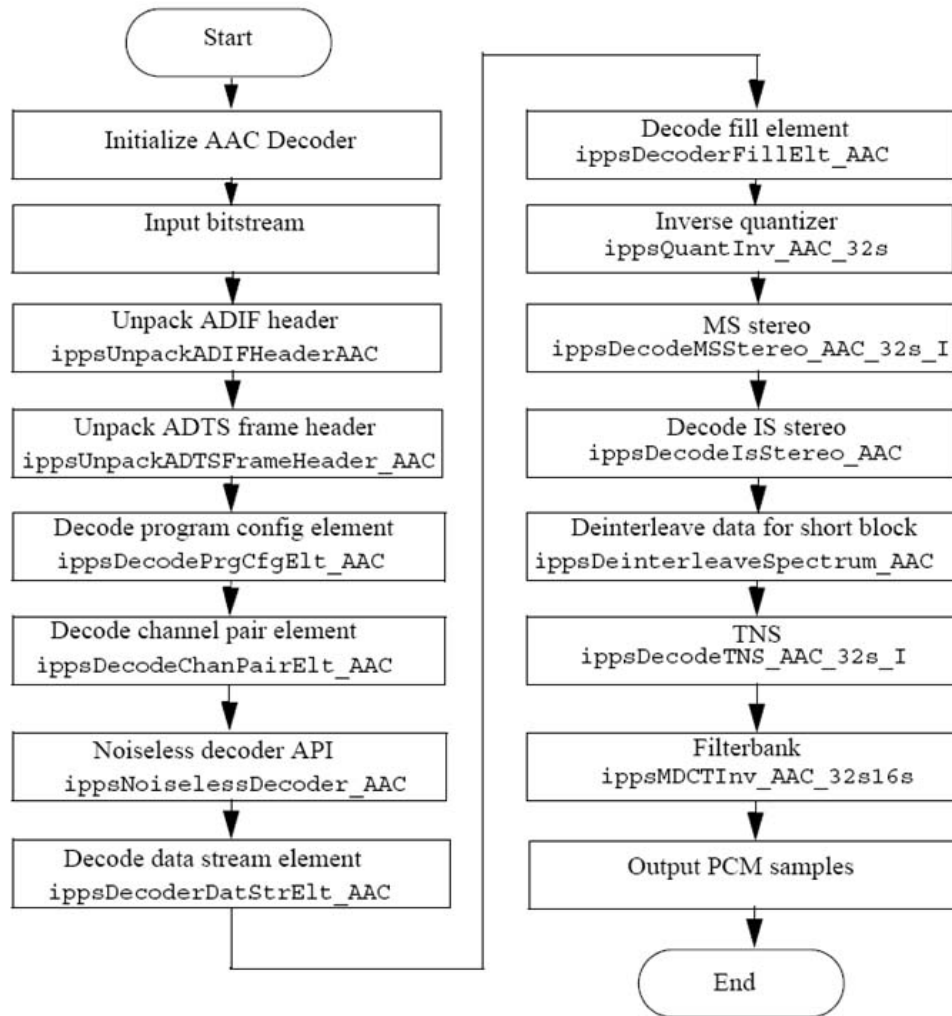
```

## AAC Decoding Functions

The AAC decoder API provides a variety of AAC LC decoder functions, including bitstream unpacking and AAC core decoding functions. This provides customers great flexibility in configuring the decoder system. See [ISO/IEC 13818-7:1997](#).

Figure 10-3 shows AAC decoding flowchart.

**Figure 10-3. AAC Decoding Flowchart**



## MPEG-2 AAC Functions

In the following sections, the parameter *maxSfb* designates the number of scale factor bands transmitted per group and unpacked from the bitstream. The parameter *numSwb* stands for the number of scale factor window bands for the short block or the number of scale factor window bands for the long block. The values are computed according to the sampling rate and the block type.

See clause 8.3.1 of [ISO/IEC 13818 - 7: 1997](#).

## UnpackADIFHeader\_AAC

*Gets the AAC ADIF format header.*

---

### Syntax

```
IppStatus ippUnpackADIFHeader_AAC(Ipp8u** ppBitStream, IppAACADIFHeader*
pADIFHeader, IppAACPrGcFgElt* pPrGcFgElt, int prGcFgEltMax);
```

### Parameters

<i>ppBitStream</i>	Double pointer to the current byte before the ADIF header.
<i>pADIFHeader</i>	Pointer to the <code>IppAACADIFHeader</code> structure.
<i>pPrGcFgElt</i>	Pointer to the <code>IppAACPrGcFgElt</code> structure. There must be <code>prGcFgEltMax</code> elements in the buffer.
<i>prGcFgEltMax</i>	Maximum program configure element number. Must be within the range of [1, 16].

### Description

This function is declared in the `ippac.h` file. The function gets the AAC ADIF format header, including program configuration elements from the input bitstream. See Table 6.2, and 6.21. of [ISO/IEC 13818 - 7: 1997](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pADIFHeader</i> , <i>pPrGcFgElt</i> , or <i>ppBitStream</i> is NULL.



`ippStsAacPrgNumErr` Indicates an error when the decoded `pADIFHeader->numPrgCfgElt > prgCfgEltMax`, or `prgCfgEltMax` is outside the range of `[1, IPP_AAC_MAX_ELT_NUM]`.



**NOTE.** `pADIFHeader->numPrgCfgElt` is the number directly unpacked from bitstream plus 1.

## UnpackADTSFrameHeader\_AAC

*Gets ADTS frame header from the input bitstream.*

### Syntax

```
IppStatus ippUnpackADTSFrameHeader_AAC(Ipp8u** ppBitStream,  
IppAACADTSFrameHeader* pADTSFrameHeader);
```

### Parameters

<code>ppBitStream</code>	Double pointer to the current byte after unpacking the ADTS frame header.
<code>pADTSFrameHeader</code>	Pointer to the <code>IppAACADTSFrameHeader</code> structure.

### Description

The function gets ADTS frame header from the input bitstream. If the CRC word is applied, the first byte of the 16-bit CRC word is stored in `pADTSFrameHeader->CRCWord[15:8]` and the second byte is stored in `pADTSFrameHeader->CRCWord[7:0]`. It does not check whether the header is corrupt.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <code>ppBitStream</code> , <code>pADTSFrameHeader</code> or <code>ppBitStream</code> is NULL.

## DecodePrgCfgElt\_AAC

*Gets program configuration element from the input bitstream.*

---

### Syntax

```
IppStatus ippDecodePrgCfgElt_AAC(Ipp8u** ppBitStream, int* pOffset,
IppAACPrGCfgElt* pPrgCfgElt);
```

### Parameters

<i>ppBitStream</i>	Double pointer to the current byte after decoding the program configuration element.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>ppBitStream</i> . Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
<i>pPrgCfgElt</i>	Pointer to <code>IppAACPrGCfgElt</code> structure.

### Description

This function is declared in the `ippac.h` file. The function gets the program configuration element from the input bitstream. See clause 8.5 and Table 6.21 of [ISO/IEC 13818 - 7: 1997](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>pPrgCfgElt</i> , or <i>ppBitStream</i> is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of [0,7].

## DecodeChanPairElt\_AAC

*Gets channel\_pair\_element from the input bitstream.*

---

### Syntax

```
IppStatus ippDecodeChanPairElt_AAC(Ipp8u** ppBitStream, int* pOffset,
IppAACIcsInfo* pIcsInfo, IppAACChanPairElt* pChanPairElt, int predSfbMax);
```

## Parameters

<i>ppBitStream</i>	Double pointer to the current byte after decoding the channel pair element.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>ppBitStream</i> . Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
<i>pIcsInfo</i>	Pointer to <i>IppAACIcsInfo</i> structure. If <i>pIcsInfo-&gt;predDataPres</i> = 0, set <i>pIcsInfo-&gt;predReset</i> = 0. Only the first <i>pIcsInfo-&gt;numWinGrp</i> elements in <i>pIcsInfo-&gt;pWinGrpLen</i> are meaningful. You must not change some members of the structure, as shown in Table 10-5 below.
<i>pChanPairElt</i>	Pointer to <i>IppAACChanPairElt</i> structure. You must not change some members of the structure, as shown in Table 10-5 below.
<i>predSfbMax</i>	Maximum prediction scale factor bands. For LC profile, set <i>predSfbMax</i> = 0.

## Description

This function is declared in the *ippac.h* file. The function gets the channel pair element from the input bitstream. Individual channel stream is not included.

If *common\_window* flag decoded from the input bitstream is 0, all members of *pIcsInfo* and *pChanPairElt* remain unchanged except for *pChanPairElt->commonWin*.

See clause 8.3 and Table 6.10, 6.11 of [ISO/IEC 13818 - 7: 1997](#).

**Table 10-5. Unchanged Members of *pIcsInfo***

Unchanged Members	Conditions
<i>sfGrouping</i>	<i>pIcsInfo-&gt;winSequence</i> != 2
<i>predResetGroupNum</i>	<i>pIcsInfo-&gt;predDataPres</i> == 0    <i>pIcsInfo-&gt;predReset</i> == 0
<i>pPredUsed[sfb]</i>	<i>pIcsInfo-&gt;predDataPres</i> == 0
<i>pMsUsed[sfb]</i>	<i>pChanPairElt-&gt;msMaskPres</i> != 1

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>ppBitStream</i> , <i>pIcsInfo</i> , or <i>pChanPairElt</i> is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of [0,7].
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>pIcsInfo-&gt;maxSfb</i> decoded from bitstream is greater than <code>IPP_AAC_MAX_SFB</code> , the maximum scale factor band for all sampling frequencies.
<code>ippStsAacPredSfbErr</code>	Indicates an error when <i>predSfbMax</i> is out of the range of [0, <code>IPP_AAC_PRED_SFB_MAX</code> ].

## NoiselessDecoder\_LC\_AAC

*Decodes all data for one channel.*

---

### Syntax

```
IppStatus ippsNoiselessDecoder_LC_AAC(Ipp8u** ppBitStream, int* pOffset, int
commonWin, IppAACChanInfo* pChanInfo, Ipp16s* pDstScalefactor, Ipp32s*
pDstQuantizedSpectralCoef, Ipp8u* pDstSfbCb, Ipp8s* pDstTnsFiltCoef);
```

### Parameters

<i>ppBitStream</i>	Double pointer to the current byte.
<i>pOffset</i>	Pointer to the offset in one byte.
<i>commonWin</i>	Common window indicator.
<i>pChanInfo</i>	Pointer to the channel information. <code>IppAACChanInfo</code> structure. Denotes <i>pIcsInfo</i> as <i>pChanInfo-&gt;pIcsInfo</i> as shown in Table 10-6.
<i>pDstScalefactor</i>	Pointer to the scalefactor or intensity position buffer. Buffer length is more than or equal to 120. Only <code>maxSfb</code> elements are stored for each group. There is no space between sequence groups.
<i>pDstQuantizedSpectralCoef</i>	Pointer to the quantized spectral coefficients data. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length is more than or equal to 1024.

<i>pDstSfbCb</i>	Pointer to the scale factor band codebook. Buffer length must be more than or equal to 120. Store <code>maxSfb</code> elements for each group. There is no space between the sequence groups.
<i>pDstTnsFiltCoef</i>	Pointer to TNS coefficients. Buffer length must be more than or equal to 60. The store sequence is TNS order elements for each filter for each window. The elements are not changed if the corresponding TNS order is zero.

## Description

This function is declared in the `ippac.h` file. The function decodes all data for one channel, including scale factor/intensity positions, spectral coefficients, TNS coefficients, and associated side information for LC profile.

You need to set `pChanInfo->pIcsInfo`, `pChanInfo->samplingRateIndex`, `pChanInfo->predSfbMax` to correct pointer/values before calling this function.

**Table 10-6 Input/Output Members List of `pChanInfo`**

Member	Output
<i>Tag</i>	Not used.
<i>id</i>	Not used.
<i>preWinShape</i>	Not used.
<i>pChanPairElt</i>	Not used.
<i>samplingRateIndex</i>	As input. Not changed.
<i>predSfbMax</i>	As input. Must be 0. Not changed.
<i>winLen</i>	As output. Set to 128, if decoded, <code>pIcsInfo-&gt;winSequence</code> is short block. Otherwise set to 1024.
<i>numWin</i>	As output. Set to 8, if decoded <code>pIcsInfo-&gt;winSequence</code> is short block. Otherwise set to 1.
<i>numSwb</i>	As output. Set to the maximum number of scale factor window bands in each group according to <code>samplingRateIndex</code> and <code>pIcsInfo-&gt;winSequence</code> . See Table 8.4-8.1 of <a href="#">ISO/IEC 13818 - 7: 1997</a> .

Member	Output
<i>globGainpulseDataPres tnsDataPres</i> <i>gainContrDataPres</i> <i>pMaxSect</i>	As output. Unpacked from bitstream.
<i>pSectCb</i>	As output. Pointer to the maximum of sections number in each group. Only <i>pIcsInfo-&gt;numWinGrp</i> elements in the buffer are meaningful.
<i>pTnsRegionLen</i>	As output. Pointer to the section codebook. Only <i>pMaxSect[g]</i> elements are stored for each group. There is no space between the sequence groups.
<i>pTnsFiltOrder</i>	As output. Pointer to the length of the region in units of scale factor bands to which one filter is applied in each window.
<i>pTnsDirection</i>	As output. Pointer to the order of the temporal noise shaping filter applied to each window.
<i>pIcsInfo</i>	As output. Pointer to the token that indicates whether the filter is applied in the upward or downward direction. 0 stands for upward and 1 for downward.
	As input if <i>commonWin == 1</i> . As output if <i>commonWin == 0</i> . If <i>pIcsInfo-&gt;predDataPres == 0</i> , set <i>pIcsInfo-&gt;predReset = 0</i> . Only the first <i>pIcsInfo-&gt;numWinGrp</i> elements in <i>pIcsInfo-&gt;pWinGrpLen</i> are meaningful. Under specific conditions, some members of the structure must remain unchanged. See <a href="#">Table 10-5</a> .

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>pChanInfo</i> , <i>pDstScalefactor</i> , <i>pDstQuantizedSpectralCoef</i> , <i>pDstSfbCb</i> , <i>pDstTnsFiltCoef</i> , <i>pChanInfo-&gt;pIcsInfo</i> , or <i>*ppBitStream</i> is NULL.
<i>ippStsAacBitOffsetErr</i>	Indicates an error when <i>pOffset</i> is out of range [0,7].
<i>ippStsAacComWinErr</i>	Indicates an error when <i>commonWin</i> exceeds [0,1].
<i>ippStsAacSmpRateIdxErr</i>	Indicates an error when <i>pChanInfo-&gt;samplingRateIndex</i> exceeds [0,11].

- `ippStsAacPredSfbErr` Indicates an error when `pChanInfo->predSfbMax` is not equal to 0.
- `ippStsAacMaxSfbErr` Indicates an error when `pChanInfo->pIcsInfo->maxSfb > pChanInfo->numSwb`.
- `ippStsAacSectCbErr` Indicates an error when the codebook pointed to by `pChanInfo->pSectCb` is illegal or when `( pChanInfo->pSectCb )==12, 13`. If the current channel is not the right channel of the channel pair element, `pSectCb = 14, 15` is also illegal.
- `ippStsAacPlsDataErr` Indicates an error when the `pChanInfo->pIcsInfo->winSequence` indicates a short sequence and `pChanInfo->pulsePres` indicates pulse data present. The start scale factor band for pulse data `>= pChanInfo->numSwb`, or pulse data position `offset >= winLen`.
- `ippStsAacGainCtrErr` Indicates an error when `pChanInfo->gainControlPres` is decoded as 1, which means that gain control data is present. Gain control data is not currently supported.
- `ippStsAacCoefValErr` Indicates an error when the quantized coefficients value pointed to by `pDstCoef` exceeds range `[-8191, 8191]`.

## DecodeDatStrElt\_AAC

Gets data stream element from the input bitstream.

### Syntax

```
IppStatus ippsDecodeDatStrElt_AAC(Ipp8u** ppBitStream, int* pOffset, int*
pDataTag, int* pDataCnt, Ipp8u* pDstDataElt);
```

### Parameters

- |                          |   |
|--------------------------|---|
| <code>ppBitStream</code> | Double pointer to the current byte after the decode data stream element.  |
| <code>pOffset</code>     | Pointer to the bit position in the byte pointed to by <code>ppBitStream</code> . Valid within 0 to 7.<br>0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte. |
| <code>pDataTag</code>    | Pointer to <code>element_instance_tag</code> . See Table 6.20 of <a href="#">ISO/IEC 13818 - 7: 1997</a> .  |

<i>pDataCn</i>	Pointer to the value of data length in bytes.
<i>pDstDataElt</i>	Pointer to the data stream buffer that contains the data stream extracted from the input bitstream. There are 512 elements in the buffer pointed to by <i>pDstDataElt</i> .

### Description

This function is declared in the `ippac.h` file. The function gets data stream element from the input bitstream.

See clause 8.6 and Table 6.20 of [ISO/IEC 13818 - 7: 1997](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers <i>ppBitStream</i> , <i>pOffset</i> , <i>ppBitStream</i> , <i>pDataTag</i> , <i>pDataCnt</i> , or <i>pDstDataElt</i> is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of range [0,7].

## DecodeFillElt\_AAC

Gets the fill element from the input bitstream.

### Syntax

```
IppStatus ippDecodeFillElt_AAC(Ipp8u** ppBitStream, int* pOffset, int* pFillCnt, Ipp8u* pDstFillElt);
```

### Parameters

<i>ppBitStream</i>	Pointer to the pointer to the current byte after the decode fill element.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>ppBitStream</i> . Valid within 0 to 7. 0 stands for the most significant bit of the byte. 7 stands for the least significant bit of the byte.
<i>pFillCnt</i>	Pointer to the value of the length of total fill data in bytes.



*pDstFillElt*

Pointer to the fill data buffer whose length must be equal to or greater than 270.

### Description

This function is declared in the `ippac.h` file. The function gets the fill element from the input bitstream.

See clause 8.7 and Table 6.22 of [ISO/IEC 13818 - 7: 1997](#).

### Return Values

*ippStsNoErr*

Indicates no error.

*ippStsNullPtrErr*

Indicates an error when at least one of the pointers *ppBitStream*, *pOffset*, *pFillCnt*, or *pDstFillElt* is NULL.

*ippStsAacBitOffsetErr*

Indicates an error when *pOffset* is out of the range of [0,7].

## QuantInv\_AAC

*Performs inverse quantization of Huffman symbols for current channel in-place.*

---

### Syntax

```
IppStatus ippsQuantInv_AAC_32s_I(Ipp32s* pSrcDstSpectralCoef, const Ipp16s*
pScalefactor, int numWinGrp, const int* pWinGrpLen, int maxSfb, const Ipp8u*
pSfbCb, int samplingRateIndex, int winLen);
```

### Parameters

*pSrcDstSpectralCoef*

On input, pointer to the input quantized coefficients. For short block the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.

On output, pointer to the destination inverse quantized coefficient in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group.

	Buffer length must be more than or equal to 1024. The maximum error of output <i>pSrcDstSpectralCoef</i> [i] is listed in Table 10-7"Computation Error List for <i>pSrcDstSpectralCoef</i> ".
<i>pScalefactor</i>	Pointer to the scalefactor buffer. Buffer length must be more than or equal to 120.
<i>numWinGrp</i>	Group number.
<i>pWinGrpLen</i>	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
<i>maxSfb</i>	Maximal scale factor bands number for the current block.
<i>pSfbCb</i>	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Only <i>maxSfb</i> elements for each group are meaningful. There are no spaces between the sequence groups.
<i>samplingRateIndex</i>	Sampling rate index. Valid within [0, 11]. See Table 6.5 of <a href="#">ISO/IEC 13818 - 7: 1997</a> .
<i>winLen</i>	Data number in one window.

## Description

This function is declared in the `ippac.h` file. The function performs inverse quantization of Huffman symbols for the current channel as shown by the formula.

$$pSrcDst[i] = \text{sign}(pSrcDst[i]) * (pSrcDst[i])^{3/4} * 2^{(1/2(pScalefactor[sfb] - 100))}$$

See clause 10 of [ISO/IEC 13818 - 7: 1997](#).

**Table 10-7 Computation Error List for *pSrcDstSpectralCoef***

Output	Conditions	
max(error ( <i>pSrcDstSpectralCoef</i> [i]))	Input abs ( <i>pSrcDstSpectralCoef</i> [i])	Output abs ( <i>pSrcDstSpectralCoef</i> [i])
3	<= 128	< 2 ^ 29
3	129~8191	<= 2 ^ 25
7	129~8191	< 2 ^ 29

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

`ippStsNullPtrErr` Indicates an error when at least one of the specified pointers is NULL.

`ippStsAacSmpRateIdxErr` Indicates an error when *samplingRateIndex* exceeds [0, 11].

`ippStsAacMaxSfbErr` Indicates an error when *maxSfb* exceeds [0, IPP\_AAC\_MAX\_SFB].

`ippStsAacWinGrpErr` Indicates an error when *numWinGrp* exceeds [0, 8] for long window or is not equal to 1 for short window.

`ippStsAacWinLenErr` Indicates an error when *winLen* is not equal to 128 or 1024;

`ippStsAacCoefValErr` Indicates an error when the quantized coefficients value pointed to by *pSrcDstSpectralCoef* exceeds [-8191, 8191].

## DecodeMsStereo\_AAC

*Processes mid-side (MS) stereo for pair channels in-place.*

---

### Syntax

```
ippStatus ippsDecodeMsStereo_AAC_32s_I(Ipp32s* pSrcDstL, Ipp32s* pSrcDstR,
int msMaskPres, const Ipp8u* pMsUsed, Ipp8u* pSfbCb, int numWinGrp, const
int* pWinGrpLen, int maxSfb, int samplingRateIndex, int winLen);
```

### Parameters

<i>pSrcDstL</i>	On input, pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024. On output, pointer to left channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
<i>pSrcDstR</i>	On input, pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.

	On output, pointer to right channel data in Q13.18 format. For short blocks, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
<i>msMaskPres</i>	MS stereo mask flag: 0 - MS off; 1 - MS on; 2 - MS all bands on.
<i>pMsUsed</i>	Pointer to the MS Stereo flag buffer. Buffer length must be more than or equal to 120.
<i>pSfbCb</i>	Pointer to the scale factor band codebook. If <code>invert_intensity (group, sfb) = -1</code> , and if <code>*pSfbCb = INTERITY_HCB</code> , let <code>*pSfbCb = INTERITY_HCB2</code> . If <code>*pSfbCb = INTERITY_HCB2</code> , let <code>*pSfbCb = INTERITY_HCB</code> . Buffer length must be more than or equal to 120. Store <code>maxSfb</code> elements for each group. There is no space between the sequence groups.
<i>numWinGrp</i>	Group number.
<i>pWinGrpLen</i>	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
<i>maxSfb</i>	Maximal scale factor bands number for the current block.
<i>samplingRateIndex</i>	Sampling rate index. Valid within [0, 11]. See Table 6.5 of <a href="#">ISO/IEC 13818 - 7: 1997</a> .
<i>winLen</i>	Data number in one window.

## Description

This function is declared in the `ippac.h` file. The function performs mid-side (MS) stereo process for pair channels and at the same time runs the `invert_intensity(group, sfb)` function and stores the values in the `pSfbCb` buffer.

In the case when MS stereo flag is on, the operation is performed according the following formulas:

$$pSrcDstL'[i] = pSrcDstL[i] + pSrcDstR[i];$$

$$pSrcDstR'[i] = pSrcDstL[i] - pSrcDstR[i];$$

See clause 12 of [ISO/IEC 13818 - 7: 1997](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when the coefficient index calculated from <code>samplingFreqIndex</code> and <code>maxSfb</code> exceeds <code>winLen</code> in each window.
<code>ippStsAacSampRateIdxErr</code>	Indicates an error when <code>pChanInfor-&gt; samplingRateIndex</code> exceeds <code>[0,11]</code> .
<code>ippStsAacWinGrpErr</code>	Indicates an error when <code>numWinGrp</code> exceeds <code>[0,8]</code> for long window or is not equal to 1 for short window.
<code>ippStsAacWinLenErr</code>	Indicates an error when <code>winLen</code> is not equal 128 or 1024.
<code>ippStsStereoMaskErr</code>	Indicates an error when the stereo mask flag is not equal 1 or 2.

## DecodIsStereo\_AAC

*Processes intensity stereo for pair channels.*

### Syntax

```
IppStatus ippsDecodeIsStereo_AAC_32s(const Ipp32s* pSrcL, Ipp32s* pDstR,
const Ipp16s* pScalefactor, const Ipp8u* pSfbCb, int numWinGrp, const int*
pWinGrpLen, int maxSfb, int samplingRateIndex, int winLen);
```

### Parameters

<code>pSrcL</code>	Pointer to left channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
<code>pScalefactor</code>	Pointer to the scalefactor buffer. Buffer length must be more than or equal to 120.
<code>pSfbCb</code>	Pointer to the scale factor band codebook, buffer length must be more than or equal to 120. Store <code>maxSfb</code> elements for each group. There is no space between the sequence groups.

	Respective values of <i>pSfbCb[sfb]</i> equal to 1, -1, or 0 indicate the intensity stereo mode, that is, direct, inverse, or none.
<i>numWinGrp</i>	Group number.
<i>pWinGrpLen</i>	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
<i>maxSfb</i>	Maximal scalefactor bands number for the current block.
<i>samplingRateIndex</i>	Sampling rate index. Valid within [0, 11]. See Table 6.5 of <a href="#">ISO/IEC 13818 - 7: 1997</a> .
<i>winLen</i>	Data number in one window.
<i>pDstR</i>	Pointer to right channel data in Q13.18 format. For short block, the coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.

## Description

This function is declared in the `ippac.h` file. The function processes intensity stereo for pair channels.

Operation is performed according to the following formula.

$$pDstR[i] = pSrc[i] * is\_intensity(g, sfb) * 2^{(-1/4(pScalefactor[sfb])}$$




---

**NOTE.** `invert_intensity(g, sfb)` is not used in the formula, because it is already decoded and stored in *pSfbCb[sfb]* in the MS stereo process primitive. Refer to clause 12 of [ISO/IEC 13818 - 7: 1997](#).

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when the coefficient index calculated from <i>samplingFreqIndex</i> and <i>maxSfb</i> exceeds <i>winLen</i> in each window.

`ippStsAacSamplRateIdxErr` Indicates an error when `pChanInfor-> samplingRateIndex` exceeds `[0,11]`.

`ippStsAacWinGrpErr` Indicates an error when `numWinGrp` exceeds `[0,8]` for long window or is not equal to 1 for short window.

`ippStsAacWinLenErr` Indicates an error when `winLen` is not equal to 128 or 1024.

## DeinterleaveSpectrum\_AAC

*Deinterleaves the coefficients for short block.*

### Syntax

```
IppStatus ippsDeinterleaveSpectrum_AAC_32s(const Ipp32s* pSrc, Ipp32s* pDst,
int numWinGrp, const int* pWinGrpLen, int maxSfb, int samplingRateIndex, int
winLen);
```

### Parameters

<code>pSrc</code>	Pointer to source coefficients buffer. The coefficients are interleaved by scale factor window bands in each group. Buffer length must be more than or equal to 1024.
<code>numWinGrp</code>	Group number.
<code>pWinGrpLen</code>	Pointer to the number of windows in each group. Buffer length must be more than or equal to 8.
<code>maxSfb</code>	Maximal scalefactor bands number for the current block.
<code>samplingRateIndex</code>	Sampling rate index. Valid in <code>[0, 11]</code> . See Table 6.5 of <a href="#">ISO/IEC 13818 - 7: 1997</a> .
<code>winLen</code>	Data number in one window.
<code>pDst</code>	Pointer to the output of coefficients. Data sequence is ordered in <code>pDst[ w*128+sfb *sfbWidth[ sfb]+i ]</code> , where <code>w</code> is window index, <code>sfb</code> is scale factor band index, <code>sfbWidth</code> is the scale factor band width table, <code>i</code> is the index within scale factor band. Buffer length must be more than or equal to 1024.

## Description

This function is declared in the `ippac.h` file. The function deinterleaves the coefficients for short block.

See clause 8.3.5 of [ISO/IEC 13818 - 7: 1997](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when the coefficient index calculated from <i>samplingFreqIndex</i> and <i>maxSfb</i> exceeds <i>winLen</i> in each window.
<code>ippStsAacSamplRateIdxErr</code>	Indicates an error when <i>pChanInfor-&gt; samplingRateIndex</i> exceeds [0,11].
<code>ippStsAacWinGrpErr</code>	Indicates an error when <i>numWinGrp</i> exceeds [0,8].
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal to 128.

## DecodeTNS\_AAC

*Decodes for Temporal Noise Shaping in-place.*

---

### Syntax

```
IppStatus ippDecodeTNS_AAC_32s_I(Ipp32s* pSrcDstSpectralCoefs, const int*
pTnsNumFilt, const int* pTnsRegionLen, const int* pTnsFiltOrder, const int*
pTnsFiltCoefRes, const Ipp8s* pTnsFiltCoef, const int* pTnsDirection, int
maxSfb, int profile, int samplingRateIndex, int winLen);
```

### Parameters

*pSrcDstSpectralCoefs* On input, pointer to the input spectral coefficients to be filtered by the all-pole filters in Q13.18 format. There are 1024 elements in the buffer .  
On output, pointer to the output spectral coefficients after filtering by the all-pole filters in Q13.18 format.  
See Table 10-8 below for the computation error compared with the double precision data.



<i>pTnsNumFilt</i>	<p>Pointer to the number of noise shaping filters used for each window of the current frame. There are 8 elements in the buffer which are arranged as follows:<i>pTnsNumFilt</i> [<i>w</i>]: the number of noise shaping filters used for window <i>w</i>, <i>w</i> = 0 to <i>numWin</i> - 1.</p>
<i>pTnsRegionLen</i>	<p>Pointer to the length of the region in units of scale factor bands to which one filter is applied in each window of the current frame.</p> <p>There are 8 elements in the buffer, which are arranged as follows:</p> <p><i>pTnsRegionLen</i>[<i>i</i>]: the length of the region to which filter <i>filt</i> is applied in window <i>w</i></p> $i = \sum_{j=0}^{w-1} pTnsNumFilt[j] + filt$ <p>, <i>w</i> = 0 to <i>numWin</i> - 1, <i>filt</i> = 0 to <i>pTnsNumFilt</i>[<i>w</i>] - 1.</p>
<i>pTnsFiltOrder</i>	<p>Pointer to the order of one noise shaping filter applied to each window of the current frame. There are 8 elements in the buffer, which are arranged as follows:</p> <p><i>pTnsFiltOrder</i>[<i>i</i>]: the order of one noise shaping filter <i>filt</i>, which is applied to window <i>w</i>, see formula above.</p>
<i>pTnsFiltCoefRes</i>	<p>Pointer to the resolution of 3 bits or 4 bits of the transmitted filter coefficients for each window of the current frame. There are 8 elements in the buffer, which are arranged as follows:</p> <p><i>pTnsFiltCoefRes</i>[<i>w</i>]: the resolution of the transmitted filter coefficients for window <i>w</i>, <i>w</i> = 0 to <i>numWin</i> - 1.</p>
<i>pTnsFiltCoef</i>	<p>Pointer to the coefficients of one noise shaping filter applied to each window of the current frame. There are 60 elements in the buffer, which are arranged as follows:</p>

*pTnsFiltCoef[i]*, *pTnsFiltCoef[i+1]*, ..., *pTnsFiltCoef[i+order - 1]*: the coefficients of one noise shaping filter *filt*, which is applied to window *w* .

The order is the same as that of the noise shaping filter *filt* as applied to window *w*, *w* = 0 to *numWin*-1, *filt*=0 to *pTnsNumFilt[w]*-1.

For example, *pTnsFiltCoef [0]*, *pTnsFiltCoef [1]*, ..., *pTnsFiltCoef [order0-1]* are the coefficients of the noise shaping filter 0, which is applied to window 0, if present.

If so, *pTnsFiltCoef[ order0]*, *pTnsFiltCoef[ order0+1]*, ..., *pTnsFiltCoef[order0+order1-1]* are the coefficients of the noise shaping filter 1 applied to window 0, if present, and so on.

*order0* is the same as that of the noise shaping filter 0 applied to window 0, and *order1* is the order of the noise shaping filter 1 applied to window 0.

After window 0 is processed, process window 1, then window 2 until all *numWin* windows are processed.

*pTnsDirection*

Pointer to the token that indicates whether the filter is applied in the upward or downward direction.

0 stands for upward and 1 for downward.

There are 8 elements in the buffer pointed to by *pTnsDirection* which are arranged as follows:

*pTnsDirection[i]*: the token indicating whether the filter *filt* is applied in upward or downward direction to window *w*, *i* see formula above.

*maxSfb*

Number of scale factor bands transmitted per window group of the current frame.

*profile*

Profile index from Table 7.1 in [ISO/IEC 13818 - 7: 1997](#).

*samplingRateIndex*

Index indicating the sampling rate of the current frame.

*winLen*

Data number in one window.

Description

This function is declared in the `ippac.h` file. The function performs decoding process for Temporal Noise Shaping (TNS) that controls the temporal shape of the quantization noise within each window of the transform.

The TNS decoding process proceeds separately for each window of the current frame by applying the all-pole filtering to selected regions of the spectral coefficients.

Table 10-8 Computation Error List for `pSrcDstSpectralCoefs`

MAX(error(pSrcDstSpectralCoefs [i]))	Condition
4095	8 == numWin
32767	1 == numWin

*numWin* is the number of windows in a window sequence of the current frame. *numWin* is equal to 8 if window sequence is `EIGHT_SHORT_SEQUENCE`, or to 1 for other window sequences.

*numSwb* is the total number of scale factor window bands for the actual window type, that is, long or short window of the current frame.



**NOTE.** This function supports LC profile only.



**NOTE.** *numWin* is the number of windows in a window sequence of the current frame. *numWin* is 8 if window sequence is `EIGHT_SHORT_SEQUENCE`, or 1 for other window sequences.



**NOTE.** *numSwb* is the total number of scale factor window bands for the actual window type, that is, long or short window, of the current frame.

Return Values

- `ippStsNoErr`Indicates no error.
- `ippStsNullPtrErr`Indicates an error when at least one of the specified pointers is NULL.
- `IppStsTnsProfileErr`Indicates an error when *profile*!= 1.

`ippStsAacTnsNumFiltErr` Indicates an error when a data error occurs: for a short window sequence `pTnsNumFilt[w]` exceeds [0,1]; for a long window sequence, `pTnsNumFilt[w]` exceeds [0,3]

`ippStsAacTnsLenErr` Indicates an error when `pTnsRegionLen` exceeds [0, `numSwb`].

`ippStsAacTnsOrderErr` Indicates an error when a data error occurs: for a short window sequence, `pTnsFiltOrder` exceeds [0,7]; for a long window sequence, `pTnsFiltOrder` exceeds [0,12].

`ippStsAacTnsCoefResErr` Indicates an error when `pTnsFiltCoefRes[w]` exceeds [3, 4].

`ippStsAacTnsCoefErr` Indicates an error when `pTnsFiltCoef` exceeds [-8, 7].

`ippStsAacTnsDirectErr` Indicates an error when `pTnsDirection` exceeds [0, 1].

## MDCTInv\_AAC\_32s16s

*Maps time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples.*

---

### Syntax

```
IppStatus ippMDCTInv_AAC_32s16s(Ipp32s* pSrcSpectralCoefs, Ipp16s*
pDstPcmAudioOut, Ipp32s* pSrcDstOverlapAddBuf, int winSequence, int winShape,
int prevWinShape, int pcmMode);
```

### Parameters

<code>pSrcSpectralCoefs</code>	Pointer to the source vector containing 1024 time-frequency domain samples in Q13.18 format.
<code>pSrcDstOverlapAddBuf</code>	Pointer to the overlap-add buffer that contains the second half of the previous block windowed sequence in Q13.18. There are 1024 elements in this buffer.
<code>winSequence</code>	Flag that indicates which window sequence is used for current block.
<code>winShape</code>	Flag that indicates which window function is selected for current block.
<code>prevWinShape</code>	Flag that indicates which window function is selected for previous block.

<i>pcmMode</i>	Flag that indicates whether the PCM audio output is interleaved, that is has the pattern LRLRLR... or not. 1 stands for not interleaved. 2 stands for interleaved.
<i>pDstPcmAudioOut</i>	Pointer to the output vector with 1024 reconstructed 16-bit signed little-endian PCM samples in Q15, interleaved, if needed. The maximum computation error is less than 1 for each vector element. The total quadratic error for the vector is less than 96.
<i>pSrcDstOverlapAddBuf</i>	Pointer to the overlap-add buffer which contains the second half of the current block windowed sequence in Q13.18. The maximum computation error is less than 4 for each vector element. The total quadratic error for the vector is less than 1536.

## Description

This function is declared in the `ippac.h` file. This function maps the time-frequency domain signal into time domain and generates 1024 reconstructed 16-bit signed little-endian PCM samples as output for each channel.

This module consists of

- IMDCT transform
- windowing
- overlap-add operation.

In order to adapt the time/frequency resolution of the filterbank to the characteristics of the input signal, a block switching tool is also adopted. For each channel, 1024 time-frequency domain samples are transformed into the time domain via the IMDCT.

After applying the windowing operation, the first half of the windowed sequence is added to the second half of the previous block windowed sequence to reconstruct 1024 output samples for each channel. Output can be interleaved according to *pcmMode*.

If *pcmMode* equals to 2, the output is in the sequence *pDstPcmAudioOut*[2\*i], i=0 to 1023, that is, 1024 output samples are stored in the sequence: *pDstPcmAudioOut* [0], *pDstPcmAudioOut*[2], *pDstPcmAudioOut*[4],..., *pDstPcmAudioOut*[2046].

If *pcmMode* equals 1, the output is in the sequence *pDstPcmAudioOut* [i], i=0 to 1023.

You should also preallocate an input-output buffer pointed to by *pSrcDstOverlapAddBuf* for the overlap-add operation.

Reset this buffer to zero before the first call and then use the output of the current call as the input of the next call for the same channel.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds [0,3].
<code>ippStsAacWinShapeErr</code>	Indicates an error when <i>winShape</i> or <i>prevWinShape</i> exceeds [0,1].
<code>ippStsAacPcmModeErr</code>	Indicates an error when <i>pcmMode</i> exceeds [1,2].

## MDCTInv\_AAC\_32s\_I

*Computes inverse modified discrete cosine transform (MDCT), windowing and overlapping of signals.*

---

### Syntax

```
IppStatus ippMDCTInv_AAC_32s_I(Ipp32s* pSrcDst, Ipp32s *pSrcDstOverlapBuf,
int winSequence, int winShape, int prevWinShape, int len);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>pSrcDstOverlapBuf</i>	Pointer to overlap-add buffer, contains the output of the 2nd half of the previous frame windowed sequence.
<i>winSequence</i>	Window sequence indicator.
<i>winShape</i>	Window shape indicator.
<i>prevWinShape</i>	Window shape indicator of the previous frame.
<i>len</i>	Number of samples in Src buffer.

## Description

The `ippMDCTInv_AAC_32s_I` function is declared in the `ippac.h` file. This function performs inverse MDCT operation. Then, after applying the windowing operation, the first half of the windowed sequence is added to the second half of the previous frame windowed sequence (contents of `pSrcDstOverlapBuf`) to reconstruct 1024 output samples. The second half of the windowed sequence is stored to `pSrcDstOverlapBuf`.

This function is implemented for using in AAC decoder. It does not perform any saturation so the user should consider overflow possibility. The source and destination have different positions of the decimal point (Q format). The Q format of destination depends on the window sequence indicator.

*WinSequence* = 0 (long window): If *Src* is in Q<sub>n</sub> format then *Dst* in Q<sub>(n+5)</sub> format

*WinSequence* = 1 (long start window): If *Src* is in Q<sub>n</sub> format then *Dst* in Q<sub>(n+5)</sub> format

*WinSequence* = 2 (eight short window): If *Src* is in Q<sub>n</sub> format then *Dst* in Q<sub>(n+2)</sub> format

*WinSequence* = 3 (long stop window): If *Src* is in Q<sub>n</sub> format then *Dst* in Q<sub>(n+5)</sub> format.

For example, to have *Dst* in Q<sub>7</sub> format, the *Src* format should be Q<sub>5</sub> in the case of eight short window and in Q<sub>2</sub> in other cases.

Only *len* = 1024 is supported in the current implementation.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the pointers passed to the function is <code>NULL</code> .
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds [0,3].
<code>ippStsAacWinShapeErr</code>	Indicates an error when <i>winShape</i> or <i>prevWinShape</i> exceeds [0,1].
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not equal to 1024.

## MPEG-4 AAC Functions

This section introduces primitive functions for MPEG-4 AAC operations.

## DecodeMainHeader\_AAC

*Gets main header information and main layer information from bit stream.*

---

### Syntax

```
IppStatus ippDecodeMainHeader_AAC(Ipp8u** ppBitStream, int* pOffset,
IppAACMainHeader* pAACMainHeader, int channelNum, int monoStereoFlag);
```

### Parameters

<i>ppBitStream</i>	Double pointer to bitstream buffer. It is updated after decoding.
<i>pOffset</i>	Pointer to the offset in one byte. It is updated after decoding.
<i>channelNum</i>	Number of channels.
<i>monoStereoFlag</i>	Current frame has mono and stereo layers.
<i>pAACMainHeader</i>	Pointer to the main element header.

### Description

This function is declared in the `ippac.h` file. The function gets main header information and main layer information from bit stream.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> exceeds [0,7].
<code>ippStsAacChanErr</code>	Indicates an error when <i>channelNum</i> exceeds [1,2].
<code>ippStsAacMonoStereoErr</code>	Indicates an error when <i>monoStereoFlag</i> exceeds [0,1].



## DecodeExtensionHeader\_AAC

*Gets extension header information and extension layer information from bit stream.*

---

### Syntax

```
IppStatus ippDecodeExtensionHeader_AAC(Ipp8u** ppBitStream, int* pOffset,
IppAACExtHeader* pAACExtHeader, int monoStereoFlag, int thisLayerStereo, int
monoLayerFlag, int preStereoMaxSfb, int hightstMonoMaxSfb, int winSequence);
```

### Parameters

<i>ppBitStream</i>	Double pointer to bitstream buffer. It is updated after decoding.
<i>pOffset</i>	Pointer to the offset in one byte. It is updated after decoding.
<i>monoStereoFlag</i>	Flag indicating that the current frame has mono and stereo layers.
<i>thisLayerStereo</i>	Flag indicating that the current layer is stereo.
<i>monoLayerFlag</i>	Flag indicating that the current frame has a mono layer.
<i>preStereoMaxSfb</i>	Previous stereo layer <i>maxSfb</i> .
<i>hightstMonoMaxSfb</i>	Last mono layer <i>maxSfb</i> .
<i>winSequence</i>	Window type, short or long.
<i>pAACExtHeader</i>	Pointer to the extension element header.

### Description

This function is declared in the `ippac.h` file. The function gets extension header information and extension layer information from the bitstream.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of [0,7].
<code>ippStsAacStereoLayerErr</code>	Indicates an error when <i>thisLayerStereo</i> exceeds [0,1].

`ippStsAacMonoLayerErr` Indicates an error when `monoLayerFlag` exceeds [0,1].

`ippStsAacMaxSfbErr` Indicates an error when at least one of the parameters `preStereoMaxSfb`, `hightstMonoMaxSfb` or the number of scale factor bands used in this layer exceeds [0,IPP\_AAC\_MAX\_SFB].

`ippStsAacMonoStereoErr` Indicates an error when `monoStereoFlag` exceeds [0,1].

`ippStsAacWinSeqErr` Indicates an error when `winSequence` exceeds [0,3].

## DecodePNS\_AAC

*Implements perceptual noise substitution (PNS)  
coding within individual channel stream (ICS).*

---

### Syntax

```
IppStatus ippDecodePNS_AAC_32s(Ipp32s* pSrcDstSpec, int* pSrcDstLtpFlag,
Ipp8u* pSfbCb, Ipp16s* pScaleFactor, int maxSfb, int numWinGrp, int*
pWinGrpLen, int samplingFreqIndex, int winLen, int* pRandomSeed);
```

### Parameters

<code>pSrcDstSpec</code>	Pointer to spectrum coefficients for perceptual noise substitution (PNS).
<code>pSrcDstLtpFlag</code>	Pointer to long term predict (LTP) flag.
<code>pSfbCb</code>	Pointer to the scale factor codebook.
<code>pScaleFactor</code>	Pointer to the scalefactor value.
<code>maxSfb</code>	Number of scale factor bands used in this layer.
<code>numWinGrp</code>	Number of window groups.
<code>pWinGrpLen</code>	Pointer to the length of every window group.
<code>samplingFreqIndex</code>	Sampling frequency index.
<code>winLen</code>	Window length. 1024 for long windows, 128 for short windows.
<code>pRandomSeed</code>	Random seed for PNS.

## Description

This function is declared in the `ippac.h` file. The function implements perceptual noise substitution (PNS) coding within the individual channel stream (ICS). Certain sets of spectral coefficients are derived from random vectors rather than from Huffman-coded symbols and inverse quantization process.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>maxSfb</i> exceeds <code>[0, IPP_AAC_MAX_SFB]</code> .
<code>ippStsAacSmpRateIdxErr</code>	Indicates an error when <i>samplingFreqIndex</i> exceeds <code>[0,16]</code> .
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal to 128 or 1024.

## DecodeMsPNS\_AAC

*Implements perceptual noise substitution (PNS) coding in the case of joint coding.*

## Syntax

```
IppStatus ippDecodeMsPNS_AAC_32s(Ipp32s* pSrcDstSpec, int* pSrcDstLtpFlag,
Ipp8u* pSfbCb, Ipp16s* pScaleFactor, int maxSfb, int numWinGrp, int*
pWinGrpLen, int samplingFreqIndex, int winLen, int* pRandomSeed, int channel,
Ipp8u* pMsUsed, int* pNoiseState);
```

## Parameters

<i>pSrcDstSpec</i>	Pointer to spectrum coefficients for perceptual noise substitution (PNS).
<i>pSrcDstLtpFlag</i>	Pointer to long term predict (LTP) flag.
<i>pSfbCb</i>	Pointer to the scalefactor code book.
<i>pScaleFactor</i>	Pointer to the scalefactor coefficients.
<i>maxSfb</i>	Number of maximum scalefactor band.
<i>numWinGrp</i>	Number of window groups.
<i>pWinGrpLen</i>	Pointer to the length of every window group.

<i>samplingFreqIndex</i>	Sampling frequency index.
<i>winLen</i>	Window length.
<i>pRandomSeed</i>	Random seed.
<i>channel</i>	Index of the current channel: 0 means left, 1 means right.
<i>pMsUsed</i>	Pointer to MS-used buffer in CPE structure.
<i>pNoiseState</i>	Pointer to the noise state buffer, which stores the left channel noise random seed for every scalefactor band. When <i>pMsUsed</i> [sfb]==1, the content in this buffer is used for the right channel.

## Description

This function is declared in the `ippac.h` file. The function implements perceptual noise substitution (PNS) coding in the case of joint coding. Certain sets of spectral coefficients are derived from random vectors rather than from Huffman-coded symbols and inverse quantization process.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>maxSfb</i> exceeds <code>[0, IPP_AAC_MAX_SFB]</code> .
<code>ippStsAacWinGrpErr</code>	Indicates an error when <i>maxWinGrp</i> exceeds <code>[0,8]</code> .
<code>ippStsAacSmpRateIdxErr</code>	Indicates an error when <i>samplingFreqIndex</i> exceeds <code>[0,16]</code> .
<code>ippStsAacWinLenErr</code>	Indicates an error when <i>winLen</i> is not equal to 128 or 1024.

## DecodeChanPairElt\_MP4\_AAC

*Gets channel\_pair\_element from the input bitstream.*

---

## Syntax

```
IppStatus ippDecodeChanPairElt_MP4_AAC(Ipp8u** ppBitStream, int* pOffset,
IppAACIcsInfo* pIcsInfo, IppAACChanPairElt* pChanPairElt, IppAACMainHeader*
pAACMainHeader, int predSfbMax, int audioObjectType);
```

## Parameters

<i>ppBitStream</i>	Double pointer to the current byte. It is updated after decoding.
<i>pOffset</i>	Pointer to the bit position in the byte pointed to by <i>ppBitStream</i> . Valid within 0 to 7: 0 stands for the most significant bit of the byte; 7 stands for the least significant bit of the byte.
<i>pIcsInfo</i>	Pointer to <code>IppAACIcsInfo</code> structure.
<i>pChanPairElt</i>	Pointer to <code>IppAACChanPairElt</code> structure.
<i>pAACMainHeader</i>	Pointer to the main element header.
<i>predSfbMax</i>	Maximum prediction scale factor bands.
<i>audioObjectType</i>	Audio object type indicator: 1 indicates the main type 2 indicates the LC type 6 indicates the scalable mode.

## Description

The `ippsDecodeChanPairElt_MP4_AAC` function is declared in the `ippac.h` file. This function gets the channel pair element from the input bitstream. Individual channel stream is not included.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacBitOffsetErr</code>	Indicates an error when <i>pOffset</i> is out of the range of [0,7].
<code>ippStsAacMaxSfbErr</code>	Indicates an error when <i>pIcsInfo-&gt;maxSfb</i> decoded from bitstream is greater than <code>IPP_AAC_MAX_SFB</code> , the maximum scale factor band for all sampling frequencies.

## LongTermReconstruct\_AAC

*Uses Long Term Reconstruct (LTR) to reduce signal redundancy between successive coding frames.*

---

### Syntax

```
IppStatus ippLongTermReconstruct_AAC_32s(Ipp32s* pSrcEstSpec, Ipp32s*
pSrcDstSpec, int* pLtpFlag, int winSequence, int samplingFreqIndex);
```

### Parameters

<i>pSrcDstSpec</i>	Pointer to spectral coefficients for LTP.
<i>pSrcEstSpec</i>	Pointer to the frequency domain vector.
<i>winSequence</i>	Window type, long or short.
<i>samplingFreqIndex</i>	Sampling frequency index.
<i>pLtpFlag</i>	Pointer to the LTP flag.

### Description

This function is declared in the `ippac.h` file. The function uses Long Term Reconstruct (LTR) to reduce signal redundancy between successive coding frames.

LTP is a forward adaptive predictor that is inherently less sensitive to the round-off numerical errors in the decoder or bi-errors in the transmitted spectral coefficients.

You should add the vector of decoded spectral coefficients and the corresponding frequency domain vector to get the vector of reconstructed spectral coefficients.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacSmpRateIdxErr</code>	Indicates an error when <i>samplingFreqIndex</i> exceeds [0,12].
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds [0,3].

## MDCTFwd\_AAC\_32s

Generates spectrum coefficient of PCM samples.

### Syntax

```
IppStatus ippMDCTFwd_AAC_32s(Ipp32s* pSrc, Ipp32s* pDst, Ipp32s*
pSrcDstOverlapAdd, int winSequence, int winShape, int preWinShape, Ipp32s*
pWindowedBuf);
```

### Parameters

<i>pSrc</i>	Pointer to temporal signals to do MDCT.
<i>pDst</i>	Output of MDCT, the spectral coefficients of PCM samples.
<i>pSrcDstOverlapAdd</i>	Pointer to overlap buffer. Not used for MPEG-4 AAC decoding.
<i>winSequence</i>	Window sequence indicating if the block is long or short.
<i>winShape</i>	Current window shape.
<i>preWinShape</i>	Previous window shape.
<i>pWindowedBuf</i>	Work buffer for MDCT. Must be at least 2048 words.

### Description

This function is declared in the `ippac.h` file. The function generates the spectrum coefficient of PCM samples in the MDCT Long Term Reconstruct (LTP) loop.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds [0,3].
<code>ippStsAacWinShapeErr</code>	Indicates an error when <i>preWinShape</i> exceeds [0,1].

## MDCTFwd\_AAC\_32s\_I

*Computes forward modified discrete cosine transform (MDCT) of windowed signals.*

---

### Syntax

```
IppStatus ippMDCTFwd_AAC_32s_I(Ipp32s* pSrcDst, int winSequence, int winShape, int prevWinShape, int len);
```

### Parameters

<i>pSrcDst</i>	Pointer to the source and destination vector for the in-place operation.
<i>winSequence</i>	Window sequence indicator.
<i>winShape</i>	Window shape indicator.
<i>prevWinShape</i>	Window shape indicator of the previous frame.
<i>len</i>	Number of samples in Src buffer.

### Description

The `ippMDCTFwd_AAC_32s_I` function is declared in the `ippac.h` file. This function performs forward MDCT operation. Before MDCT is calculated, the source data is multiplied by the window described with *winSequence*, *winShape*, and *prevWinShape* parameters.

This function is implemented for using in AAC decoder. It does not perform any saturation so the user should consider overflow possibility. The source and destination have different positions of the decimal point (Q format). The Q format of destination depends on the window sequence indicator.

*WinSequence* = 0 (long window): If *Src* is in Q<sub>n</sub> format then *Dst* in Q(n-12) format

*WinSequence* = 1 (long start window): If *Src* is in Q<sub>n</sub> format then *Dst* in Q(n-12) format

*WinSequence* = 2 (eight short window): If *Src* is in Q<sub>n</sub> format then *Dst* in Q(n-9) format

*WinSequence* = 3 (long stop window): If *Src* is in Q<sub>n</sub> format then *Dst* in Q(n-12) format.

For example, if *Src* is in Q<sub>14</sub> format, then *Dst* is in Q<sub>5</sub> format in the case of eight short window and in Q<sub>2</sub> in other cases.

Only *len* = 2048 is supported in the current implementation.

Below see code example 10-13 of using `ippMDCTFwd_AAC_32s_I` function.



---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the pointers passed to the function is <code>NULL</code> .
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds <code>[0,3]</code> .
<code>ippStsAacWinShapeErr</code>	Indicates an error when <i>winShape</i> or <i>prevWinShape</i> exceeds <code>[0,1]</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is not equal to 2048.

## Example 10-13 *ipp*sMDCTFwd\_AAC Usage

```
#undef ONLY_LONG_SEQUENCE

#undef LONG_START_SEQUENCE

#undef EIGHT_SHORT_SEQUENCE

#undef LONG_STOP_SEQUENCE

#undef WINDOW_SHAPE_FHG

#undef WINDOW_SHAPE_DOLBY

#define ONLY_LONG_SEQUENCE    0
#define LONG_START_SEQUENCE  1
#define EIGHT_SHORT_SEQUENCE 2
#define LONG_STOP_SEQUENCE   3
#define WINDOW_SHAPE_FHG     0
#define WINDOW_SHAPE_DOLBY   1

IppStatus mdctfwdaac(void)
{
    Ipp32s pSrcDst[2048];

    IppStatus st;

    int i;

    for (i = 0; i < 2048; i++)
        pSrcDst[i] = (Ipp32s)(cos(IPP_2PI * i / 2048) * 16384); /* Q14 */

    st = ipp_sMDCTFwd_AAC_32s_I(pSrcDst, ONLY_LONG_SEQUENCE,
                                WINDOW_SHAPE_DOLBY, WINDOW_SHAPE_DOLBY, 2048);

    /* Output will be in Q2 format */
}
```

```

printf("\n pSrcDst = ");
for (i = 0; i < 10; i++)
    printf("%i ", pSrcDst[i]);
printf("\n");
return st;
}
//Output (first 10 elements):
//   pSrcDst = 3174 -2549 -333 -300 26 -8 -3 -1 1 1

```

## EncodeTNS\_AAC

*Performs reversion of TNS in the Long Term  
Reconstruct loop in-place.*

---

### Syntax

```

IppStatus ippsEncodeTNS_AAC_32s_I(Ipp32s* pSrcDst, const int* pTnsNumFilt,
const int* pTnsRegionLen, const int* pTnsFiltOrder, const int*
pTnsFiltCoefRes, const Ipp8s* pTnsFiltCoef, const int* pTnsDirection, int
maxSfb, int profile, int samplingRateIndex, int winLen);

```

### Parameters

<i>pSrcDst</i>	On input, pointer to the spectral coefficients for the TNS encoding operation. On output, pointer to the spectral coefficients after the TNS encoding operation.
<i>pTnsNumFilt</i>	Pointer to the number of TNS filters.
<i>pTnsRegionLen</i>	Pointer to the length of TNS filter.
<i>pTnsFiltOrder</i>	Pointer to the TNS filter order.
<i>pTnsFiltCoefRes</i>	Pointer to the TNS coefficient resolution flag.
<i>pTnsFiltCoef</i>	Pointer to the TNS filter coefficients.
<i>pTnsDirection</i>	Pointer to the TNS direction flag.
<i>maxSfb</i>	Maximum scale factor number.
<i>profile</i>	Audio profile.

*samplingRateIndex*      Sampling frequency index.  
*winLen*                      Window length.

## Description

This function is declared in the `ippac.h` file. The function performs in-place reversion of TNS in the LTP loop, or Analysis Temporal Noise Shaping.

## Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error when at least one of the specified pointers is NULL.  
`ippStsTnsProfileErr`            Indicates an error when *profile* != 1.  
`ippStsAacTnsNumFiltErr`        Indicates an error when *pTnsNumFilt* exceeds [0, 1] for the short window sequence or [0, 3] for the long window sequence.  
`ippStsAacTnsLenErr`            Indicates an error when *pTnsRegionLen* exceeds [0, numSwb].  
`ippStsAacTnsOrderErr`        Indicates an error when *pTnsFiltOrder* exceeds [0, 7] for the short window sequence or [0, 12] for the long window sequence.  
`ippStsAacTnsCoefResErr`        Indicates an error when *pTnsFiltCoefRes* exceeds [3, 4].  
`ippStsAacTnsCoefErr`           Indicates an error when *pTnsFiltCoef* exceeds [-8, 7].  
`ippStsAacTnsDirectErr`        Indicates an error when *\*pTnsDirection* exceeds [0, 1].  
`ippStsAacSmpRateIdxErr`       Indicates an error when *samplingRateIndex* exceeds [0, 11].  
`ippStsAacWinLenErr`           Indicates an error when *winLen* is not equal to 128 or 1024.

## LongTermPredict\_AAC

*Gets the predicted time domain signals in the Long Term Reconstruct (LTP) loop.*

---

## Syntax

```
IppStatus ippLongTermPredict_AAC_32s(Ipp32s* pSrcTimeSignal, Ipp32s*
pDstEstTimeSignal, IppAACLtpInfo* pAACLtpInfo, int winSequence);
```

## Parameters

<i>pSrcTimeSignal</i>	Pointer to the temporal signals to be predicted in the temporary domain.
<i>pDstEstTimeSignal</i>	Pointer to the output of samples after LTP.
<i>pAACLtpInfo</i>	Pointer to the LTP information.
<i>winSequence</i>	Window type, short or long.
<i>pDstEstTimeSignal</i>	Pointer to the prediction output in time domain.

## Description

This function is declared in the `ippac.h` file. The function gets the predicted time domain signals in the LTP loop.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the pointers is <code>NULL</code> .
<code>ippStsAacWinSeqErr</code>	Indicates an error when <i>winSequence</i> exceeds <code>[0,3]</code> .

# NoiselessDecode\_AAC

*Performs noiseless decoding.*

## Syntax

```
IppStatus ippNoiselessDecode_AAC(Ipp8u** ppBitStream, int* pOffset,
IppAACMainHeader* pAACMainHeader, Ipp16s* pDstScaleFactor, Ipp32s*
pDstQuantizedSpectralCoef, Ipp8u* pDstSfbCb, Ipp8s* pDstTnsFiltCoef,
IppAACChanInfo* pChanInfo, int winSequence, int maxSfb, int commonWin, int
scaleFlag, int audioObjectType);
```

## Parameters

<i>ppBitStream</i>	Double pointer to the bitstream to be parsed. It is updated after decoding.
<i>pOffset</i>	Pointer to the offset in one byte. It is updated after decoding.
<i>pAACMainHeader</i>	Pointer to main header information. Not used for scalable objects.

	When <i>commonWin</i> == 0 && <i>scaleFlag</i> ==0, you need to decode LTP information and save it in <i>pAACMainHeader-&gt; pLtpInfo[]</i> .
<i>pChanInfo</i>	Pointer to channel information structure.
<i>windowSequence</i>	Window type, short or long.
<i>maxSfb</i>	Number of scale factor bands.
<i>commonWin</i>	Indicates if the channel pair uses the same ICS information.
<i>scaleFlag</i>	Flag indicating whether the scalable type is used.
<i>audioObjectType</i>	Audio object type indicator: 1 indicates the main type, 2 indicates the LC type, 6 indicates the scalable mode.
<i>pDstScaleFactor</i>	Pointer to the parsed scalefactor.
<i>pDstQuantizedSpectralCoef</i>	Pointer to the quantized spectral coefficients after Huffman decoding.
<i>pDstSfbCb</i>	Pointer to the scale factor codebook index.
<i>pDstTnsFiltCoef</i>	Pointer to TNS filter coefficients. Not used for scalable objects.

## Description

This function is declared in the *ippac.h* file. This is a general noiseless decoding module for MPEG-2 and MPEG-4 objects.

In case one scale factor band uses PNS in MPEG-4 AAC scalable object, *pDstScaleFactor* contains the noise energy for this scale factor band and *pDstSfbCb[sfb]* must be *NOISE\_HCB(13)*. The spectrum in this scale factor band is not necessarily Huffman-decoded, and the *pDstQuantizedSpectralCoef* of this scale factor band can be zero.

In AAC scalable object, *pDstTnsFiltCoef* and *pAACMainHeader* are not used.

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when at least one of the pointers is <i>NULL</i> .
<i>ippStsAacBitOffsetErr</i>	Indicates an error when <i>pOffset</i> is out of the range [0,7].
<i>ippStsAacComWinErr</i>	Indicates an error when <i>commonWin</i> exceeds [0,1].
<i>ippStsAacMaxSfbErr</i>	Indicates an error when <i>maxSfb</i> exceeds [0, <i>IPP_AAC_MAX_SFB</i> ].

`ippStsAacSmpRateIdxErr` Indicates an error when `pChanInfo-> samplingRateIndex` exceeds `[0,11]`.

`ippStsAacCoefValErr` Indicates an error when the quantized coefficients value pointed to by `pDstCoef` exceeds the range of `[-8191,8191]`.

## LtpUpdate\_AAC

*Performs required buffer update in the Long Term Reconstruct (LTP) loop.*

---

### Syntax

```
IppStatus ippLtpUpdate_AAC_32s(Ipp32s* pSpecVal, Ipp32s* pLtpSaveBuf, int
winSequence, int winShape, int preWinShape, Ipp32s* pWorkBuf);
```

### Parameters

<code>pSpecVal</code>	Pointer to spectral value after TNS decoder in LTP loop.
<code>pLtpSaveBuf</code>	Pointer to save buffer for LTP. Buffer length must be <code>3*frameLength</code> . The value is saved for the next frame.
<code>winSequence</code>	Window type: 0 stands for long, 1 stands for long start, 2 stands for short, 3 stands for long stop.
<code>winShape</code>	KBD or SIN window shape.
<code>preWinShape</code>	Previous window shape.
<code>pWorkBuf</code>	Work buffer for LTP update, length of <code>pWorkBuf</code> should be at least <code>2048*3 = 6144</code> words.

### Description

This function is declared in the `ippac.h` file. The function performs required buffer update in the Long Term Reconstruct (LTP) loop. This operation includes IMDCT and updating the save buffer.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

`ippStsAacWinSeqErr` Indicates an error when *winSequence* exceeds [0,3].

`ippStsAacWinShapeErr` Indicates an error when *winShape* or *preWinShape* exceeds [0,1].

## Spectral Band Replication Functions

SBR (Spectral Band Replication) is a new audio coding bandwidth extension technology, which is standardized in ISO/IEC 14496-3:2001/Amd.1:2003 [[ISO14496A](#)].

It improves the performance of low bitrate audio and speech codecs by either increasing the audio bandwidth at a given bitrate or by improving coding efficiency at a given quality level. For example, SBR can be used in conjunction with MP3 (mp3PRO), AAC (HE-AAC), CELP, HVXC to achieve a quality at 64 kbps stereo (MP3, AAC) that compares to conventional stereo at a bitrate exceeding 100 kbps.

SBR can be used with mono and stereo as well as with multichannel audio.

HE-AAC profile is combining AAC LC profile and SBR technology. HE-AAC is widely for mobile multimedia services.

In accordance with the mathematical model used by SBR in most audio materials a very high correlation is observed between the lower frequencies and the higher frequencies of a spectrum. Therefore “smart” transposition allows the transmitted signals in higher frequencies to be restructured so as to avoid coding and transmitting them as spectral data. The subsequent SBR adjustment block improves the highest reconstructed frequency.

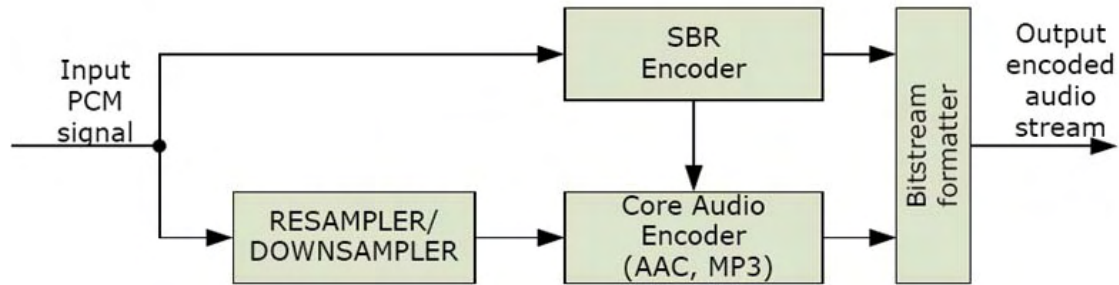
### SBR Audio Encoder Functions

This section provides a reference guide to Intel IPP for SBR audio encoder. Figures 10-4 and 10-5 show use of SBR Encoder for improvement of the core audio encoder and the common pipeline of SBR encoder.

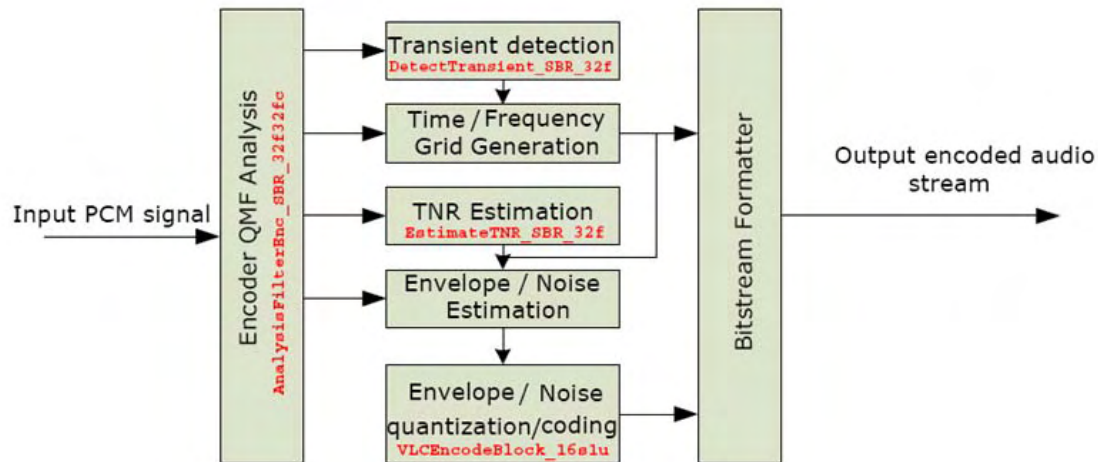
#### Figure 10-4 SBR Encoder Usage

---





**Figure 10-5 SBR Encoder Pipeline**



Intel IPP functions described in this section implement algorithms that can be used to develop HE-AAC Encoder compliant with the ETSI/3GPP [EC126] standards.

## DetectTransient\_SBR

*Estimates stationarity of signal by calculating deviation of spectrum audio signal.*

---

### Syntax

```
IppStatus ippDetectTransient_SBR_32f(const Ipp32f* pSrc, Ipp32f* pInOutThr,
Ipp32f* pDst);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector that stores the spectrum component of the audio signal.
<i>pInOutThr</i>	Pointer to the threshold of deviation.
<i>pDst</i>	Pointer to the current relative deviations.

### Description

The function `ippDetectTransient_SBR` is declared in the `ippac.h` file. This function implements transient detection algorithm described by [EC126](#) standard. The function may be used by SBR encoder to estimate stationarity of the signal by calculation of 2 thresholds. The output from this function is the *pDst* vector.

*pInOutThr* and *pDst* are static channel-dependent arrays of length 64 that need to be stored between calls of `ippDetectTransient_SBR`. On start-up, all elements in both arrays must be set to zero. These arrays are updated by the function.

Before the first call of `ippDetectTransient_SBR` (over one frame), the *pDst* vector has to be updated by the external application as follows:

$$pDst[n] = pDst[n + 32], \quad 0 \leq n < 16;$$

$$pDst[n] = 0, \quad 16 \leq n < 48.$$

The functionality of `ippDetectTransient_SBR` is described as follows:

$$m = \frac{1}{48} \sum_{n=16}^{63} pSrc[n]$$

$$\sigma = \sqrt{\frac{1}{47} \sum_{n=16}^{63} (m - pSrc[n])^2}$$

$$pInOutThr[0] = \text{MAX}(128000, 0.66 \cdot pInOutThr[0] + 0.34 \cdot \sigma)$$

```
for(n=16; n<48; n++)
    for(d=1; d<4; d++)
```

$$L = 0.5 \cdot \left\{ pSrc \left[ 2 \cdot \text{NINT} \left( \frac{n-d}{2} \right) \right] + pSrc \left[ 2 \cdot \text{NINT} \left( \frac{n-d}{2} \right) \right] + 1 \right\},$$

$$R = 0.5 \cdot \left\{ pSrc \left[ 2 \cdot \text{NINT} \left( \frac{n+d}{2} \right) \right] + pSrc \left[ 2 \cdot \text{NINT} \left( \frac{n+d}{2} \right) \right] + 1 \right\},$$

$$\begin{cases} pDst[n] = pDst[n] + \frac{R-L-pInOutThr[i]}{pInOutThr[i]}, & R-L > pThr[i] \\ pDst[n] = pDst[n], & \text{otherwise} \end{cases}$$

This function is called 64 times per one frame for each channel.

### Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when at least one of the specified pointers is NULL.

## EstimateTNR\_SBR

*Measures Tonality-to-Noise Ratio of complex QMF subband samples.*

---

### Syntax

```
ippStatus ippsestimateTNR_SBR_32f(const Ipp32fc* pSrc, Ipp32f* pTNR0, Ipp32f* pTNR1, Ipp32f* pMeanNrg);
```

### Parameters

<code>pSrc</code>	Pointer to the vector that stores the QMF-processed subband samples for one subband.
<code>pTNR0</code>	Pointer to the first TNR estimate of the input signal.
<code>pTNR1</code>	Pointer to the second TNR estimate of the input signal.
<code>pMeanNrg</code>	Pointer to mean energy of QMF-processed subband.

### Description

The function `ippsestimateTNR_SBR` is declared in the `ippac.h` file. This function implements Tonality-to-Noise Ratio algorithm described by EC126 standard. The function may be used by SBR encoder to estimate TNR of the input signal. The outputs from this function are TNR measurements of the input signal. These estimates may be used by the noise estimation algorithm, inverse filtering estimation algorithm, and sines estimation algorithm.

The functionality of `ippsestimateTNR` is described as follows:

$$\phi_1(i, j) = \sum_{n=2}^{15} pSrc[n-i+16 \cdot 1] \cdot conj(pSrc[n-j+16 \cdot 1]), \begin{cases} 0 \leq i < 3 \\ 1 \leq j < 3 \\ 0 \leq l < 1 \end{cases}$$

$$d^1 = \phi_1(2, 2) \cdot \phi_1(1, 1) - \frac{1}{1 + 10^{-6}} \cdot |\phi_1(1, 2)|^2,$$

$$\alpha_1^1 = \begin{cases} \frac{\phi_1(0, 1) \cdot \phi_1(1, 2) - \phi_1(0, 2) \cdot \phi_1(1, 1)}{d^1}, & d^1 \neq 0 \\ 0 & d^1 = 0 \end{cases}$$

$$\alpha_0^1 = \begin{cases} -\frac{\phi_1(0, 1) + \alpha_1^1 \cdot \text{conj}(\phi_1(1, 2))}{\phi_1(1, 1)}, & \phi_1(1, 1) \neq 0 \\ 0 & \phi_1(1, 1) = 0 \end{cases}$$

$$\begin{cases} p_{TNR0}[0] = f(0) \\ p_{TNR1}[0] = f(1) \end{cases}'$$

$$f(l) = \frac{\text{real}\{\alpha_0^l \cdot \text{conj}(\phi_1(0, 1)) + \alpha_1^l \cdot \text{conj}(\phi_1(0, 2))\}}{\text{real}\{\phi_1(0, 0)\} - \text{real}\{\alpha_0^l \cdot \text{conj}(\phi_1(0, 1)) + \alpha_1^l \cdot \text{conj}(\phi_1(0, 2))\}}$$

$$p_{MeanNrg}[0] = \frac{1}{2} \sum_{l=0}^1 \text{Re}\{\phi_1(0, 0)\}$$

## Return Values

`ippStsNoErr`

Indicates no error.

`ippStsNullPtrErr` Indicates an error when at least one of the specified pointers is NULL.

## AnalysisFilterEncGetSize\_SBR

*Returns size of analysis FilterSpec\_SBR specification structures, init and work buffers.*

---

### Syntax

```
IppStatus ippAnalysisFilterEncGetSize_SBR_32f(int* pSizeSpec, int* pSizeInitBuf);
```

### Parameters

*pSizeSpec* Pointer to the size (in bytes) of the analysis SBR specification structure.

*pSizeInitBuf* Pointer to the size (in bytes) of the buffer for initialization functions.

### Description

This function is declared in the `ippac.h` file. The function returns the size of the specification structure and the initialization process buffer.

### Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error when at least one of the specified pointers is NULL.

## AnalysisFilterEncInit\_SBR

*Initializes analysis specification structure.*

---

### Syntax

```
IppStatus ippAnalysisFilterEncInit_SBR_32f(IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u *pInitBuf);
```

## Parameters

<i>pFilterSpec</i>	Pointer to the analysis SBR specification structure.
<i>pInitBuf</i>	Pointer to the working buffer.

## Description

This function is declared in the `ippac.h` file. The function initializes the SBR specification structure *pFilterSpec* in the external buffer *pInitBuf*. Before calling this function, the size of the SBR specification structure must be computed by the function `ippsAnalysisFilterEncGetSize_SBR`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

# AnalysisFilterEnc\_SBR

Performs subband filtering of the input audio signal.

## Syntax

```
IppStatus ippsAnalysisFilterEnc_SBR_32f32fc(const Ipp32f* pSrc, Ipp32fc* pDst, const IppsFilterSpec_SBR_C_32f* pFilterSpec);
```

## Parameters

<i>pSrc</i>	Pointer to the input audio signal. The most recent 576 (640-64) samples must be contained in the vector <i>pSrc[i]</i> , where $i = 0, 1, \dots, 575$ , the samples associated with the current frame must be contained in the vector <i>pSrc[j]</i> , where $j = 576, \dots, 639$ .
<i>pDst</i>	Pointer to the vector; contains subband filtering samples.
<i>pFilterSpec</i>	Pointer to the initialized analysis specification structure.

### Description

This function is declared in the `ippac.h` file. The function performs subband filtering of the audio signal. Call the `ippsAnalysisFilterEnc_SBR` function 32 times per frame on each channel.

The functionality of `ippsAnalysisFilterEnc_SBR` is described as follows:

$$z[n] = pSrc[n] \cdot c[n], \quad 0 \leq n < 640$$

$$u[n] = \sum_{j=0}^4 z[n + 128 \cdot j], \quad 0 \leq n < 640$$

$$pDst[k] = \sum_{n=0}^{127} u[n] \cdot \exp\left\{\frac{\pi}{128} \cdot i \cdot \left(k + \frac{1}{2}\right)(2n+1)\right\}, \quad 0 \leq k < 64$$

Here coefficients  $c[n]$  are coefficients of the QMF bank window [ISO14496A](#)].

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when `pFilterSpec` pointer is NULL.  
`ippStsContextMatchErr` Indicates an error when the `pFilterSpec` identifier is incorrect.

## AnalysisFilterEncInitAlloc\_SBR

*Creates and initializes analysis SBR pecification structure.*

---

### Syntax

```
IppStatus ippsAnalysisFilterEncInitAlloc_SBR_32f(IppsFilterSpec_SBR_C_32f**  
ppFilterSpec);
```



### Parameters

*ppFilterSpec* Double pointer to the analysis SBR specification structure.

### Description

This function is declared in the `ippac.h` file. The function creates and initializes the analysis specification structure.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when at least one of the specified pointers is NULL.  
`ippStsMemAllocErr` Indicates an error when no memory is allocated.

## AnalysisFilterEncFree\_SBR

Closes analysis filter specification structure.

### Syntax

```
IppStatus ippAnalysisFilterEncFree_SBR_32f(IppsFilterSpec_SBR_C_32f*  
pFilterSpec);
```

### Parameters

*pFilterSpec* Pointer to the initialized analysis SBR specification structure.

### Description

This function is declared in the `ippac.h` file. The function closes the analysis specification structure *pFilterSpec* by freeing all memory associated with the specification created by [ippAnalysisFilterEncInitAlloc\\_SBR](#). Call `ippAnalysisFilterEncFree_SBR` after the transform is completed.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when *pFilterSpec* pointer is NULL.

`ippStsContextMatchErr` Indicates an error when the `pFilterSpec` identifier is incorrect.

## SBR Audio Decoder Functions

This section provides a reference guide to Intel IPP for SBR audio decoder. Figure shows the audio decoder pipeline that supports SBR technology.

**Figure 10-6 SBR Decoder Usage**

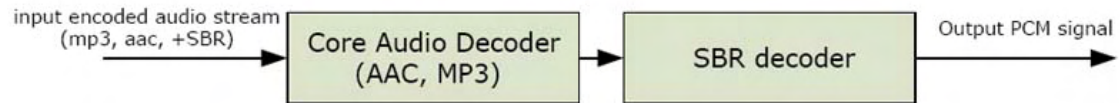


Figure shows the common pipeline of SBR decoder.

**Figure 10-7 SBR Decoder Pipeline**



In accordance with the [EC126](#) standard the SBR decoding technology can operate on complex-valued signals in the mode called High Quality (HQ) or on real-valued signals in the mode called Low Power (LP)). On the other hand, the SBR decoding technology can produce output signals with core sample rate or double sample rate. All these modes are supported by Intel IPP for SBR audio decoder.

## QMF Functions

A Quadrature Mirror Filter (QMF) bank is used by MPEG-4 SBR decoder. QMF consists of:

- Analysis Filter
- Synthesis Filter
- Synthesis Down Filter.
- Analysis Filter bank is used to split the time domain signal output from the core decoder into 32 subband signals. The output from the filterbank, that is, the subband samples, is complex-valued for HQ SBR mode or real-valued for LP SBR mode.
- Synthesis Filter bank is used to filter SBR-processed subband signals. The output from the filterbank is real-valued time domain samples. Filtering is achieved using a 64-subband Filter bank.
- Down Synthesis Filter bank is used to filter SBR-processed subband signals. The output from the filterbank is real-valued time domain samples. Filtering is achieved using a 32-subband Filter bank.

QMF filterbanks use the functions given below in low power (LP) and high quality (HQ) modes. Each mode is characterized by three function groups:

- AnalysisFilter
- SynthesisFilter
- SynthesisDownFilter.

To use the SBR\_QMF functions, initialize the corresponding filter specification structure, which contains tables of twiddle factors and internal buffer. To allocate memory and initialize the structure, call `ipps(**)FilterInitAlloc_SBR_(***)` functions, where `(**)` is a function group name, that is `Analysis`, `Synthesis` or `SynthesisDown`, and `(***)` is a flavor descriptor (for example, `_SBRHQ_32s32sc`, `_SBR_LP_32s`). Currently, this method of the specification structure initialization is implemented only for operations on fixed-point signal.

Alternatively, the structure can be initialized by functions for initialization and size getting. In that case, the general operational algorithm is as follows:

- Call `ipps(**)FilterGetSize_SBR_(***)` to get the sizes of the necessary buffers, where `(**)` is a function group name, that is `Analysis`, `Synthesis` or `SynthesisDown`, and `(***)` is a flavor descriptor (for example, `_RToC_32f32fc`, `_RToR_32f`)
- Allocate memory (external operation)
- Call `ipps(**)FilterInit_SBR_(***)` to initialize the corresponding specification structure
- Call `ipps(**)Filter_SBR_(***)` for the main operation
- Deallocate the memory (external operation).

The use of the functions described in this section is demonstrated in Intel® IPP Samples downloadable from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>.

## Memory Allocation and Initialization

These functions allocate memory and initialize SBR specification structure for real (LP) and complex (HQ) signals.

## AnalysisFilterInitAlloc\_SBR

*Allocates memory and initializes analysis SBR specification structure for real and complex signals.*

### Syntax

```
IppStatus
ippsAnalysisFilterInitAlloc_SBRHQ_32s32sc(IppsFilterSpec_SBR_C_32s**
ppFilterSpec);

IppStatus ippsAnalysisFilterInitAlloc_SBRLP_32s(IppsFilterSpec_SBR_R_32s**
ppFilterSpec);
```

### Parameters

*ppFilterSpec*                      Double pointer to the analysis SBR specification structure.

### Description

These functions are declared in the `ippac.h` file. The functions allocate memory, create, and initialize the analysis analysis SBR specification structure.

The `ippsAnalysisFilterInitAlloc_SBRHQ_32s32sc` function allocates memory and initializes the complex specification structure.

The `ippsAnalysisFilterInitAlloc_SBRLP_32s` function allocates memory and initializes the real specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppFilterSpec</i> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## SynthesisFilterInitAlloc\_SBR

*Allocates memory and initializes synthesis SBR specification structure for real and complex signals.*

### Syntax

```
IppStatus  
ippsSynthesisFilterInitAlloc_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc**  
ppFilterSpec);  
  
IppStatus ippsSynthesisFilterInitAlloc_SBRLP_32s(IppsFilterSpec_SBR_R_32s**  
ppFilterSpec);
```

### Parameters

<i>ppFilterSpec</i>	Double pointer to the synthesis SBR specification structure to be created.
---------------------	--

### Description

These functions are declared in the `ippac.h` file. The functions allocate memory, create, and initialize the synthesis SBR specification structure.

The `ippsSynthesisFilterInitAlloc_SBRHQ_32sc32s` function allocates memory and initializes the complex specification structure.

The `ippsSynthesisFilterInitAlloc_SBRLP_32s` function allocates memory and initializes the real specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppFilterSpec</i> pointer is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

## SynthesisDownFilterInitAlloc\_SBR

*Allocates memory and initializes downsample synthesis SBR specification structure for real and complex signals.*

---

### Syntax

```
IppStatus
ippsSynthesisDownFilterInitAlloc_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc**
ppFilterSpec);

IppStatus
ippsSynthesisDownFilterInitAlloc_SBRLP_32s(IppsFilterSpec_SBR_R_32s**
ppFilterSpec);
```

### Parameters

*ppFilterSpec*                      Double pointer to the downsample synthesis SBR specification structure to be created.

### Description

These functions are declared in the `ippac.h` file. The functions allocate memory, create, and initialize the downsample synthesis SBR specification structure.

The `ippsSynthesisDownFilterInitAlloc_SBRHQ_32sc32s` function allocates memory and initializes the complex specification structure.

The `ippsSynthesisDownFilterInitAlloc_SBRLP_32s` function allocates memory and initializes the real specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>ppFilterSpec</i> pointer is <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error when no memory is allocated.

### Memory Release

These functions close SBR specification structure for real and complex signals.

## AnalysisFilterFree\_SBR

*Closes analysis SBR specification structure for real and complex signals.*

---

### Syntax

```
IppStatus ippsAnalysisFilterFree_SBRHQ_32s32sc(IppsFilterSpec_SBR_C_32s*  
pFilterSpec);  
  
IppStatus ippsAnalysisFilterFree_SBRLP_32s(IppsFilterSpec_SBR_R_32s*  
pFilterSpec);
```

### Parameters

*pFilterSpec*                      Pointer to the analysis SBR specification structure.

### Description

These functions are declared in the `ippac.h` file. The functions close the analysis SBR specification structure *pFilterSpec\_SBR* by freeing all memory associated with the specification created by `ippsAnalysisFilterInitAlloc_SBR`. Call `ippsAnalysisFilterFree` after the transform is completed.

The `ippsAnalysisFilterFree_SBRHQ_32s32sc` function closes the complex SBR specification structure.

The `ippsAnalysisFilterInitAlloc_SBRLP_32s` function closes the real SBR specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFilterSpec</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFilterSpec</i> is incorrect.

## SynthesisFilterFree\_SBR

*Closes synthesis SBR specification structure for real and complex signals.*

---

### Syntax

```

IppStatus ippsSynthesisFilterFree_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc*
pFilterSpec);

IppStatus ippsSynthesisFilterFree_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
pFilterSpec);

```

### Parameters

*pFilterSpec*                      Pointer to the synthesis SBR specification structure.

### Description

These functions are declared in the `ippac.h` file. The functions close the synthesis SBR specification structure by freeing all memory associated with the specification created by [ippsSynthesisFilterInitAlloc\\_SBR](#). Call `ippsSynthesisFilterFree` after the transform is completed.

The `ippsSynthesisFilterFree_SBRHQ_32sc32s` function closes the complex SBR specification structure.

The `ippsSynthesisFilterInitAlloc_SBRLP_32s` function closes the real SBR specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pFilterSpec</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error when the specification identifier <i>pFilterSpec</i> is incorrect.



## SynthesisDownFilterFree\_SBR

*Closes downsample synthesis SBR specification structure for real and complex signals.*

---

### Syntax

```
IppStatus ippsSynthesisDownFilterFree_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc*
pFilterSpec);

IppStatus ippsSynthesisDownFilterFree_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
pFilterSpec);
```

### Parameters

*pFilterSpec*                      Pointer to the downsample synthesis SBR specification structure.

### Description

These functions are declared in the `ippac.h` file. The functions close the downsample synthesis SBR specification structure *pFilterSpec* by freeing all memory associated with the specification created by [ippsSynthesisDownFilterInitAlloc\\_SBR](#). Call `ippsSynthesisDownFilterFree` after the transform is completed.

The `ippsSynthesisDownFilterFree_SBRHQ_32sc32s` function closes the complex SBR specification structure.

The `ippsSynthesisDownFilterInitAlloc_SBRLP_32s` function closes the real SBR specification structure.

### Return Values

`ippStsNoErr`                      Indicates no error.

`ippStsNullPtrErr`               Indicates an error when the *pFilterSpec* pointer is `NULL`.

`ippStsContextMatchErr`       Indicates an error when the specification identifier *pFilterSpec* is incorrect.

### Size Getting

These functions return the size of SBR specification structures in bytes. The float-point flavors also return the sizes of the initialization process buffer and the work buffer.

## AnalysisFilterGetSize\_SBR

Returns size of analysis SBR specification structures, initialization and work buffers.

---

### Syntax

#### Case 1: Operation on float-point signal.

```
IppStatus ippsAnalysisFilterGetSize_SBR_RT0C_32f32fc(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);
```

```
IppStatus ippsAnalysisFilterGetSize_SBR_RT0C_32f(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);
```

```
IppStatus ippsAnalysisFilterGetSize_SBR_RT0R_32f(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);
```

#### Case 2: Operation on fixed-point signal.

```
IppStatus ippsAnalysisFilterGetSize_SBRHQ_32s32sc(int* pSizeSpec);
```

```
IppStatus ippsAnalysisFilterGetSize_SBRLP_32s(int* pSizeSpec);
```

### Parameters

<i>pSizeSpec</i>	Pointer to the size (in bytes) of the analysis SBR specification structure.
<i>pSizeInitBuf</i>	Pointer to the size (in bytes) of the buffer for initialization functions.
<i>pSizeWorkBuf</i>	Pointer to the size (in bytes) of the work buffer.

### Description

These functions are declared in the `ippac.h` file. The functions return the size of the specification structure. The floating point flavors also return the sizes of the initialization process buffer and the work buffer.

`ippsAnalysisFilterGetSize_SBR_RT0C_32f32fc` returns the specified sizes for `ippsAnalysisFilter_SBR_RT0C_32f32fc_D2L`.

`ippsAnalysisFilterGetSize_SBR_RT0C_32f` returns the specified sizes for `ippsAnalysisFilter_SBR_RT0C_32f_D2L`.

`ippsAnalysisFilterGetSize_SBR_RT0R_32f` returns the specified sizes for `ippsAnalysisFilter_SBR_RT0R_32f_D2L`.

`ippsAnalysisFilterGetSize_SBRHQ_32s32sc` returns the size of specification structure for `ippsAnalysisFilter_SBRHQ_32s32sc`.

`ippsAnalysisFilterGetSize_SBRLP_32s` returns the size of specification structure for `ippsAnalysisFilter_SBRLP_32s`.

AAC decoder included into IPP Samples uses the functions `ippsAnalysisFilterGetSize_SBR_RToC_32f` and `ippsAnalysisFilterGetSize_SBR_RToR_32f` in the float-point version, and the functions `ippsAnalysisFilterGetSize_SBRHQ_32s32sc` and `ippsAnalysisFilterGetSize_SBRLP_32s` in the fixed-point version. See [introduction](#) to this section.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

## SynthesisFilterGetSize\_SBR

*Returns size of synthesis SBR specification structure, initialization and work buffers.*

---

### Syntax

#### Case 1: Operation on float-point signal.

```
IppStatus ippsSynthesisFilterGetSize_SBR_CToR_32fc32f(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);
```

```
IppStatus ippsSynthesisFilterGetSize_SBR_CToR_32f(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);
```

```
IppStatus ippsSynthesisFilterGetSize_SBR_RToR_32f(int* pSizeSpec, int* pSizeInitBuf, int* pSizeWorkBuf);
```

#### Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisFilterGetSize_SBRHQ_32sc32s(int* pSizeSpec);
```

```
IppStatus ippsSynthesisFilterGetSize_SBRLP_32s(int* pSizeSpec);
```

### Parameters

<code>pSizeSpec</code>	Pointer to the size (in bytes) of the synthesis SBR specification structure.
------------------------	--

<i>pSizeInitBuf</i>	Pointer to the size (in bytes) of the buffer for initialization functions.
<i>pSizeWorkBuf</i>	Pointer to the size (in bytes) of the work buffer.

## Description

These functions are declared in the `ippac.h` file. The functions return the size of the synthesis SBR specification structure. The floating point flavors also return the sizes of the initialization process buffer and the work buffer.

`ippsSynthesisFilterGetSize_SBR_CToR_32fc32f` returns the specified sizes for `ippsSynthesisFilter_SBR_CToR_32fc32f_D2L`.

`ippsSynthesisFilterGetSize_SBR_CToR_32f` returns the specified sizes for `ippsSynthesisFilter_SBR_CToR_32f_D2L`.

`ippsSynthesisFilterGetSize_SBR_RToR_32f` returns the specified sizes for `ippsSynthesisFilter_SBR_RToR_32f_D2L`.

`ippsSynthesisFilterGetSize_SBRHQ_32sc32s` returns the size of specification structure for `ippsSynthesisFilter_SBRHQ_32sc32s`.

`ippsSynthesisFilterGetSize_SBRLP_32s` returns the size of specification structure for `ippsSynthesisFilter_SBRLP_32s`.

AAC decoder included into IPP Samples uses the functions `ippsSynthesisFilterGetSize_SBR_RToC_32f` and `ippsSynthesisFilterGetSize_SBR_RToR_32f` in the float-point version, and functions `ippsSynthesisFilterGetSize_SBRHQ_32sc32s` and `ippsSynthesisFilterGetSize_SBRLP_32s` in the fixed-point version. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

## SynthesisDownFilterGetSize\_SBR

Returns size of downsample synthesis SBR specification structure, initialization and work buffers.

---

### Syntax

#### Case 1: Operation on float-point signal.

```
IppStatus ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f(int* pSizeSpec,  
int* pSizeInitBuf, int* pSizeWorkBuf);  
  
IppStatus ippsSynthesisDownFilterGetSize_SBR_CToR_32f(int* pSizeSpec, int*  
pSizeInitBuf, int* pSizeWorkBuf);  
  
IppStatus ippsSynthesisDownFilterGetSize_SBR_RToR_32f(int* pSizeSpec, int*  
pSizeInitBuf, int* pSizeWorkBuf);
```

#### Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisDownFilterGetSize_SBRHQ_32sc32s(int* pSizeSpec);  
IppStatus ippsSynthesisDownFilterGetSize_SBRLP_32s(int* pSizeSpec);
```

### Parameters

<i>pSizeSpec</i>	Pointer to the size (in bytes) of the downsample synthesis SBR specification structure.
<i>pSizeInitBuf</i>	Pointer to the size (in bytes) of the buffer for initialization functions.
<i>pSizeWorkBuf</i>	Pointer to the size (in bytes) of the work buffer.

### Description

These functions are declared in the `ippac.h` file. The functions return the size of the downsample synthesis SBR specification structure. The floating point flavors also return the sizes of the initialization process buffer and the work buffer.

`ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f` returns the specified sizes for `ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L`.

`ippsSynthesisDownFilterGetSize_SBR_CToR_32f` returns the specified sizes for `ippsSynthesisDownFilter_SBR_CToR_32f_D2L`.

`ippsSynthesisDownFilterGetSize_SBR_RTOr_32f` returns the specified sizes for `ippsSynthesisDownFilter_SBR_RTOr_32f_D2L`.

`ippsSynthesisDownFilterGetSize_SBRHQ_32sc32s` returns the size of specification structure for `ippsSynthesisDownFilter_SBRHQ_32sc32s`.

`ippsSynthesisDownFilterGetSize_SBRLP_32s` returns the size of specification structure for `ippsSynthesisDownFilter_SBRLP_32s`.

AAC decoder included into IPP Samples use the functions `ippsSynthesisDownFilterGetSize_SBR_RTOr_32f` and `ippsSynthesisDownFilterGetSize_SBR_RTOr_32f` in the float-point version, and functions `ippsSynthesisDownFilterGetSize_SBRHQ_32sc32s` and `ippsSynthesisDownFilterGetSize_SBRLP_32s` in the fixed-point version. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

## Initialization

These functions initialize the filter SBR specification structures.

## AnalysisFilterInit\_SBR

*Initializes analysis SBR specification structure.*

---

### Syntax

#### Case 1: Operation on float-point signal.

```
IppStatus ippsAnalysisFilterInit_SBR_RTOr_32f32fc(IppsFilterSpec_SBR_C_32fc**
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

```
IppStatus ippsAnalysisFilterInit_SBR_RTOr_32f(IppsFilterSpec_SBR_C_32f**
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

```
IppStatus ippsAnalysisFilterInit_SBR_RTOr_32f(IppsFilterSpec_SBR_R_32f**
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

#### Case 2: Operation on fixed-point signal.

```
IppStatus ippsAnalysisFilterInit_SBRHQ_32s32sc(IppsFilterSpec_SBR_C_32sc*
pFilterSpec);
```

```
IppStatus ippsAnalysisFilterInit_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
pFilterSpec);
```

### Parameters

<i>ppFilterSpec</i>	Double pointer to the analysis SBR specification structure.
<i>pMemSpec</i>	Pointer to the area for the analysis SBR specification structure.
<i>pInitBuf</i>	Pointer to the initialization buffer.

### Description

These functions are declared in the `ippac.h` file. The functions initialize the analysis structures:

`ippsAnalysisFilterInit_SBR_RToC_32f32fc` initializes the specification structure for [ippsAnalysisFilter\\_SBR\\_RToC\\_32f32fc\\_D2L](#).

`ippsAnalysisFilterInit_SBR_RToC_32f` initializes the specification structure for [ippsAnalysisFilter\\_SBR\\_RToC\\_32f\\_D2L](#).

`ippsAnalysisFilterInit_SBR_RToR_32f` initializes the specification structure for [ippsAnalysisFilter\\_SBR\\_RToR\\_32f\\_D2L](#).

`ippsAnalysisFilterInit_SBRHQ_32s32sc` initializes the specification structure for [ippsAnalysisFilter\\_SBRHQ\\_32s32sc](#).

`ippsAnalysisFilterInit_SBRLP_32s` initializes the specification structure for [ippsAnalysisFilter\\_SBRLP\\_32s](#).

AAC decoder included into IPP Samples uses the functions `ippsAnalysisFilterInit_SBR_RToC_32f` and `ippsAnalysisFilterInit_SBR_RToR_32f` in the float-point version, and functions `ippsAnalysisFilterInit_SBRHQ_32s32sc` and `ippsAnalysisFilterInit_SBRLP_32s` in the fixed-point version. See [introduction](#) to this section.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is <code>NULL</code> .

## SynthesisFilterInit\_SBR

*Initializes synthesis specification structure.*

---

### Syntax

#### Case 1: Operation on float-point signal.

```
ippStatus ippsSynthesisFilterInit_SBR_CToR_32fc32f(IppsFilterSpec_SBR_C_32fc**  
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

```
ippStatus ippsSynthesisFilterInit_SBR_CToR_32f(IppsFilterSpec_SBR_C_32f**  
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

```
ippStatus ippsSynthesisFilterInit_SBR_RToR_32f(IppsFilterSpec_SBR_R_32f**  
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);
```

#### Case 2: Operation on fixed-point signal.

```
ippStatus ippsSynthesisFilterInit_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc*  
pFilterSpec);
```

```
ippStatus ippsSynthesisFilterInit_SBRLP_32s(IppsFilterSpec_SBR_R_32s*  
pFilterSpec);
```

### Parameters

<i>ppFilterSpec</i>	Double pointer to the synthesis SBR specification structure.
<i>pMemSpec</i>	Pointer to the area for the synthesis SBR specification structure.
<i>pInitBuf</i>	Pointer to the initialization buffer.

### Description

These functions are declared in the `ippac.h` file. The functions initialize the synthesis SBR structures:

`ippsSynthesisFilterInit_SBR_CToR_32fc32f` initializes the specification structure for [ippsSynthesisFilter\\_SBR\\_CToR\\_32fc32f\\_D2L](#).

`ippsSynthesisFilterInit_SBR_CToR_32f` initializes the specification structure for [ippsSynthesisFilter\\_SBR\\_CToR\\_32f\\_D2L](#).

`ippsSynthesisFilterInit_SBR_RToR_32f` initializes the specification structure for [ippsSynthesisFilter\\_SBR\\_RToR\\_32f\\_D2L](#).



`ippsSynthesisFilterInit_SBRHQ_32sc32s` initializes the specification structure for `ippsSynthesisFilter_SBRHQ_32sc32s`.

`ippsSynthesisFilterInit_SBRLP_32s` initializes the specification structure for `ippsSynthesisFilter_SBRLP_32s`.

AAC decoder included into IPP Samples uses the functions `ippsSynthesisFilterInit_SBR_RToC_32f` and `ippsSynthesisFilterInit_SBR_RToR_32f` in the float-point version, and functions `ippsSynthesisFilterInit_SBRHQ_32sc32s` and `ippsSynthesisFilterInit_SBRLP_32s` in the fixed-point version. See [introduction](#) to this section.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

## SynthesisDownFilterInit\_SBR

*Initializes downsample synthesis SBR specification structure.*

---

### Syntax

#### Case 1: Operation on float-point signal.

```

IppStatus
ippsSynthesisDownFilterInit_SBR_CToR_32fc32f(IppsFilterSpec_SBR_C_32fc**
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);

IppStatus ippsSynthesisDownFilterInit_SBR_CToR_32f(IppsFilterSpec_SBR_C_32f**
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);

IppStatus ippsSynthesisDownFilterInit_SBR_RToR_32f(IppsFilterSpec_SBR_R_32f**
ppFilterSpec, Ipp8u* pMemSpec, Ipp8u* pInitBuf);

```

#### Case 2: Operation on fixed-point signal.

```

IppStatus ippsSynthesisDownFilterInit_SBRHQ_32sc32s(IppsFilterSpec_SBR_C_32sc*
pFilterSpec);

IppStatus ippsSynthesisDownFilterInit_SBRLP_32s(IppsFilterSpec_SBR_R_32s*
pFilterSpec);

```

## Parameters

<i>ppFilterSpec</i>	Double pointer to the downsample synthesis SBR specification structure.
<i>pMemSpec</i>	Pointer to the area for the downsample synthesis SBR specification structure.
<i>pInitBuf</i>	Pointer to the initialization buffer.

## Description

These functions are declared in the `ippac.h` file. The functions initialize the down sample synthesis structures:

`ippsSynthesisDownFilterInit_SBR_CToR_32fc32f` initializes the specification structure for `ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L`.

`ippsSynthesisDownFilterInit_SBR_CToR_32f` initializes the specification structure for `ippsSynthesisDownFilter_SBR_CToR_32f_D2L`.

`ippsSynthesisDownFilterInit_SBR_RToR_32f` initializes the specification structure for `ippsSynthesisDownFilter_SBR_RToR_32f_D2L`.

`ippsSynthesisDownFilterInit_SBRHQ_32sc32s` initializes the specification structure for `ippsSynthesisDownFilter_SBRHQ_32sc32s`.

`ippsSynthesisDownFilterInit_SBRLP_32s` initializes the specification structure for `ippsSynthesisDownFilter_SBRLP_32s`.

AAC decoder included into IPP Samples uses the functions `ippsSynthesisDownFilterInit_SBR_RToC_32f` and `ippsSynthesisDownFilterInit_SBR_RToR_32f` in the float-point version, and functions `ippsSynthesisDownFilterInit_SBRHQ_32sc32s` and `ippsSynthesisDownFilterInit_SBRLP_32s` in the fixed-point version. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

## QMF Main Functions

These functions perform main operations: transform time domain samples from the core decoder to the SBR-processed subband signals and vice versa.

## AnalysisFilter\_SBR

*Transforms time domain signal output from the core decoder into frequency subband signals.*

### Syntax

#### Case 1: Operation on float-point signal.

```
IppStatus ippsAnalysisFilter_SBR_RToC_32f32fc_D2L(const Ipp32f* pSrc, Ipp32fc* pDst[], const Ipp32f* pSbrTableWindowDown, int numLoop, int offset, int kx, const IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);
```

```
IppStatus ippsAnalysisFilter_SBR_RToC_32f_D2L(const Ipp32f* pSrc, Ipp32f* pDstRe[], Ipp32f* pDstIm[], const Ipp32f* pSbrTableWindowDown, int numLoop, int offset, int kx, const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

```
IppStatus ippsAnalysisFilter_SBR_RToR_32f_D2L(const Ipp32f* pSrc, Ipp32f* pDst[], const Ipp32f* pSbrTableWindowDown, int numLoop, int offset, int kx, const IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

#### Case 2: Operation on fixed-point signal.

```
IppStatus ippsAnalysisFilter_SBRHQ_32s32sc(const Ipp32s* pSrc, Ipp32sc* pDst, int kx, const IppsFilterSpec_SBR_C_32sc* pFilterSpec);
```

```
IppStatus ippsAnalysisFilter_SBRLP_32s(const Ipp32s* pSrc, Ipp32s* pDst, int kx, const IppsFilterSpec_SBR_R_32s* pFilterSpec);
```

### Parameters

<i>pSrc</i>	Pointer to the input audio signal.
<i>pDst</i>	Pointer or array of pointers to the subband samples.
<i>pDstRe</i>	Array of pointers to the real parts of subband samples.
<i>pDstIm</i>	Array of pointersto the imaginary parts of subband samples.
<i>pSbrTableWindowDown</i>	Pointer to the window table that is used by decoder SBR Analysis Filter. This parameter is used only for float-point cases.
<i>offset</i>	Desired displacement in number of rows when the matrix <i>pDst</i> is calculated; must be greater than or equal to 0.

<i>numLoop</i>	Parameter equal to 32 if frame size of the core decoded signal is 1024, and to 30 if frame size of the core decoded signal is 960.
<i>kx</i>	First SBR subband in the SBR range [0..32].
<i>pFilterSpec</i>	Pointer to the analysis SBR specification structure.
<i>pWorkBuf</i>	Pointer to the work buffer.

## Description

These functions are declared in the `ippac.h` file. The functions filter input signal according to the *pFilterSpec* specification parameters.

### Float-point case.

The function `ippsAnalysisFilter_SBR_RToC_32f32fc_D2L` uses the real input data type and complex output data type, processes the input real array *pSrc* and stores the result in the complex matrix *pDst*.

- The sizes of the work buffer, init buffer, and specification structure must be calculated by the function `ippsAnalysisFilterGetSize_SBR_RToC_32f32fc` beforehand.
- The analysis filter SBR specification structure must be initialized by the function `ippsAnalysisFilterInit_SBR_RToC_32f32fc`. Deallocate *pInitBuf* after completing initialization.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32fc` type.

The function `ippsAnalysisFilter_SBR_RToC_32f_D2L` uses the real data type and stores the result separately in *pDstRe* matrix and *pDstIm* matrix.

- The sizes of the work buffer, init buffer, and specification structure must be calculated by the function `ippsAnalysisFilterGetSize_SBR_RToC_32f` beforehand.
- The analysis filter SBR specification structure must be initialized by the function `ippsAnalysisFilterInit_SBR_RToC_32f`. Deallocate *pInitBuf* after completing initialization.
- *pFilterSpec* specification parameter has to be of `IppsFilterSpec_SBR_C_32f` type.

The functionality of `ippsAnalysisFilter_SBR_RToC_32f32fc_D2L` and `ippsAnalysisFilter_SBR_RToC_32f_D2L` is described as follows:

### Step 1. Buffer Updating

$$x[n] = x[n - 32], \quad 320 > n \geq 32$$

$$x[n] = pSrc[1 \times 32 + 31 - n], \quad 31 \geq n \geq 0$$

**Step 2. Polyphase Filtering**

$$u[n] = \sum_{j=0}^4 x[n+j \times 64] \times pSbrTableWindowDown[n+j \times 64], \quad 0 \leq n < 64$$

**Step 3. Special Fourier Transform**

$$pDst[l+offset][k] = 2 \sum_{n=0}^{63} u[n] \times \exp\left[i \frac{\pi}{64} \times \left(k + \frac{1}{2}\right) \left(2n - \frac{1}{2}\right)\right], \quad 0 \leq k < 32$$

**Step 4. Clearing Upper Part of Spectrum.**

$$pDst[l+offset][k] = 0, \quad k \leq k < 32$$

Note that Steps 1 through 4 are repeated for  $0 \leq l < numLoop$  and

$$pSrcTableWindowDown[n] = c[2n], \quad n = 0, \dots, 320,$$

where coefficients  $c[i]$  are coefficients of the QMF bank window from [ISO14496](#).

The function `ippsAnalysisFilter_SBR_RToR_32f_D2L` uses the real data type, processes the input real array `pSrc` and stores the result in the real matrix `pDst`.

- The sizes of the work buffer, init buffer, and specification structure must be calculated by the function `ippsAnalysisFilterGetSize_SBR_RToR_32f` beforehand.
- The analysis filter SBR specification structure must be initialized by the function `ippsAnalysisFilterInit_SBR_RToR_32f`. Deallocate `pInitBuf` after completing initialization.
- `pFilterSpec` specification parameter has to be of `IppsFilterSpec_SBR_R_32f` type.

The functionality of `ippsAnalysisFilter_SBR_RToR_32f_D2L` can be described in the same way as the above `AnalysisFilter_SBR_D2L` functions but Step 3 changes as follows:

**Step 3. Special Fourier Transform**

$$pDst[l+offset][k] = 2 \sum_{n=0}^{63} u[n] \times \cos\left[\frac{\pi}{64} \times \left(k + \frac{1}{2}\right)(2n - 96)\right], \quad 0 \leq k < 32.$$

The functions `ippsAnalysisFilter_SBR_RToC_32f_D2L` and `ippsAnalysisFilter_SBR_RToR_32f_D2L` are used in the float-point version of AAC decoder included into Intel IPP Samples. See [introduction](#) to this section.

## Fixed-point case.

The functions with integer values do not saturate any intermediate or output data so the user must watch closely for overload.

In `ippsAnalysisFilter_SBRHQ_32s32sc` and `ippsAnalysisFilter_SBRLP_32s` the source and destination have different positions of decimation point (Q format). If `pSrc` has Q(N) format, `pDst` has Q(N-6) format. The recommended Q of the source is Q12.

The functionality of `ippsAnalysisFilter_SBRHQ_32s32sc` is described as follows:

### Step 1. Buffer Updating

$$x[n] = x[n - 32], \quad 32 \leq n < 320$$

$$x[n] = pSrc[31 - n], \quad 0 \leq n < 32$$

### Step 2. Polyphase Filtering

$$u[n] = \sum_{j=0}^4 x[n + j \times 64] \times pSbrTableWindowDown[n + j \times 64], \quad 0 \leq n < 64$$

### Step 3. Special Fourier Transform

$$pDst[l+offset][k] = 2 \sum_{n=0}^{63} u[n] \times \exp\left[i \frac{\pi}{64} \times \left(k + \frac{1}{2}\right)\left(2n - \frac{1}{2}\right)\right], \quad 0 \leq k < 32$$

#### Step 4. Clearing Upper Part of Spectrum

$$pDst[k] = 0, kx \leq k < 32$$

The functionality of `ippsAnalysisFilter_SBRLP_32s` can be described in the same way as the above function but Step 3 changes as follows:

#### Step 3. Special Fourier Transform

$$pDst[k] = \sum_{n=0}^{63} u[n] \times \cos \left[ \frac{\pi i}{64} \left( k + \frac{1}{2} \right) (2n - 96) \right], \quad 0 \leq k < 32.$$

$pWin[n] = c[2n]$  and  $c[i]$  are coefficients of the QMF bank window from [ISO14496A](#).

The functions `ippsAnalysisFilter_SBRHQ_32s32sc` and `ippsAnalysisFilter_SBRLP_32s` are used in the fixed-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

#### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the identifier <code>pFilterSpec</code> is incorrect.
<code>ippStsSizeErr</code>	Indicates an error when $kx$ is less than 0 or greater than 32.

## SynthesisFilter\_SBR

*Transforms SBR-processed subband signals into time domain samples.*

---

#### Syntax

##### Case 1: Operation on float-point signal.

```

IppStatus ippsSynthesisFilter_SBR_CToR_32fc32f_D2L (const Ipp32fc* pSrc[],
Ipp32f* pDst, const Ipp32f* pSbrTableWindow, int numLoop, const
IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);

```

```
IppStatus ippsSynthesisFilter_SBR_CToR_32f_D2L(const Ipp32f* pSrcRe[], const
Ipp32f* pSrcIm[], Ipp32f* pDst, const Ipp32f* pSbrTableWindow, int numLoop,
const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

```
IppStatus ippsSynthesisFilter_SBR_RToR_32f_D2L(const Ipp32f* pSrc[], Ipp32f*
>pDst, const Ipp32f* pSbrTableWindow, int numLoop, const
IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);
```

## Case 2: Operation on fixed-point signal.

```
IppStatus ippsSynthesisFilter_SBRHQ_32sc32s(const Ipp32sc* pSrc, Ipp32s*
pDst, const IppsFilterSpec_SBR_C_32sc* pFilterSpec);
```

```
IppStatus ippsSynthesisFilter_SBRLP_32s(const Ipp32s* pSrc, Ipp32s* pDst,
const IppsFilterSpec_SBR_R_32s* pFilterSpec);
```

## Parameters

<i>pSrc</i>	Pointer or array of pointers to the SBR-processed subband signals.
<i>pSrcRe</i>	Array of pointers to the real parts of SBR-processed subband signals.
<i>pSrcIm</i>	Array of pointers to the imaginary parts of SBR-processed subband signals.
<i>pDst</i>	Pointer to the output vector with time domain output samples.
<i>pSbrTableWindowDown</i>	Pointer to the window table that is used by decoder SBR synthesis filter. This parameter is used only for float-point flavors.
<i>numLoop</i>	Parameter equal to 32 if frame size of the core decoded signal is 1024, and to 30 if frame size of the core decoded signal is 960.
<i>pFilterSpec</i>	Pointer to the synthesis SBR specification structure.
<i>pWorkBuf</i>	Pointer to the work buffer.

## Description

These functions are declared in the `ippac.h` file. The functions transform SBR-processed subband signals into time domain samples according to *pFilterSpec* specification parameters.

## Float-point case.



The function `ippsSynthesisFilter_SBR_CToR_32fc32f_D2L` uses the complex input data type and real output data type, processes the input complex array of pointers `pSrc` and stores the result in the real vector `pDst`.

- The sizes of the work buffer, initialization buffer, and specification structure must be calculated by the function `ippsSynthesisFilterGetSize_SBR_CToR_32fc32f` beforehand.
- The synthesis filter SBR specification structure must be initialized by the function `ippsSynthesisFilterInit_SBR_CToR_32fc32f`. Deallocate `pInitBuf` after completing initialization.
- `pFilterSpec` specification parameter has to be of `IppsFilterSpec_SBR_C_32fc` type.

The function `ippsSynthesisFilter_SBR_CToR_32f_D2L` uses the real data type and stores the result separately in `pSrcRe` matrix and `pSrcIm` matrix.

- The sizes of the work buffer, initialization buffer, and specification structure must be calculated by the function `ippsSynthesisFilterGetSize_SBR_CToR_32f` beforehand.
- The synthesis filter SBR specification structure must be initialized by the function `ippsSynthesisFilterInit_SBR_CToR_32f`. Deallocate `pInitBuf` after completing initialization.
- `pFilterSpec` specification parameter has to be of `IppsFilterSpec_SBR_C_32f` type.

The functionality of `ippsSynthesisFilter_SBR_CToR_32fc32f_D2L` and `ippsSynthesisFilter_SBR_CToR_32f_D2L` is described as follows:

#### Step 1. Buffer Updating

$$v[n] = v[n - 128], \quad 1280 > n \geq 128$$

#### Step 2. Special Fourier Transform

$$v[n] = \frac{1}{64} \sum_{k=0}^{63} \operatorname{Re} \left\{ pSrc[1][k] \times \exp \left[ i \frac{\pi}{128} \times \left( k + \frac{1}{2} \right) (2n - 255) \right] \right\}, \quad 0 \leq n < 128$$

## Step 3. Polyphase Filtering

$$w[128 \times n + k] = v[256 \times n + k] \times pSbrTableWindow[128 \times n + k]$$

$$\left\{ \begin{array}{l} 0 \leq n < 5 \\ 128 \leq n < 64 \end{array} \right. \quad w[128 \times n + 64 + k] = v[256 \times n + 192 + k] \times pSbrTableWindow[128 \times n + 64 + k]$$

## Step 4. Output Vector Updating

$$pDst[64 \times l + k] = \sum_{n=0}^9 w[64 \times n + k], \quad 0 \leq k < 64.$$

Note that Steps 1 through 4 are repeated for  $0 \leq l < numLoop$  and

$$pSrcTableWindowDown[n] = c[n], \quad n = 0, \dots, 640,$$

where coefficients  $c[i]$  are coefficients of the QMF bank window from [ISO14496](#).

The function `ippsSynthesisFilter_SBR_RToR_32f_D2L` uses the real data type, processes the input real matrix `pSrc` and stores the result in the real array `pDst`.

- The sizes of the work buffer, initialization buffer, and specification structure must be calculated by the function `ippsSynthesisFilterGetSize_SBR_RToR_32f` beforehand.
- The synthesis filter SBR specification structure must be initialized by the function `ippsSynthesisFilterInit_SBR_RToR_32f`. Deallocate `pInitBuf` after completing initialization.
- `pFilterSpec` specification parameter has to be of `IppsFilterSpec_SBR_R_32f` type.

The functionality of `ippsSynthesisFilter_SBR_RToR_32f` can be described in the same way as the above `SynthesisFilter_SBR` functions but Step 2 changes as follows:

## Step 2. Special Fourier Transform

### Step 2. Special Fourier Transform

$$v[n] = \frac{1}{32} \sum_{k=0}^{63} pSrc[l][k] \times \cos\left[\frac{\pi}{128} \times \left(k + \frac{1}{2}\right)(2n - 64)\right], \quad 0 \leq n < 128.$$

The functions `ippsSynthesisFilter_SBR_CToR_32f_D2L` and `ippsSynthesisFilter_SBR_RToR_32f_D2L` are used in the float-point version of AAC decoder included into Intel IPP Samples. See [introduction](#) to this section.

**Fixed-point case.**

The functions with integer values do not saturate any intermediate or output data so the user should watch closely for overload.

In `ippsSynthesisFilter_SBRHQ_32sc32s` and `ippsSynthesisFilter_SBRLP_32s` the source and destination have different positions of decimation point (Q format). If  $pSrc$  has Q(N) format,  $pDst$  has Q(N-6) format. The recommended Q of the source is Q12.

The functionality of `ippsSynthesisFilter_SBRHQ_32sc32s` is described as follows:

**Step 1. Buffer Updating**

$$v[n] = v[n - 128], \quad 128 \leq n < 1280$$

**Step 2. Special Fourier Transform**

$$v[n] = \sum_{k=0}^{63} \operatorname{Re} \left\{ pSrc[k] \times \exp \left[ \frac{\pi i}{128} \times \left( k + \frac{1}{2} \right) (2n - 255) \right] \right\}, \quad 0 \leq n < 128$$

**Step 3. Polyphase Filtering**

$$\begin{aligned} w[128 \times n + k] &= v[256 \times n + k] \times pWin[128 \times n + k] \\ w[128 \times n + 64 + k] &= v[256 \times n + 192 + k] \times pWin[128 \times n + 64 + k] \end{aligned}, \quad \begin{cases} 0 \leq n < 5 \\ 0 \leq k < 64 \end{cases}$$

**Step 4. Output Vector Updating**

$$pDst[k] = \sum_{n=0}^9 w[64 \times n + k], \quad 0 \leq k < 64,$$

$pWin[n] = c[n]$  are coefficients of the QMF bank window from [ISO14496A](#).

The functionality of `ippsSynthesisFilter_SBRLP_32s` can be described in the same way as the above function but Step 2 changes as follows:

## Step 2. Special Fourier Transform

$$v[n] = \sum_{k=0}^{63} pSrc[k] \times \cos\left[\frac{\pi i}{128} \times \left(k + \frac{1}{2}\right)(2n - 64)\right], \quad 0 \leq n < 128.$$

The functions `ippsSynthesisFilter_SBRHQ_32sc32s` and `ippsSynthesisFilter_SBRLP_32s` are used in the fixed-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the identifier <code>pFilterSpec</code> is incorrect.

## SynthesisDownFilter\_SBR

*Transforms SBR-processed subband signals into time domain samples and performs downsampling at the same time.*

---

## Syntax

### Case 1: Operation on float-point signal.

```

IppStatus ippsSynthesisFilter_SBR_CToR_32fc32f_D2L (const Ipp32fc* pSrc[],
Ipp32f* pDst, const Ipp32f* pSbrTableWindow, int numLoop, const
IppsFilterSpec_SBR_C_32fc* pFilterSpec, Ipp8u* pWorkBuf);

IppStatus ippsSynthesisFilter_SBR_CToR_32f_D2L(const Ipp32f* pSrcRe[], const
Ipp32f* pSrcIm[], Ipp32f* pDst, const Ipp32f* pSbrTableWindow, int numLoop,
const IppsFilterSpec_SBR_C_32f* pFilterSpec, Ipp8u* pWorkBuf);

IppStatus ippsSynthesisFilter_SBR_RToR_32f_D2L(const Ipp32f* pSrc[], Ipp32f*
>pDst, const Ipp32f* pSbrTableWindow, int numLoop, const
IppsFilterSpec_SBR_R_32f* pFilterSpec, Ipp8u* pWorkBuf);

```

**Case 2: Operation on fixed-point signal.**

```

IppStatus ippsSynthesisFilter_SBRHQ_32sc32s(const Ipp32sc* pSrc, Ipp32s*
pDst, const IppsFilterSpec_SBR_C_32sc* pFilterSpec);

IppStatus ippsSynthesisFilter_SBR_LP_32s(const Ipp32s* pSrc, Ipp32s* pDst,
const IppsFilterSpec_SBR_R_32s* pFilterSpec);

```

**Parameters**

<i>pSrc</i>	Pointer or array of pointers to the SBR-processed subband signals.
<i>pSrcRe</i>	Array of pointers to the real parts of SBR-processed subband signals.
<i>pSrcIm</i>	Array of pointers to the imaginary parts of SBR-processed subband signals.
<i>pDst</i>	Pointer to the output vector with time domain output samples.
<i>pSbrTableWindowDown</i>	Pointer to the window table that is used by decoder SBR synthesis filter. This parameter is used only for float-point flavors.
<i>numLoop</i>	Parameter equal to 32 if frame size of the core decoded signal is 1024, and to 30 if frame size of the core decoded signal is 960.
<i>pFilterSpec</i>	Pointer to the synthesis SBR specification structure.
<i>pWorkBuf</i>	Pointer to the work buffer.

**Description**

These functions are declared in the `ippac.h` file. The functions transform SBR-processed subband signals into time domain samples and perform downsampling at the same time according to *pFilterSpec* specification parameters.

**Float-point case.**

The function `ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L` uses the complex input data type and real output data type, processes the input complex matrix *pSrc* and stores the result in the real array *pDst*.

- The sizes of the work buffer, initialization buffer, and specification structure must be calculated by the function `ippsSynthesisDownFilterGetSize_SBR_CToR_32fc32f` beforehand.

- The synthesis filter SBR specification structure must be initialized by the function `ippsSynthesisDownFilterInit_SBR_CToR_32fc32f`. Deallocate `pInitBuf` after completing initialization.
- `pFilterSpec` specification parameter has to be of `IppsFilterSpec_SBR_C_32fc` type.

The function `ippsSynthesisDownFilter_SBR_CToR_32f_D2L` uses the real data type and stores the result separately in `pSrcRe` matrix and `pSrcIm` matrix.

- The sizes of the work buffer, initialization buffer, and specification structure must be calculated by the function `ippsSynthesisDownFilterGetSize_SBR_CToR_32f` beforehand.
- The synthesis filter SBR specification structure must be initialized by the function `ippsSynthesisDownFilterInit_SBR_CToR_32f`. Deallocate `pInitBuf` after completing initialization.
- `pFilterSpec` specification parameter has to be of `IppsFilterSpec_SBR_C_32f` type.

The functionality of `ippsSynthesisDownFilter_SBR_CToR_32fc32f_D2L` and `ippsSynthesisDownFilter_SBR_CToR_32f_D2L` is described as follows:

### Step 1. Buffer Updating

$$v[n] = v[n - 64], \quad 640 > n \geq 64$$

### Step 2. Special Fourier Transform

$$v[n] = \frac{1}{64} \sum_{k=0}^{31} \operatorname{Re} \left\{ pSrc[1][k] \times \exp \left[ i \frac{\pi}{64} \times \left( k + \frac{1}{2} \right) (2n - 127) \right] \right\}, \quad 0 \leq n < 64$$

### Step 3. Polyphase Filtering

$$\begin{aligned} w[64 \times n + k] &= v[128 \times n + k] \times pSbrTableWindow[128 \times n + k] \\ w[64 \times n + 32 + k] &= v[128 \times n + 96 + k] \times pSbrTableWindow[128 \times n + 96 + k] \end{aligned} \quad , \quad \begin{cases} 0 \leq n < 5 \\ 0 \leq k < 32 \end{cases}$$

### Step 4. Output Vector Updating

$$pDst[32 \times l + k] = \sum_{n=0}^9 w[32 \times n + k], \quad 0 \leq k < 32.$$

Note that Steps 1 through 4 are repeated for  $0 \leq l < numLoop$  and

$pSrcTableWindowDown[n] = c[2n], n = 0, \dots, 320,$

where coefficients  $c[i]$  are coefficients of the QMF bank window from [ISO14496](#).

The function `ippsSynthesisDownFilter_SBR_RToR_32f_D2L` uses the real data type, processes the input real matrix  $pSrc$  and stores the result in the real array  $pDst$ .

- The sizes of the work buffer, initialization buffer, and specification structure must be calculated by the function `ippsSynthesisDownFilterGetSize_SBR_RToR_32f` beforehand.
- The synthesis filter SBR specification structure must be initialized by the function `ippsSynthesisDownFilterInit_SBR_RToR_32f`. Deallocate  $pInitBuf$  after completing initialization.
- $pFilterSpec$  specification parameter has to be of `IppsFilterSpec_SBR_R_32f` type.

The functionality of `ippsSynthesisDownFilter_SBR_RToR_32f_D2L` can be described in the same way as the above `SynthesisFilter_SBR` functions but Step 2 changes as follows:

#### Step 2. Special Fourier Transform

$$v[n] = \frac{1}{32} \sum_{k=0}^{31} pSrc[l][k] \times \cos\left[\frac{\pi}{64} \times \left(k + \frac{1}{2}\right)(2n - 32)\right], \quad 0 \leq n < 64.$$

The functions `ippsSynthesisDownFilter_SBR_CToR_32f_D2L` and `ippsSynthesisDownFilter_SBR_RToR_32f_D2L` are used in the float-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.

#### Fixed-point case.

The functions with integer values do not saturate any intermediate or output data so the user should watch closely for overload.

In `ippsSynthesisDownFilter_SBRHQ_32sc32s` and `ippsSynthesisDownFilter_SBRLP_32s` the source and destination have different positions of decimation point (Q format). If  $pSrc$  has Q(N) format,  $pDst$  has Q(N-6) format. The recommended Q of the source is Q12.

The functionality of `ippsSynthesisDownFilter_SBRHQ_32sc32s` is described as follows:

## Step 1. Buffer Updating

$$v[n] = v[n - 64], \quad 64 \leq n < 640$$

## Step 2. Special Fourier Transform

$$v[n] = \sum_{k=0}^{31} \operatorname{Re} \left\{ pSrc[k] \times \exp \left[ \frac{\pi i}{128} \times \left( k + \frac{1}{2} \right) (2n - 127.5) \right] \right\}, \quad 0 \leq n < 64$$

## Step 3. Polyphase Filtering

$$\begin{aligned} w[64 \times n + k] &= v[128 \times n + k] \times pWin[64 \times n + k] \\ w[64 \times n + 32 + k] &= v[128 \times n + 96 + k] \times pWin[64 \times n + 32 + k] \end{aligned} \quad , \quad \begin{cases} 0 \leq n < 5 \\ 0 \leq k < 32 \end{cases}$$

## Step 4. Output Vector Updating

$$pDst[k] = \sum_{n=0}^9 w[32 \times n + k], \quad 0 \leq k < 32,$$

Here  $pWin[n] = c[2n]$  and  $c[n]$  are coefficients of the QMF bank window from [ISO14496A](#).

The functionality of `ippsSynthesisDownFilter_SBRLP_32s` can be described in the same way as the above function but Step 2 changes as follows:

## Step 2. Special Fourier Transform

$$v[n] = \sum_{k=0}^{31} pSrc[k] \times \cos \left[ \frac{\pi i}{128} \times \left( k + \frac{1}{2} \right) (2n - 32) \right], \quad 0 \leq n < 64.$$

The functions `ippsSynthesisDownFilter_SBRHQ_32sc32s` and `ippsSynthesisDownFilter_SBRLP_32s` are used in the fixed-point version of AAC decoder included into IPP Samples. See [introduction](#) to this section.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error when the identifier <i>pFilterSpec</i> is incorrect.

## Prediction Functions

SBR prediction functions provide inverse filtering for correct reconstruction of high frequency transmitted signals.

## PredictCoef\_SBR

*Obtains prediction filter coefficients using covariance method.*

---

### Syntax

```
IppStatus ippsPredictCoef_SBR_C_32fc_D2L(const Ipp32fc* pSrc[], Ipp32fc*
pAlpha0, Ipp32fc* pAlpha1, int k0, int len);

IppStatus ippsPredictCoef_SBR_C_32f_D2L(const Ipp32f* pSrcRe[], const Ipp32f*
pSrcIm[], Ipp32f* pAlpha0Re, Ipp32f* pAlpha0Im, Ipp32f* pAlpha1Re, Ipp32f*
pAlpha1Im, int k0, int len);

IppStatus ippsPredictCoef_SBR_R_32f_D2L(const Ipp32f* pSrc[], Ipp32f* pAlpha0,
Ipp32f* pAlpha1, int k0, int len);
```

### Parameters

<i>pSrc</i>	Array of pointers to the source real or complex matrix ([40][32]) holding the low frequency QMF-processed subband signals.
<i>pSrcRe</i>	Array of pointers ([40][32]) that contains real parts of the low frequency SBR-processed subband signals.
<i>pSrcIm</i>	Array of pointers ([40][32]) that contains imaginary parts of the low frequency SBR-processed subband signals.
<i>pAlpha0, pAlpha1</i>	Pointers to the prediction filter coefficients used by high frequency filter.

*pAlpha0Re, pAlpha1Re* Pointers to the arrays that contain real parts of the corresponding complex filter coefficients.

*pAlpha0Im, pAlpha1Im* Pointers to the arrays that contain imaginary parts of the corresponding complex filter coefficients.

*k0* First QMF subband in the *f\_master* table.

*len* Autocorrelation size.

## Description

These functions are declared in the `ippac.h` file. The function `ippsPredictCoef_SBR` obtains prediction filter coefficients using covariance method.

The function `ippsPredictCoef_SBR_C_32fc` uses the complex input data type and complex output data type, processes the input complex matrix *pSrc* and stores the result in complex arrays *pAlpha0* and *pAlpha1*.

The function `ippsPredictCoef_SBR_C_32f` uses the real input and output data type, storing the complex input data separately in matrices *pSrcRe* and *pSrcIm* and stores complex output data separately in *pAlpha0Re, pAlpha0Im, pAlpha1Re, pAlpha1Im*.

The functions `ippsPredictCoef_SBR_C_32fc` and `ippsPredictCoef_SBR_C_32f` are used in high quality SBR mode.

The function `ippsPredictCoef_SBR_R_32f` uses the real input and output data type. This function is used in low power SBR mode.

The prediction filter coefficients are obtained with the covariance method using the following formula:

$$\phi_k(i, j) = \sum_{n=0}^{len-1} pSrc[n-i+2][k] \cdot conj(pSrc[n-j+2][k]),$$

$$0 \leq i < 3, 1 \leq j < 3, 0 \leq k < k_0.$$

The coefficients *pAlpha0*( *k* ) and *pAlpha1* ( *k* ) are calculated as follows:

$$d(k) = \phi_k(2, 2) \cdot \phi_k(1, 1) - \frac{1}{1 + \varepsilon_{inv}} |\phi_k(1, 2)|^2,$$

$$pAlpha1(k) = \begin{cases} \frac{\phi_k(0, 1) \times \phi_k(1, 2) - \phi_k(0, 2) \times \phi_k(1, 1)}{d(k)}, & d(k) \neq 0 \\ 0, & d(k) = 0 \end{cases}$$

$$pAlpha0(k) = \begin{cases} \frac{\phi_k(0, 1) \times pAlpha1(k) \times conj(\phi_k(1, 2))}{\phi_k(1, 1)}, & \phi_k(1, 1) \neq 0 \\ 0, & \phi_k(1, 1) = 0 \end{cases}$$

The functions `ippsPredictCoef_SBR_C_32f_D2L` and `ippsPredictCoef_SBR_R_32f_D2L` are used in the float-point version of AAC decoder included into Intel IPP Samples. See [introduction](#) to this section.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

## PredictOneCoef\_SBR

*Obtains a prediction filter coefficient using covariance method.*

### Syntax

```

IppStatus ippsPredictOneCoef_SBRHQ_32sc_D2L(const Ipp32sc* pSrc[], Ipp32sc*
pAlpha0, Ipp32sc* pAlpha1, int k, int lenCorr);

IppStatus ippsPredictOneCoef_SBRLP_32s_D2L(const Ipp32s* pSrc[], Ipp32s*
pAlpha0, Ipp32s* pAlpha1, Ipp32s* pRefCoef, int k, int lenCorr, int flag);

```

### Parameters

<i>pSrc</i>	Array of pointers to the source real or complex matrix holding the low frequency QMF-processed subband signals.
-------------	---

<i>pAlpha0, pAlpha1</i>	Pointers to the prediction filter coefficients used by high frequency filter.
<i>pRefCoef</i>	Pointer to the reflection coefficient that is used by the aliasing detection algorithm (LP mode only).
<i>k</i>	Number of the input matrix column. This parameter is used for calculating prediction coefficients.
<i>lenCorr</i>	Autocorrelation size, this parameter is used for calculating prediction coefficients.
<i>flag</i>	Flag that specifies the result of calculation (LP mode only). If it is NULL (0), then only the reflection coefficient is calculated. Otherwise both prediction and reflection coefficients are calculated.

## Description

These functions are declared in the `ippac.h` file and obtain a prediction filter coefficient using covariance method. The `ippsPredictOneCoef_SBRHQ_32sc_D2L` function is used in high quality SBR mode and the `ippsPredictOneCoef_SBRLP_32s_D2L` function is used in low power SBR mode.

Prediction and reflection coefficients have the same position of decimation point (Q format), independent of Q format of *pSrc*. It is Q(29). Q format of *pSrc* can be arbitrary but the recommended value is Q(5).

The prediction filter coefficient is obtained with the covariance method using the following formula:

$$\phi(i, j) = \sum_{n=0}^{lenCorr-1} pSrc[n-i+2][k] \cdot conj(pSrc[n-j+2][k]),$$

$$d = \phi(2, 2) \cdot \phi(1, 1) - \frac{1}{1 + \varepsilon_{inv}} |\phi(1, 2)|^2,$$

$$pAlpha1[0] = \begin{cases} \frac{\phi(0, 1) \times \phi(1, 2) - \phi(0, 2) \times \phi(1, 1)}{d}, & d \neq 0 \\ 0, & d = 0 \end{cases}$$

$$pAlpha0[0] = \begin{cases} \frac{\phi(0, 1) \times pAlpha[0] \times \text{conj}(\phi(1,1))}{\phi(1,1)}, & \phi(1,1) \neq 0 \\ 0, & \phi(1,1) = 0 \end{cases}$$

$$pRefCoef[0] = \begin{cases} \min(\max(-\frac{\phi(0, 1)}{\phi(1,1)}, -1), 1), & \phi(1,1) \neq 0 \\ 0, & \phi(1,1) = 0 \end{cases}$$

$$\begin{cases} pAlpha0[0] = 0 \\ pAlpha1[0] = 0 \end{cases}, \text{ if } |pAlpha0[0]| + |pAlpha1[0]| \geq 4$$

The functions `ippsPredictOneCoef_SBRHQ_32sc_D2L` and `ippsPredictOneCoef_SBRLP_32s_D2L` are used in the fixed-point version of AAC decoder included into Intel IPP Samples. See [introduction](#) to this section.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when at least one of the specified pointers is NULL.

## Parametric Stereo Functions

Parametric Stereo (PS) coding is a technique to efficiently code a stereo audio signal as a monaural signal plus a small amount of stereo parameters (from a few kbit/s for medium quality and up to about 9 kbit/s for higher quality). PS technique is a part of the MPEG-4 audio parametric coding scheme published by the ISO/IEC 14496-3:2001/Amd.2:2004 [[ISO14496B](#)].

You can combine the PS technique with the SBR technology, for example, HEAACv2 audio codec.

The High Efficiency AAC v2 Profile (HEAAC v2) combines the AAC LC profile, the SBR technology, and the Parametric Stereo technique. HE-AAC v2 is widely used for mobile multimedia services.

## PS Audio Decoder Functions

This section describes Intel IPP functions for PS decoding operations.

### AnalysisFilter\_PS

*Increases frequency resolution of the first lower subbands by hybrid filtering.*

---

#### Syntax

```
IppStatus ippAnalysisFilter_PS_32fc_D2(const Ipp32fc* pSrc, Ipp32fc
dst[32][12], IppAC_PS_DEC_ANALYSIS config);
```

#### Parameters

<i>pSrc</i>	Pointer to the lower QMF subband.
<i>dst</i>	Output matrix that contains hybrid filtered subband samples.
<i>config</i>	Flag that describes configuration of hybrid filtering.

#### Description

The function is declared in the `ippac.h` file. This function performs hybrid filtering of the lower QMF subbands to obtain a higher frequency resolution. Depending on the stereo bands configuration and a number of subbands, five filter configurations are defined as follows:

`IPPAC_PC_CONF0` - indicates that 10 or 20 stereo band configuration is applied, and the number of QMF subbands is 2.

`IPPAC_PC_CONF1` - indicates that 34 stereo band configuration is applied, and the number of QMF subbands is 4.

`IPPAC_PC_CONF2` - indicates that 10 or 20 stereo band configuration is applied, and the number of QMF subbands is 8.

`IPPAC_PC_CONF3` - indicates that 34 stereo band configuration is applied, and the number of QMF subbands is 8.

IPPAC\_PC\_CONF4 - indicates that 34 stereo band configuration is applied, and the number of QMF subbands is 12.

Note that length of the vector  $pSrc$  is independent of filter configurations and is always 45.

The functionality of `ippsAnalysisFilter_PS_32fc_D2` is described as follows:

$$dst[sample][q] = \sum_{n=0}^{12} pSrc[sample+n] \cdot g_{conf}[n] \cdot \exp\left\{\frac{2\pi(n-6)(q+0.5)}{Q_{conf}}\right\}$$

In the case of IPPAC\_PC\_CONF0,  $\exp\left\{\frac{2\pi(n-6)(q+0.5)}{Q_{conf}}\right\}$  is replaced with

$$\cos\left\{\frac{2\pi(n-6)}{Q_{conf}}q\right\}, \quad \text{where}$$

$$\begin{cases} 0 \leq sample < 32 \\ 0 \leq q < Q_{conf} \end{cases}$$

$g_{conf}[n]$  represents the filter coefficients that correspond to the respective filtering configuration.

$Q_{conf}$  is the number of subbands.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <code>config</code> is incorrect.

## DTS Audio Coding Functions

DTS (Digital Theater Systems) is a multi-channel digital surround sound format, also known as DTS Coherent Acoustics. DTS Coherent Acoustics standard is published by the ETSI (ETSI 102 114 v1.2.1) [ETSI02].

The core DTS specification supports up to 6 audio channels (5.1 configuration) with the sampling frequency between 8kHz and 48kHz. Further specification extensions allow for an additional 2 channels and/or additional high frequency data to be included. Audio is encoded by splitting it into 32 subbands, which are then encoded using ADPCM.

Table lists the DTS macro and constant definitions.

**Table 10-9. DTS Macro and Constant Definitions**

Global Macro Name	Definition	Notes
IPPAC_DTS_NONPERFECT	0	nonperfect reconstruction filter bank
IPPAC_DTS_PERFECT	1	perfect reconstruction filter bank

### DTS Audio Decoder

This section describes Intel IPP functions for DTS decoding operations.

### SynthesisFilterInit\_DTS

*Initializes DTS synthesis filter specification structure.*

---

#### Syntax

```
IppStatus ippsSynthesisFilterInit_DTS_32f(IppsFilterSpec_DTS_32f*  
pFilterSpec);
```

#### Parameters

<i>pFilterSpec</i>	Pointer to the DTS synthesis filter specification structure.
--------------------	--



### Description

This function is declared in the `ippac.h` file. The function creates and initializes a DTS synthesis filter specification structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the specified pointer is <code>NULL</code> .

## SynthesisFilterInitAlloc\_DTS

*Allocates memory for DTS synthesis filter specification structure and initializes it.*

---

### Syntax

```
IppStatus ippSynthesisFilterInitAlloc_DTS_32f(IppsFilterSpec_DTS_32f**  
ppFilterSpec);
```

### Parameters

<code>ppFilterSpec</code>	Double pointer to the DTS synthesis filter specification structure.
---------------------------	---

### Description

This function is declared in the `ippac.h` file. The function allocates memory for a DTS synthesis filter specification structure, creates, and initializes this structure.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the specified pointer is <code>NULL</code> .

## SynthesisFilterGetSize\_DTS

*Returns size of DTS synthesis filter specification structure.*

---

### Syntax

```
IppStatus ippSynthesisFilterGetSize_DTS_32f(int* pSizeSpec);
```

## Parameters

*pSizeSpec* Pointer to the size value of the DTS synthesis filter specification structure.

## Description

This function is declared in the `ippac.h` file. The function gets size of the DTS synthesis filter specification structure and stores the result in *pSizeSpec*.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the specified pointer is `NULL`.

## SynthesisFilterFree\_DTS

*Frees memory allocated for DTS synthesis filter specification structure.*

---

## Syntax

```
IppStatus ippSynthesisFilterFree_DTS_32f(IppsFilterSpec_DTS_32f*
pFilterSpec);
```

## Parameters

*pFilterSpec* Pointer to the DTS synthesis filter specification structure.

## Description

This function is declared in the `ippac.h` file. The function frees memory that `ippSynthesisFilterInitAlloc_DTS` allocates for the DTS synthesis filter specification structure.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when the specified pointer is `NULL`.  
`ippStsContextMatchErr` Indicates an error when the *pFilterSpec* identifier is incorrect.

## SynthesisFilter\_DTS

*Transforms QMF DTS-processed subband signals into time domain samples.*

---

### Syntax

```
IppStatus ippSynthesisFilter_DTS_32f(const Ipp32f* pSrc, Ipp32f* pDst, int
filtType, const IppsFilterSpec_DTS_32f* pFilterSpec);
```

### Parameters

<i>pSrc</i>	Array of pointers to the QMF DTS-processed subband signals.
<i>pDst</i>	Pointer to the output vector holding time domain output samples.
<i>filtType</i>	Flag that indicates the filter bank: <code>IPPAC_DTS_PERFECT</code> - indicates perfect reconstruction filter bank, <code>IPPAC_DTS_NONPERFECT</code> - indicates nonperfect reconstruction filter bank.
<i>pFilterSpec</i>	Pointer to the DTS synthesis filter specification structure.

### Description

This function is declared in the `ippac.h` file. The function transforms QMF DTS-processed subband signals into time domain samples. For each input block (32 subband samples), this function generates an output sequence of 32 PCM samples in the vector pointed to by *pDst*. The *filtType* flag indicates the filter bank to be used: `IPPAC_DTS_PERFECT` indicates perfect reconstruction filter bank, `IPPAC_DTS_NONPERFECT` - nonperfect reconstruction filter bank.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when any of the specified pointers is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error when <i>filtType</i> is incorrect.
<code>ippStsContextMatchErr</code>	Indicates an error when the <i>pFilterSpec</i> identifier is incorrect.

---

---

# String Functions

# 11

This chapter describes the Intel® IPP functions that perform operations with a text. First part describes the functions for simple string manipulation. Second part contains functions that perform more sophisticated matching operation using patterns of the regular expressions.

The full list of functions in this group is given in Table 11-1 below.

**Table 11-1 Intel IPP String Functions**

Function Base Name	Operation
String Manipulation Functions	
<code>Find</code> , <code>FindRev</code>	Looks for the first occurrence of the substring matching the specified string.
<code>FindC</code> , <code>FindRevC</code>	Looks for the first occurrence of the specified element within the source string.
<code>FindCAny</code> , <code>FindRevCAny</code>	Looks for the first occurrence of any element of the specified array within the source string.
<code>Insert</code>	Inserts a string into another string.
<code>Remove</code>	Removes a specified number of elements from the string.
<code>Compare</code>	Compares two strings of the fixed length.
<code>CompareIgnoreCase</code> , <code>CompareIgnoreCaseLatin</code>	Compares two strings of the fixed length ignoring case.
<code>Equal</code>	Compares two strings of the fixed length for equality.
<code>TrimC</code>	Deletes all occurrences of a specified symbol both in the beginning and in the end of the string.
<code>TrimCAny</code>	Deletes all occurrences of any of the specified symbols both in the beginning and in the end of the source string.
<code>TrimStartCAny</code> , <code>TrimEndtCAny</code>	Deletes all occurrences of any of the specified symbols either in the beginning or in the end of the source string, respectively.
<code>ReplaceC</code>	Replaces all occurrences of a specified element in the source string with another element.
<code>Uppercase</code> , <code>UppercaseLatin</code>	Converts alphabetic characters of a string to all uppercase symbols.
<code>Lowercase</code> , <code>LowercaseLatin</code>	Converts alphabetic characters of a string to all lowercase symbols.
<code>Hash</code>	Calculates a hash value for the string.
<code>Concat</code>	Concatenates several strings together.
<code>ConcatC</code>	Concatenates several strings together and inserts symbol delimiters between them.
<code>SplitC</code>	Splits source string into separate parts.
Functions for Work with Regular Expressions	

Function Base Name	Operation
<a href="#">RegExpInitAlloc</a>	Allocates memory and initializes the structure for processing matching operation with regular expressions.
<a href="#">RegExpFree</a>	Frees the memory allocated for a regular expression state structure.
<a href="#">RegExpInit</a>	Initializes the structure for processing matching operation with regular expressions.
<a href="#">RegExpGetSize</a>	Computes the size of the regular expression state structure.
<a href="#">RegExpSetMatchLimit</a>	Sets the value of the <i>matchLimit</i> parameter.
<a href="#">RegExpFind</a>	Looks for the occurrences of the substrings matching the specified regular expression.
<a href="#">RegExpSetFormat</a>	Sets source encoding format for given compiled pattern.
<a href="#">ConvertUTF</a>	Converts UTF16BE or UTF16LE format to UTF8 and vice versa.
<a href="#">RegExpMultiGetSize</a>	Computes the size the multiple patterns search engine memory.
<a href="#">RegExpMultiInit</a>	Initializes state structure for multi patterns search engine.
<a href="#">RegExpMultiInitAlloc</a>	Allocates and initializes the multiple patterns search engine memory.
<a href="#">RegExpMultiFree</a>	Frees memory allocated for the state structure for the multiply patterns search engine.
<a href="#">RegExpMultiAdd</a>	Adds the specified pattern to the multiple patterns database.
<a href="#">RegExpMultiDelete</a>	Deletes the specified patterns from the multiple pattern database.
<a href="#">RegExpMultiModify</a>	Modifies the specified patterns in the multiple pattern database.
<a href="#">RegExpMultiFind</a>	Looks for the occurrences of the substrings matching the multiple patterns.
<a href="#">RegExpReplaceGetSize</a>	Calculates the size of the state structure for the find-replace operation.
<a href="#">RegExpReplaceInit</a>	Initialize the state structure for the find-replace operation.
<a href="#">RegExpReplace</a>	Performs find and replace operation.

## String Manipulation

This section describes the Intel IPP functions that perform operations with strings. Intel IPP string functions do not consider zero as the end of the string, but require that the length of the string (number of elements) be specified explicitly. Overlapping of the strings is not supported (for not in-place operations). Intel IPP string functions operate with two data types, `Ipp8u` and `Ipp16u`.

### Find, FindRev

*Looks for the first occurrence of the substring matching the specified string.*

---

#### Syntax

```
IppStatus ippsFind_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind, int lenFind, int* pIndex); IppStatus ippsFind_Z_8u(const Ipp8u* pSrcZ, const Ipp8u* pFindZ, int* pIndex);
```

```
IppStatus ippsFind_16u(const Ipp16u* pSrc, int len, const Ipp16u* pFind, int lenFind, int* pIndex); IppStatus ippsFind_Z_16u(const Ipp16u* pSrc, const Ipp16u* pFind, int* pIndex);
```

```
IppStatus ippsFindRev_8u(const Ipp8u* pSrc, int len, const Ipp8u* pFind, int lenFind, int* pIndex);
```

```
IppStatus ippsFindRev_16u(const Ipp16u* pSrc, int len, const Ipp16u* pFind, int lenFind, int* pIndex);
```

#### Parameters

<i>pSrc</i>	Pointer to the source string.
<i>pSrcZ</i>	Pointer to the zero-ended source string.
<i>len</i>	Number of elements in the source string.
<i>pFind</i>	Pointer to the reference string.
<i>pFindZ</i>	Pointer to the zero-ended reference string.
<i>lenFind</i>	Number of elements in the reference string.
<i>pIndex</i>	Pointer to the result index.

## Description

The functions `ippsFind` and `ippsFindRev` are declared in the `ippch.h` file. These functions search through the source string `pSrc` for a substring of elements that match the specified reference string `pFind`. Starting point of the first occurrence of the matching substring is stored in `pIndex`. If no matching substring is found, then `pIndex` is set to `-1`.

The function flavor `ippsFind_Z` operates with the zero-ended source and reference strings. The function `ippsFindRev` searches the source string in the reverse direction. The search is case-sensitive.

Code example 11-1 below shows how to use the function `ippsFind_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>len</code> or <code>lenFind</code> is negative.



### Example 11-1 Using the functions `ippsFind`, `ippsFindC`, `ippsFindRevCAny`

```

Ipp8u string[] = "abracadabra";

*/
           -----
           0123456789a

*/

Ipp8u substring[] = "abra";
Ipp8u any_of [] = "ftr";
int index;

ippsFind_8u( string, sizeof (string) - 1, substring, sizeof (substring) - 1, &index );
printf ( "ippsFind_8u returned index = %d.\n", index );
ippsFindC_Z_8u( string, " c ", &index );
printf ( "ippsFind_Z_8u returned index = %d.\n", index );
ippsFindRevCAny_8u( string, sizeof (string) - 1, any_of , sizeof ( any_of ) - 1, &index );
printf ( "ippsFindRevCAny_8u returned index = %d.\n", index );

```

Output:

```

ippsFind_8u returned index = 0.
ippsFind_Z_8u returned index = 4.
ippsFindRevCAny_8u returned index = 9.

```

## FindC, FindRevC

*Looks for the first occurrence of the specified element within the source string.*

---

### Syntax

```

IppStatus ippsFindC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind, int*
pIndex); IppStatus ippsFindC_Z_8u(const Ipp8u* pSrcZ, Ipp8u valFind, int*
pIndex);

```

```

IppStatus ippsFindC_16u(const Ipp16u* pSrc, int len, Ipp16u valFind, int*
pIndex); IppStatus ippsFindC_Z_16u(const Ipp16u* pSrcZ, Ipp16u valFind, int*
pIndex);

IppStatus ippsFindRevC_8u(const Ipp8u* pSrc, int len, Ipp8u valFind, int*
pIndex);

IppStatus ippsFindRevC_16u(const Ipp16u* pSrc, int len, Ipp16u valFind, int*
pIndex);

```

## Parameters

<i>pSrc</i>	Pointer to the source string.
<i>pSrcZ</i>	Pointer to the source zero-ended string.
<i>len</i>	Number of elements in the source string.
<i>valFind</i>	Value of the specified element.
<i>pIndex</i>	Pointer to the result index.

## Description

The functions `ippsFindC` and `ippsFindRevC` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for the first occurrence of the specified element with the value *valFind*. The position of this element is stored in *pIndex*. If no matching element is found, then *pIndex* is set to -1. The function flavor `ippsFindC_Z` operates with the zero-ended source string. The function `ippsFindRevC` searches the source string in the reverse direction. The search is case-sensitive.

Code [example 11-1](#) above shows how to use the function `ippsFindC_Z_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

## FindCAny, FindRevCAny

*Looks for the first occurrence of any element of the specified array within the source string.*

---

### Syntax

```
IppStatus ippsFindCAny_8u(const Ipp8u* pSrc, int len, const Ipp8u* pAnyOf,
int lenAnyOf, int* pIndex);

IppStatus ippsFindCAny_16u(const Ipp16u* pSrc, int len, const Ipp16u* pAnyOf,
int lenAnyOf, int* pIndex);

IppStatus ippsFindRevCAny_8u(const Ipp8u* pSrc, int len, const Ipp8u* pAnyOf,
int lenAnyOf, int* pIndex);

IppStatus ippsFindRevCAny_16u(const Ipp16u* pSrc, int len, const Ipp16u*
pAnyOf, int lenAnyOf, int* pIndex);
```

### Parameters

<i>pSrc</i>	Pointer to the source string.
<i>len</i>	Number of elements in the source string.
<i>pAnyOf</i>	Pointer to the array containing reference elements.
<i>lenAnyOf</i>	Number of elements in the array.
<i>pIndex</i>	Pointer to the result index.

### Description

The functions `ippsFindCAny` and `ippsFindRevCAny` are declared in the `ippch.h` file. These functions search through the source string *pSrc* for the first occurrence of any reference element from the specified array *pAnyOf*. The position of this element is stored in *pIndex*. If no matching element is found, then *pIndex* is set to -1. The function `ippsFindRevCAny` searches the source string in the reverse direction. The search is case-sensitive.

Code [example 11-1](#) above shows how to use the function `ippsFindCAny_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .

`ippStsLengthErr` Indicates an error condition if `len` or `lenAnyOf` is negative.

## Insert

*Inserts a string into another string.*

---

### Syntax

```
IppStatus ippInsert_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pInsert,
int insertLen, Ipp8u* pDst, int startIndex);
```

```
IppStatus ippInsert_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
pInsert, int insertLen, Ipp16u* pDst, int startIndex);
```

```
IppStatus ippInsert_8u_I(const Ipp8u* pInsert, int insertLen, Ipp8u* pSrcDst,
int* pSrcDstLen, int startIndex);
```

```
IppStatus ippInsert_16u_I(const Ipp16u* pInsert, int insertLen, Ipp16u*
pSrcDst, int* pSrcDstLen, int startIndex);
```

### Parameters

<code>pSrc</code>	Pointer to the source string.
<code>srcLen</code>	Number of elements in the source string.
<code>pInsert</code>	Pointer to the string to be inserted.
<code>insertLen</code>	Number of elements in the string to be inserted.
<code>pDst</code>	Pointer to the destination string.
<code>pSrcDst</code>	Pointer to the source and destination string for in-place operation.
<code>pSrcDstLen</code>	Pointer to the number of elements in the source and destination string for in-place operation.
<code>startIndex</code>	Index of the insertion point.

### Description

The function `ippInsert` is declared in the `ippch.h` file. This function inserts the string `pInsert` containing `insertLen` elements into a source string `pSrc` of length `srcLen`. Insertion position is specified by `startIndex`. The result is stored in the `pDst`.

The in-place flavors of `ippInsert` insert the string `pInsert` in the source string `pSrcDst` and store the result in the destination string `pSrcDst`.

Example 11-2 below shows how to use the function `ippsInsert_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if one of the <i>srcLen</i> , <i>insertLen</i> , <i>pSrcSdtLen</i> , <i>startIndex</i> is negative, or <i>startIndex</i> is greater than <i>srcLen</i> or <i>pSrcDstLen</i> .

### Example 11-2. Using the Functions `ippsInsert` and `ippsRemove`

```
Ipp8u string[] = " 1st string part 2nd string part ";
Ipp8u substring[] = " substring ";
Ipp8u dst_string [ sizeof (string) + sizeof (substring) - 1];
int dst_string_len ;

ippsInsert_8u( string, sizeof (string) - 1, substring, sizeof (substring) - 1, dst_string
, 16 );

dst_string [ sizeof ( dst_string ) - 1] = 0;

printf ( "ippsInsert_8u returned: %s.\n", (char*) dst_string );

dst_string_len = sizeof ( dst_string ) - 1;

ippsRemove_8u_I( dst_string , & dst_string_len , 16, sizeof (substring) - 1 );

dst_string [ dst_string_len ] = 0;

printf ( "ippsRemove_8u_I returned: %s.\n", (char*) dst_string );
```

Output:

`ippsInsert_8u` returned: 1st string part substring 2nd string part .

`ippsRemove_8u_I` returned: 1st string part 2nd string part .

## Remove

*Removes a specified number of elements from the string.*

---

### Syntax

```
IppStatus ippsRemove_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int
startIndex, int len);
```

```
IppStatus ippsRemove_16u(const Ipp16u* pSrc, int srcLen, Ipp16u* pDst, int
startIndex, int len);
```

```
IppStatus ippsRemove_8u_I(Ipp8u* pSrcDst, int* pSrcDstLen, int startIndex,
int len);
```

```
IppStatus ippsRemove_16u_I(Ipp16u* pSrcDst, int* pSrcDstLen, int startIndex,
int len);
```

### Parameters

<i>pSrc</i>	Pointer to the source string.
<i>srcLen</i>	Number of elements in the source string.
<i>pDst</i>	Pointer to the destination string.
<i>pSrcDst</i>	Pointer to the source and destination string for in-place operation.
<i>pSrcDstLen</i>	Pointer to the number of elements in the source and destination string for in-place operation.
<i>startIndex</i>	Index of the starting point.
<i>len</i>	Number of elements to be removed.

### Description

The function `ippsRemove` is declared in the `ippch.h` file. This function removes the *len* elements from a source string *pSrc* of length *srcLen*. Starting position is specified by *startIndex*. The result is stored in the *pDst*.

The in-place flavors of `ippsRemove` remove the *len* elements from the source string *pSrcDst* and store the result in the destination string *pSrcDst*.

Code [example 11-2](#) above shows how to use the function `ippsRemove_8u_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if one of the <i>srcLen</i> , <i>len</i> , <i>pSrcSdtLen</i> , <i>startIndex</i> is negative, or ( <i>startIndex</i> + <i>len</i> ) is greater than <i>srcLen</i> or <i>pSrcDstLen</i> .

## Compare

Compares two strings of the fixed length.

### Syntax

```
IppStatus ippCompare_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len,
int* pResult);

IppStatus ippCompare_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, int len,
int* pResult);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source string.
<i>pSrc2</i>	Pointer to the second source string.
<i>len</i>	Maximum number of elements to be compared.
<i>pResult</i>	Pointer to the result.

### Description

The function `ippCompare` is declared in the `ippch.h` file. This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. The value *pResult* = *pSrc1*[*i*] - *pSrc2*[*i*] is computed successively for each *i*-th element, *i* = 0, ..., *len*-1. When the first pair of non-matching elements occurs (that is, when *pResult* is not equal to zero), the function stops operation and returns the value *pResult*. The returned value is positive when *pSrc1*[*i*] > *pSrc2*[*i*] and negative when *pSrc1*[*i*] < *pSrc2*[*i*]. If the strings are equal, the function returns *pResult* = 0. The comparison is case-sensitive.

Example 11-3 below shows how to use the function `ippCompare_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>len</code> is negative.

## Example 11-3 Using the Functions `ippsCompare` and `ippsCompareIgnoreCaseLatin`

```

Ipp8u string0[] = "These functions compare two strings";
Ipp8u string1[] = "These FUNCTIONS compare two strings";

int result;

ippsCompare_8u( string0, string1, sizeof (string0) - 1, &result );
printf ( "ippsCompare_8u said: " );

printf ( "string0 is %s string1.\n", (result < 0) ? "less than" :
((result > 0) ? "greater than" : "equal to") );

ippsCompareIgnoreCaseLatin_8u( string0, string1, sizeof (string0) - 1,
&result );

printf ( "ippsCompareIgnoreCaseLatin_8u said: " );
printf ( "string0 is %s string1.\n", (result < 0) ? "less than" :
((result > 0) ? "greater than" : "equal to") );

```

Output:

```

ippsCompare_8u said: string0 is greater than string1.
ippsCompareIgnoreCaseLatin_8u said: string0 is equal to string1.

```

## CompareIgnoreCase, CompareIgnoreCaseLatin

*Compares two strings of the fixed length ignoring case.*

---

### Syntax

```

IppStatus ippsCompareIgnoreCase_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2,
int len, int* pResult);

```



```
IppStatus ippsCompareIgnoreCaseLatin_8u(const Ipp8u* pSrc1, const Ipp8u*
pSrc2, int len, int* pResult);

IppStatus ippsCompareIgnoreCaseLatin_16u(const Ipp16u* pSrc1, const Ipp16u*
pSrc2, int len, int* pResult);
```

## Parameters

<i>pSrc1</i>	Pointer to the first source string.
<i>pSrc2</i>	Pointer to the second source string.
<i>len</i>	Maximum number of elements to be compared.
<i>pResult</i>	Pointer to the result.

## Description

The functions `ippsCompareIgnoreCase` and `ippsCompareIgnoreCaseLatin` are declared in the `ippch.h` file. These functions compare first *len* elements of two strings *pSrc1* and *pSrc2*. If all pairs of elements in the strings are equal, the function returns *pResult* = 0. If the pair of non-matching elements occurs in the *i*-th position, the function stops operation and returns *pResult*. The returned value is positive when *pSrc1[i]* > *pSrc2[i]* and negative when *pSrc1[i]* < *pSrc2[i]*. The comparison is case-insensitive.

The function `ippsCompareIgnore` operates with Unicode characters. The function `ippsCompareIgnoreLatin` operates with ASCII characters.

Code [example 11-3](#) shows how to use the function `ippsCompareIgnoreCaseLatin_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

## Equal

*Compares two string of the fixed length for equality.*

---

### Syntax

```
IppStatus ippsEqual_8u(const Ipp8u* pSrc1, const Ipp8u* pSrc2, int len, int* pResult);

IppStatus ippsEqual_16u(const Ipp16u* pSrc1, const Ipp16u* pSrc2, int len, int* pResult);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source string.
<i>pSrc2</i>	Pointer to the second source string.
<i>len</i>	Maximum number of elements to be compared.
<i>pResult</i>	Pointer to the result.

### Description

The function `ippsEqual` is declared in the `ippch.h` file. This function compares first *len* elements of two strings *pSrc1* and *pSrc2*. Each element of the first string is compared with the corresponding element of the second string. When the first pair of non-matching elements is found, the function stops operation and stores 0 in *pResult*. If the strings are equal, the function stores 1 in *pResult*. The comparison is case-sensitive.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

## TrimC

*Deletes all occurrences of a specified symbol in the beginning and in the end of the string.*

---

### Syntax

```
IppStatus ippsTrimC_8u(const Ipp8u* pSrc, int srcLen, Ipp8u odd, Ipp8u* pDst,
int* pDstLen);
```

```
IppStatus ippsTrimC_16u(const Ipp16u* pSrc, int srcLen, Ipp16u odd, Ipp16u*
pDst, int* pDstLen);
```

```
IppStatus ippsTrimC_8u_I(Ipp8u* pSrcDst, int* pLen, Ipp8u odd);
```

```
IppStatus ippsTrimC_16u_I(Ipp16u* pSrcDst, int* pLen, Ipp16u odd);
```

### Parameters

<i>pSrc</i>	Pointer to the source string.
<i>srcLen</i>	Number of elements in the source string.
<i>pSrcDst</i>	Pointer to the source and destination string for the in-place operation.
<i>pDst</i>	Pointer to the destination string.
<i>pDstLen</i>	Pointer to the computed number of elements in the destination string.
<i>pLen</i>	Pointer to the number of elements in the source and destination string for the in-place operation.
<i>odd</i>	Symbol to be deleted.

### Description

The function `ippsTrimC` is declared in the `ippch.h` file. This function deletes all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrc* containing *srcLen* elements. The function stores the result string containing *pDstLen* elements in *pDst*.

The in-place flavors of `ippsTrimC` delete all occurrences of a specified symbol *odd* if it is present in the beginning and in the end of the source string *pSrcDst* containing *pLen* elements. These functions store the result string containing *pLen* elements in *pSrcDst*.

The operation is case-sensitive.

Code example 11-4 below shows how to use the function `ippsTrimC_8u_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>srcLen</i> or <i>pLen</i> is negative.

## Example 11-4 Using the functions `ippsTrimC` and `ippsTrimCAny`

```

Ipp8u string[] = " ### abracadabra $$$ ";
Ipp8u trim[] = " $*# ";
Ipp8u dst_string [ sizeof (string)];
int string_len , dst_string_len ;

ippsTrimCAny_8u( string, sizeof (string) - 1, trim, sizeof (string) - 1, dst_string,&
dst_string_len );

dst_string [ dst_string_len ] = 0;

printf ( "ippsTrimCAny_8u returned: %s.\n", (char*) dst_string );

string_len = sizeof (string) - 1;

ippsTrimC_8u_I( string, & string_len , ' # ' );

string[ string_len ] = 0;

printf ( "ippsTrimC_8u_I returned: %s.\n", (char*)string );

```

Output:

```

ippsTrimCAny_8u returned: abracadabra .
ippsTrimC_8u_I returned: abracadabra$$$ .

```

## TrimCAny, TrimStartCAny, TrimEndCAny

*Deletes all occurrences of any of the specified symbols in the beginning and in the end of the source string.*

---

### Syntax

```
IppStatus ippsTrimCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u* pTrim,
int trimLen, Ipp8u* pDst, int* pDstLen);
```

```
IppStatus ippsTrimCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);
```

```
IppStatus ippsTrimStartCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u*
pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
```

```
IppStatus ippsTrimStartCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);
```

```
IppStatus ippsTrimEndCAny_8u(const Ipp8u* pSrc, int srcLen, const Ipp8u*
pTrim, int trimLen, Ipp8u* pDst, int* pDstLen);
```

```
IppStatus ippsTrimEndCAny_16u(const Ipp16u* pSrc, int srcLen, const Ipp16u*
pTrim, int trimLen, Ipp16u* pDst, int* pDstLen);
```

### Parameters

<i>pSrc</i>	Pointer to the source string.
<i>srcLen</i>	Number of elements in the source string.
<i>pTrim</i>	Pointer to the array containing the specified elements.
<i>trimLen</i>	Number of elements in the array.
<i>pDst</i>	Pointer to the destination string.
<i>pDstLen</i>	Pointer to the computed number of elements in the destination string.

### Description

The functions `ippsTrimCAny`, `ippsTrimStartCAny`, and `ippsTrimEndCAny` are declared in the `ippch.h` file. The function `ippsTrimCAny` deletes all occurrences of any of the specified elements stored in the array `pTrim` if they are present either in the beginning or in the end of the source string `pSrc`, and stores the result string containing `pDstLen` elements in `pDst`. The

function stops operation when it finds the first non-matching element. The functions `ippsTrimStartCAny` and `ippsTrimEndCAny` perform this operation only in the beginning or in the end of the source string `pSrc`, respectively. The operation is case-sensitive.

Code [example 11-4](#) shows how to use the function `ippsTrimCAny_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>srcLen</code> or <code>trimLen</code> is negative.

## ReplaceC

*Replaces all occurrences of a specified element in the source string with another element.*

---

### Syntax

```
ippStatus ippsReplaceC_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, Ipp8u oldVal, Ipp8u newVal);
```

```
ippStatus ippsReplaceC_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len, Ipp16u oldVal, Ipp16u newVal);
```

### Parameters

<code>pSrc</code>	Pointer to the source string.
<code>len</code>	Number of elements in the source string.
<code>pDst</code>	Pointer to the destination string.
<code>oldVal</code>	Element to be replaced.
<code>newVal</code>	Element that replaces <code>oldVal</code> .

### Description

The function `ippsReplaceC` is declared in the `ippch.h` file. This function replaces all occurrences of a specified element `oldVal` in the source string `pSrc` with another specified element `newVal`, and stores the new string in the `pDst`. The operation is case-sensitive.

Example 11-5 below shows how to use the function `ippsReplaceC_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>len</code> is negative.

## Example 11-5 Using the function `ippsReplaceC`

```
Ipp8u string[] = "abracadabra";  
Ipp8u dst_string [ sizeof (string)];  
ippsReplaceC_8u( string, dst_string , sizeof (string), 'a', 'o' );  
printf ( "ippsReplaceC_8u returned: %s.\n", (char*) dst_string );  
  
Output:  
  
ippsReplaceC_8u returned: obrocodobro.
```

## Uppercase, UppercaseLatin

*Converts alphabetic characters of a string to all uppercase symbols.*

---

### Syntax

```
IppStatus ippsUppercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);  
IppStatus ippsUppercase_16u_I(Ipp16u* pSrcDst, int len);  
IppStatus ippsUppercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);  
IppStatus ippsUppercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);  
IppStatus ippsUppercaseLatin_8u_I(Ipp8u* pSrcDst, int len);  
IppStatus ippsUppercaseLatin_16u_I(Ipp16u* pSrcDst, int len);
```

### Parameters

<code>pSrc</code>	Pointer to the source string.
<code>pDst</code>	Pointer to the destination string.
<code>pSrcDst</code>	Pointer to the source and destination string for the in-place operation.

*len*                                      Number of elements in the string.

## Description

The functions `ippsUppercase` and `ippsUppercaseLatin` are declared in the `ippch.h` file. These functions convert each alphabetic character of the source string *pSrc* to upper case and stores the result in *pDst*.

The in-place flavors of these functions convert each alphabetic character of the source string *pSrcDst* to upper case and store the result in *pSrcDst*.

The function `ippsUppercase` operates with Unicode characters. The function `ippsUppercaseLatin` operates with ASCII characters.

Code [example 11-6](#) shows how to use the function `ippsUppercaseLatin_8u`.

## Return Values

<code>ippsStsNoErr</code>	Indicates no error.
<code>ippsStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippsStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

## Lowercase, LowercaseLatin

*Converts alphabetic characters of a string to all lowercase symbols.*

---

### Syntax

```

IppStatus ippsLowercase_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsLowercase_16u_I(Ipp16u* pSrcDst, int len);
IppStatus ippsLowercaseLatin_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len);
IppStatus ippsLowercaseLatin_16u(const Ipp16u* pSrc, Ipp16u* pDst, int len);
IppStatus ippsLowercaseLatin_8u_I(Ipp8u* pSrcDst, int len);
IppStatus ippsLowercaseLatin_16u_I(Ipp16u* pSrcDst, int len);

```

### Parameters

*pSrc*                                      Pointer to the source string.



<i>pDst</i>	Pointer to the destination string.
<i>pSrcDst</i>	Pointer to the source and destination string for the in-place operation.
<i>len</i>	Number of elements in the string.

## Description

The functions `ippsLowercase` and `ippsLowercaseLatin` are declared in the `ippch.h` file. These functions convert each alphabetic character of the source string *pSrc* to lower case and store the result in *pDst*.

The in-place flavors of these functions convert each alphabetic character of the source string *pSrcDst* to lower case and store the result in *pSrcDst*.

The function `ippsLowercase` operates with Unicode characters. The function `ippsLowercaseLatin` operates with ASCII characters.

Example 11-6 below shows how to use the function `ippsLowercaseLatin_8u_I`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

## Example 11-6 Using the functions `ippsUppercaseLatin` and `ippsLowercaseLatin`

```
Ipp8u string[] = "These Functions Vary the Case!";
ippsLowercaseLatin_8u_I( string, sizeof (string) - 1 );
printf ( "Lower: %s\n", (char*)string );
ippsUppercaseLatin_8u_I( string, sizeof (string) - 1 );
printf ( "Upper: %s\n", (char*)string );
```

Output:

```
Lower: these functions vary the case!
Upper: THESE FUNCTIONS VARY THE CASE!
```

## Hash

*Calculates the hash value for the string.*

---

### Syntax

```

IppStatus ippsHash_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHash_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHashSJ2_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHashSJ2_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHashMSCS_8u32u(const Ipp8u* pSrc, int len, Ipp32u* pHashVal);
IppStatus ippsHashMSCS_16u32u(const Ipp16u* pSrc, int len, Ipp32u* pHashVal);

```

### Parameters

<i>pSrc</i>	Pointer to the source string.
<i>len</i>	Number of elements in the string.
<i>pHashVal</i>	Pointer to the result value.

### Description

The function `ippsHash` is declared in the `ippch.h` file. This function produces the hash value *pHashVal* for the specified string *pSrc*. The hash value is fairly unique to the given string. If hash values of two strings are different, the strings are different as well. If the hash values are the same, the strings are probably identical. The hash value is calculated in the following manner: initial value is set to 0, then the hash value is calculated successively for each *i*-th string element as  $pHashVal[i] = 2 * pHashVal[i-1] \wedge pSrc[i]$ , where  $\wedge$  denotes a bitwise exclusive OR (XOR) operator. The hash value for the last element is the hash value for the whole string.

Code example 11-7 below shows how to use the functions `ippsHash_8u32u`, `ippsHashSJ2_8u32u`, and `ippsHashMSCS_8u32u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len</i> is negative.

### Example 11-7 Using the function `ippsHash`

```
Ipp8u string[] = "monkey";  
Ipp32u hash;  
hash = 0;  
ippsHash_8u32u( string, sizeof (string) - 1, &hash );  
printf ( "ippsHash_8u32u hash value: %u\n", hash );  
hash = 0;  
ippsHashSJ2_8u32u( string, sizeof (string) - 1, &hash );  
printf ( "ippsHashSJ2_8u32u hash value: %u\n", hash );  
hash = 0;  
ippsHashMSCS_8u32u( string, sizeof (string) - 1, &hash );  
printf ( "ippsHashMSCS_8u32u hash value: %u\n", hash );  
Output:  
ippsHash_8u32u hash value: 2367  
ippsHashSJ2_8u32u hash value: 3226471379  
ippsHashMSCS_8u32u hash value: 1466279646
```

## Concat

*Concatenates several strings together.*

---

### Syntax

```
IppStatus ippsConcat_8u_D2L(const Ipp8u* const pSrc[], const int* pSrcLen[],  
int numSrc, Ipp8u* pDst);  
IppStatus ippsConcat_16u_D2L(const Ipp16u* const pSrc[], const int* pSrcLen[],  
int numSrc, Ipp16u* pDst);  
IppStatus ippsConcat_8u(const Ipp8u* pSrc1, int len1, const Ipp8u* pSrc2,  
int len2, Ipp8u* pDst);  
IppStatus ippsConcat_16u(const Ipp16u* pSrc1, int len1, const Ipp16u* pSrc2,  
int len2, Ipp16u* pDst);
```

## Parameters

<i>pSrc1</i>	Pointer to the first source string.
<i>len1</i>	Number of elements in the first string.
<i>pSrc2</i>	Pointer to the second source string.
<i>len2</i>	Number of elements in the second string.
<i>pSrc</i>	Pointer to the array of source strings.
<i>pSrcLen</i>	Pointer to the array of lengths of the source strings.
<i>numSrc</i>	Number of source strings.
<i>pDst</i>	Pointer to the destination string.

## Description

The function `ippsConcat` is declared in the `ippch.h` file. This function concatenates several strings together. Functions with `D2L` suffix operate with multiple *numSrc* strings *pSrc* [ ], while functions without this suffix operate with two strings *pSrc1* and *pSrc2* only. Resulting string is stored in the *pDst*. Necessary memory blocks must be allocated for the destination string before the function is called. Summary length of the strings to be concatenated can not be greater than `IPP_MAX_32S`.

Example 11-8 below shows how to use the functions `ippsConcat_8u` and `ippsConcat_8u_D2L`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>len1</i> or <i>len2</i> is negative, or <i>srcLen</i> [i] is negative for <i>i</i> < <i>numSrc</i> .
<code>ippStsSizeErr</code>	Indicates an error condition if <i>numSrc</i> is equal to or less than 0.

**Example 11-8. Using the functions `ippsConcat` and `ippsConcatC`**

```

Ipp8u string0[] = "This is the initial string.";
Ipp8u string1[] = "Extra text added to the string...";
Ipp8u* string_ptr [2] = { string0, string1 };
int string_len_ptr [2] = { sizeof (string0) - 1, sizeof (string1)};
Ipp8u dst_string [ sizeof (string0) + sizeof (string1)];
ippsConcat_8u( string0, sizeof (string0) - 1, string1, sizeof (string1), dst_string );
printf ("ippsConcat_8u said: %s\n", (char*) dst_string );
ippsConcat_8u_D2L( string_ptr , string_len_ptr , 2, dst_string );
printf ("ippsConcat_8u_D2L said: %s\n", (char*) dst_string );
ippsConcatC_8u_D2L( string_ptr , string_len_ptr, 2, '#', dst_string );
printf ("ippsConcatC_8u_D2L said: %s\n", (char*) dst_string );

```

Output:

```

ippsConcat_8u said: This is the initial string. Extra text added to the string...
ippsConcat_8u_D2L said: This is the initial string. Extra text added to the string...
ippsConcatC_8u_D2L said: This is the initial string. # Extra text added to the string...

```

**ConcatC**

*Concatenates several strings together and inserts symbol delimiters between them.*

---

**Syntax**

```

IppStatus ippsConcatC_8u_D2L(const Ipp8u* const pSrc[], const int* pSrcLen[],
int numSrc, Ipp8u delim, Ipp8u* pDst);

IppStatus ippsConcatC_16u_D2L(const Ipp16u* const pSrc[], const int*
pSrcLen[], int numSrc, Ipp16u delim, Ipp16u* pDst);

```

**Parameters**

<i>pSrc</i>	Pointer to the array of source strings.
<i>pSrcLen</i>	Pointer to the array of lengths of the source strings.
<i>numSrc</i>	Number of source strings.

<i>delim</i>	Symbol delimiter.
<i>pDst</i>	Pointer to the destination string.

### Description

The function `ippsConcatC` is declared in the `ippch.h` file. This function concatenates *numSrc* strings *pSrc* together and inserts a specified delimiter symbol *delim* between them in the resulting string *pDst*.

Code [example 11-8](#) above shows how to use the function `ippsConcatC_8u_D2L`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if at least one of the specified pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>srcLen[i]</code> is negative for <code>i &lt; numSrc</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if <i>numSrc</i> is equal to or less than 0.

## SplitC

*Splits source string into separate parts.*

---

### Syntax

```
IppStatus ippsSplitC_8u_D2L(const Ipp8u* pSrc, int srcLen, Ipp8u delim,
Ipp8u* pDst[], int* pDstLen[], int* pNumDst);

IppStatus ippsSplitC_16u_D2L(const Ipp16u* pSrc, int srcLen, Ipp16u delim,
Ipp16u* pDst[], int* pDstLen[], int* pNumDst);
```

### Parameters

<i>pSrc</i>	Pointer to the source strings.
<i>srcLen</i>	Number of elements in the source string.
<i>delim</i>	Symbol delimiter.
<i>pDst</i>	Pointer to the array of the destination strings.
<i>pDstLen</i>	Pointer to array of the destination string lengths.
<i>pNumDst</i>	Number of destination strings.

## Description

The function `ippsSplitC` is declared in the `ippch.h` file. This function breaks source string `pSrc` into `pNumDst` separate strings `pDst` using a specified symbol `delim` as a delimiter. If `n` specified delimiters occur in the beginning or in the end of the source string, then `n` empty strings are appended to the array of destination strings. If `n` specified delimiters occur in a certain position within the source string, then  $(n-1)$  empty strings are inserted into the array of destination strings, where `n` is the number of delimiter occurrences in this position.

Code example 11-9 below shows how to use the function `ippsSplitC_8u_2DL`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.
<code>ippStsLengthErr</code>	Indicates an error condition if <code>srcLen</code> is negative, or <code>pDstLen</code> is negative for $i < pNumDst$ .
<code>ippStsSizeErr</code>	Indicates an error condition if <code>pNumDst</code> is equal to or less than 0.
<code>ippStsOvermatchStrings</code>	Indicates a warning if number of output strings exceeds the initially specified number <code>pNumDst</code> ; in this case odd strings are discarded.
<code>ippStsOverlongString</code>	Indicates a warning if in some output strings the number of elements exceeds the initially specified value <code>pDstLen</code> ; in this case corresponding strings are truncated to initial lengths.

## Example 11-9 Using the function `ippSplitC`

```

Ipp8u string[] = "1st string # 2nd string";
Ipp8u dst_string0[ sizeof (string)];
Ipp8u dst_string1[ sizeof (string)];
Ipp8u* dst_string_ptr [] = { dst_string0, dst_string1 };
int dst_string_len_ptr [] = { sizeof (dst_string0), sizeof (dst_string1) };
int dst_string_num = 2;
int i ;

ippSplitC_8u_D2L( string, sizeof (string) - 1, '#', dst_string_ptr, dst_string_len_ptr,
& dst_string_num );

printf ( "Destination strings number: %d\n", dst_string_num );

for( i = 0; i < dst_string_num ; i ++ ) {
    dst_string_ptr [ i ][ dst_string_len_ptr [ i ]] = 0;
    printf ( "%d: %s.\n", i, (char*) dst_string_ptr [ i ] );
}

```

Output:

Destination strings number: 2

0: 1st string.

1: 2nd string.

## Regular Expressions

This section describes the Intel IPP functions that perform matching operations with the Perl-compatible regular expression patterns. See <http://search.cpan.org/dist/perl/pod/perlre.pod> for more details about Perl-compatible regular expressions.

The current version of the Intel IPP functions for regular expressions have some limitations, specifically they do not support literal (metacharacters `\l`, `\L`, `\u`, `\U`, `\N{name}`), embedded Perl code (`{code}`), extended regular expression (`??{code}`).



## RegexInitAlloc

*Allocates memory and initializes the structure for processing matching operation with regular expressions.*

---

### Syntax

```
IppStatus ippsRegexInitAlloc(const char* pPattern, const char* pOptions,  
IppRegexState** ppRegexState, int* pErrOffset);
```

### Parameters

<i>pPattern</i>	Pointer to the pattern of regular expression.
<i>pOptions</i>	Pointer to options for compiling and executing regular expressions (possible values <i>i</i> , <i>s</i> , <i>m</i> , <i>x</i> , <i>g</i> ). It must be <i>NULL</i> if no options are required.
<i>ppRegexState</i>	Double pointer to the structure containing internal form of a regular expression.
<i>pErrOffset</i>	Pointer to the offset in the pattern if compiling is break.

### Description

The function `ippsRegexInitAlloc` is declared in the `ippch.h` file. This function allocates memory and initializes a regular expression state structure *ppRegexState*. The function compiles the initial pattern of regular expressions *pPattern* in accordance with the compiling options specified by *pOptions*, converts it to the specific internal form, and stores it in the initialized structure. This structure is used by the function `ippsRegexFind` to perform matching operation.

If the compiling is not completed, the function returns the pointer *pErrOffset* pointed to the position in the pattern where the compiling is interrupted.

Code [example 11-10](#) shows how to use the function `ippsRegexInitAlloc`.

### Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error condition if one of the specified pointers is <i>NULL</i> .

`ippStsRegExpOptionsErr` Indicates an error if specified options are incorrect  
`ippStsRegExpQuantifierErr` Indicates an error if the quantifier is incorrect  
`ippStsRegExpGroupingErr` Indicates an error if the grouping is incorrect  
`ippStsRegExpBackRefErr` Indicates an error if the back reference is incorrect  
`ippStsRegExpChClassErr` Indicates an error if the character class is incorrect  
`ippStsRegExpMetaChErr` Indicates an error if the metacharacter is incorrect

## RegExpFree

*Frees the memory allocated for a regular expression state structure.*

---

### Syntax

```
IppStatus ippRegExpFree(IppsRegExpState* pRegExpState);
```

### Parameters

`pRegExpState` Pointer to the pattern of regular expression.

### Description

The function `ippRegExpFree` is declared in the `ippch.h` file. This function frees memory allocated by the function `ippRegExpInitAlloc` for the regular expressions state structure `pRegExpState`.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error condition if `pRegExpState` pointer is `NULL`.

## RegExpInit

*Initializes the structure for processing matching operation with regular expressions.*

---

### Syntax

```
IppStatus ippRegExpInit(const char* pPattern, const char* pOptions,  
IppsRegExpState* pRegExpState, int* pErrOffset);
```

## Parameters

<i>pPattern</i>	Pointer to the pattern of regular expression.
<i>pOptions</i>	Pointer to options for compiling and executing regular expressions (possible values <i>i</i> , <i>s</i> , <i>m</i> , <i>x</i> , <i>g</i> ). It must be NULL if no options are required.
<i>pRegExpState</i>	Pointer to the structure containing internal form of a regular expression.
<i>pErrOffset</i>	Pointer to the offset in the pattern if compiling is break.

## Description

The function `ippsRegExpInit` is declared in the `ippch.h` file. This function initializes a regular expression state structure *pRegExpState* in the external buffer. The size of this buffer must be computed previously by calling the function `ippsRegExpGetSize`. The function compiles the initial pattern of regular expressions *pPattern* in accordance with the compiling options specified by *pOptions*, converts it to the specific internal form, and stores it in the initialized structure. This structure is used by the function `ippsRegExpFind` to perform matching operation.

If the compiling is not completed, the function returns the pointer *pErrOffset* pointed to the position in the pattern where the compiling is interrupted.



**CAUTION.** The parameter *pPattern* must be the same for both functions `ippsRegExpInit` and `ippsRegExpGetSize`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.
<code>ippStsRegExOptionsErr</code>	Indicates an error if specified options are incorrect
<code>ippStsRegExQuantifierErr</code>	Indicates an error if the quantifier is incorrect
<code>ippStsRegExGroupingErr</code>	Indicates an error if the grouping is incorrect
<code>ippStsRegExBackRefErr</code>	Indicates an error if the back reference is incorrect
<code>ippStsRegExChClassErr</code>	Indicates an error if the character class is incorrect
<code>ippStsRegExMetaChErr</code>	Indicates an error if the metacharacter is incorrect

## RegExpGetSize

*Computes the size of the regular expression state structure.*

---

### Syntax

```
IppStatus ippsRegExpGetSize(const char* pPattern, int* pRegExpStateSize);
```

### Parameters

<i>pPattern</i>	Pointer to the pattern of regular expression.
<i>pRegExpStateSize</i>	Pointer to the computed size of the regular expression structure.

### Description

The function `ippsRegExpGetSize` is declared in the `ippch.h` file. This function computes the size of the memory that is necessary to initialize by the function `ippsRegExpInit` the regular expression state structure containing the pattern *pPattern* in the internal form. The value of the computed size is stored in the *pRegExpStateSize*.




---

**CAUTION.** The parameter *pPattern* must be the same for both functions `ippsRegExpInit` and `ippsRegExpGetSize`.

---

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.

## RegExpSetMatchLimit

*Sets the value of the `matchLimit` parameter.*

---

### Syntax

```
IppStatus ippsRegExpSetMatchLimit(int matchLimit, IppRegExpState* pRegExpState);
```

## Parameters

<i>matchLimit</i>	Value of the of the matches kept in stack.
<i>pRegExpState</i>	Pointer to the structure containing internal form of a regular expression.

## Description

The function `ippsRegExpSetMatchLimit` is declared in the `ippch.h` file. This function sets the value of the parameter *matchLimit* that specifies how many times the function `ippsRegExpFind` can be called through the single execution avoiding the possible stack overflow. The default value is set very large, so you should set this parameter to the reasonable value in accordance with your needs.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if the <i>pRegExpState</i> pointer is <code>NULL</code> .

# RegExpFind

*Looks for the occurrences of the substrings matching the specified regular expression.*

---

## Syntax

```
IppStatus ippsRegExpFind_8u(const Ipp8u* pSrc, int srcLen, IppRegExpState* pRegExpState, IppRegExpFind* pFind, int* pNumFind);
```

## Parameters

<i>pSrc</i>	Pointer to the source strings.
<i>srcLen</i>	Number of elements in the source string.
<i>pRegExpState</i>	Pointer to the structure containing internal form of regular expression.
<i>pFind</i>	Array of pointers to the matching substrings.
<i>pNumFind</i>	Size of the array <i>pFind</i> on input, number of matching substrings on output.

## Description

The function `ippsRegExpFind` is declared in the `ippch.h` file. This function search through the `srcLen` elements of the source string `pSrc` for substrings that match the specified regular expression in accordance with the regular expression pattern that is stored in the structure `pRegExpState`. This structure must be initialized by the functions `ippsRegExpInit` or `ippsRegExpInitAlloc` beforehand. Initially the parameter `pNumFind` specifies the size of array `pFind`, the output parameter `pNumFind` returns the number of the matching substrings.

`pFind->pFind` specifies the offset of the pointer to the matching substring, and `pFind->lenFind` - number of elements in the matching substring. `pFind [0]` points to the substring that matches the whole regular expression, `pFind [1]` points to the substring that matches the first grouping, `pFind[2]` points to the substring that matches the second grouping, and so on.



**CAUTION.** It is recommended to set the default value of the parameter `matchLimit` in accordance with real necessity by calling the function `ippsRegExpSetMatchLimit` beforehand.

Code example 11-10 below shows how to use the function `ippsRegExpFind_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error condition if <code>srcLen</code> is negative, or <code>pNumFind</code> is less than or equal to 0.
<code>ippStsRegExpErr</code>	The state structure <code>pRegExpState</code> contains wrong data.
<code>ippStsRegExpMatchLimitErr</code>	The match limit has been exhausted.

### Example 11-10. Using the Function `ippsRegExpFind`

```

Ipp8u string[] = "Hello World!";

IppRegExpFind find[2];

int find_num = sizeof (find)/ sizeof (find[0]);

IppRegExpState *state;

IppStatus status;

int i , err;

ippsRegExpInitAlloc ("[ hH ] ello (\\w+)", NULL, &state, &err );

ippsRegExpFind_8u( string, sizeof (string) - 1, state, find, & find_num );

printf ( "Found strings number = %d\\n", find_num );

for( i = 0; i < find_num ; i ++ ) {

    Ipp8u tmp = ((Ipp8u*)find[ i ]. pFind )[find[ i ]. lenFind ];

    ((Ipp8u*)find[ i ]. pFind )[find[ i ]. lenFind ] = 0;

    printf ( "%d: %s\\n", i, (char*)find[ i ]. pFind );

    ((Ipp8u*)find[ i ]. pFind )[find[ i ]. lenFind ] = tmp ;

}

```

Output:

Found strings number = 2

0: Hello World

1: World

## RegExpSetFormat

*Sets source encoding format for given compiled pattern.*

---

### Syntax

```

IppStatus ippsRegExpSetFormat(IppRegExpFormat fmt, IppRegExpState*
pRegExpState);

```

## Parameters

<i>fmt</i>	New source encoding mode.
<i>pRegExpState</i>	Pointer to the structure containing internal form of a regular expression.

## Description

The function `ippsRegExpSetFormat` is declared in the `ippch.h` file. The function sets the new source encoding format for given compiled pattern. Default source encoding format after `ippsRegExpInit` is UTF-8 with ASCII auto detection.

The enumeration `IppRegExpFormat` for representing a source encoding mode is defined as

```
typedef enum {
    ippFmtASCII=0,
    ippFmtUTF8,
} IppRegExpFormat;
```



**CAUTION.** The function `ippsRegExpFind` returns `ippStsRegExprErr` when pattern and source string are coded with different encoding, or pattern contains unsupported features by chosen encoding format.

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pRegExpState</i> pointer is <code>NULL</code> .
<code>ippStsRangeErr</code>	Indicates an error when mode is not a valid element of the enumerated type <code>IppRegExpFormat</code> .



## ConvertUTF

*Converts the UTF16BE or UTF16LE format to UTF8 and vice versa.*

---

### Syntax

```
IppStatus ippsConvertUTF_8u16u(const Ipp8u* pSrc, Ipp32u* pSrcLen, Ipp16u* pDst, Ipp32u* pDstLen, int BEFlag);
```

```
IppStatus ippsConvertUTF_16u8u(const Ipp16u* pSrc, Ipp32u* pSrcLen, Ipp8u* pDst, Ipp32u* pDstLen, int BEFlag);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pSrcLen</i>	Length of the <i>pSrc</i> vector on input; its used length on output.
<i>pDst</i>	Pointer to the destination vector.
<i>pDstLen</i>	Length of the <i>pDst</i> vector on input; its used length on output.
<i>BEFlag</i>	Flag to indicate the UTF16BE format. 0 means the UTF16LE format.

### Description

The function `ippsConvertUTF` is declared in the `ippch.h` file. The function flavor `ippsConvertUTF_8u16u` converts the UTF8 format to the UTF16LE or UTF16BE format. The function flavor `ippsConvertUTF_16u8u` converts UTF16LE or UTF16BE format to the UTF8 format.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error condition if one of the specified pointers is NULL.

## RegExpMultiGetSize

*Computes the size of the memory for the multi-pattern search engine.*

---

### Syntax

```
IppStatus ippsRegExpMultiGetSize(Ipp32u numPatterns, int* pSize);
```

### Parameters

<i>numPatterns</i>	Maximum number of patterns.
<i>pSize</i>	Pointer to the computed size of the memory for the multi-pattern search engine.

### Description

The function `ippsRegExpMultiGetSize` is declared in the `ippch.h` file. This function computes the size of the memory for state structure containing the multi-pattern search engine. The value of the computed size is stored in the *pSize*.




---

**CAUTION.** The value of the parameter *numPatterns* must be the same as used for the `ippsRegExpMultiInit` function call.

---

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>numPatterns</i> is equal to zero.

## RegExpMultiInit

*Initializes the state structure for multi-pattern search engine.*

---

### Syntax

```
IppStatus ippsRegExpMultiInit(IppRegExpMultiState* pState, Ipp32u numPatterns);
```

## Parameters

<i>pState</i>	Pointer to the state structure for the multi-pattern search.
<i>numPatterns</i>	Maximum number of patterns.

## Description

The function `ippsRegExpMultiInit` is declared in the `ippch.h` file. This function initializes the state structure for multi-pattern search engine *pState* in the external buffer. The size of this buffer must be computed previously by calling the function `ippsRegExpMultiGetSize`.



### CAUTION.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pState</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>numPatterns</i> is equal to zero.

## RegExpMultiInitAlloc

*Allocates and initializes the state structure for multi-pattern search engine.*

## Syntax

```
IppStatus ippsRegExpMultiInitAlloc(IppRegExpMultiState** ppState, Ipp32u numPatterns);
```

## Parameters

<i>ppState</i>	Double pointer to the state structure for the multip-pattern search engine.
<i>numPatterns</i>	Maximum number of patterns.

## Description

The function `ippsRegExpMultiInitAlloc` is declared in the `ippch.h` file. This function allocates memory and initializes the state structure for multi-pattern search engine `pState`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>ppState</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>numPatterns</code> is equal to zero.

## RegExpMultiFree

*Frees memory allocated for the state structure for the multi-pattern search engine.*

---

### Syntax

```
void ippsRegExpMultiFree(IppRegExpMultiState* pState);
```

### Parameters

<code>pState</code>	Pointer to the state structure for the multi-pattern search engine.
---------------------	---

### Description

The function `ippsRegExpMultiFree` is declared in the `ippch.h` file. This function frees memory allocated for the multiple patterns search engine state structure `pState`.

## RegExpMultiAdd

*Adds the specified pattern to the multiple patterns database.*

---

### Syntax

```
IppStatus ippsRegExpMultiAdd(const IppRegExpState* pRegExpState, Ipp32u regExpID, IppRegExpMultiState* pState);
```

## Parameters

<i>pRegExpState</i>	Pointer to the compiled pattern state structure.
<i>regexID</i>	Pattern ID, must be not equal to zero.
<i>pState</i>	Pointer to the state structure for the multi-pattern search engine.

## Description

The function `ippsRegExpMultiAdd` is declared in the `ippch.h` file. This function adds a new pattern *regexID* to the pattern database in the state structure *pState* for multiple patterns search engine. The pattern state structure *pRegExpState* must be compiled by the functions [ippsRegExpInit](#) or [ippsRegExpInitAlloc](#) beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pRegExpState</i> or <i>pState</i> pointer is NULL.
<code>ippStsBadArgErr</code>	Indicates an error when <i>regexID</i> is equal to zero; or pattern with such ID exists in the database.
<code>ippStsMemAllocErr</code>	Indicates an error when number of patterns is greater than maximum pattern number the database is initialized for.

## RegExpMultiDelete

*Deletes specified patterns from the multiple pattern database.*

---

### Syntax

```
IppStatus ippsRegExpMultiDelete(Ipp32u regexID, IppRegExpMultiState* pState);
```

## Parameters

<i>regexID</i>	Pattern ID, must not be equal to zero.
<i>pState</i>	Pointer to the state structure for the multi-pattern search engine.

## Description

The function `ippsRegExpMultiDelete` is declared in the `ippch.h` file. This function deletes a specified pattern `regexID` from the pattern database in the state structure `pState` for multi-pattern search engine.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pState</code> pointer is <code>NULL</code> .
<code>ippStsBadArgeErr</code>	Indicates an error when <code>regexID</code> is equal to zero; or pattern with such ID does not exist in the database.

## RegExpMultiModify

*Modifies specified patterns in the multiple pattern database.*

---

### Syntax

```
IppStatus ippsRegExpMultiModify(const IppRegExpState* pRegExpState, Ipp32u regexID, IppRegExpMultiState* pState);
```

### Parameters

<code>pRegExpState</code>	Pointer to the compiled pattern state structure.
<code>regexID</code>	Pattern ID, must be not equal to zero.
<code>pState</code>	Pointer to the state structure for the multi-pattern search engine.

### Description

The function `ippsRegExpMultiModify` is declared in the `ippch.h` file. This function modifies a specified pattern `regexID` in the pattern database in the state structure `pState` for multi-pattern search engine. The pattern state structure `pRegExpState` must be compiled by the functions `ippsRegExpInit` or `ippsRegExpInitAlloc` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

---

<code>ippStsNullPtrErr</code>	Indicates an error when the <code>pRegExpState</code> or <code>pState</code> pointer is NULL.
<code>ippStsBadArgeErr</code>	Indicates an error when <code>regexpID</code> is equal to zero.

## RegExpMultiFind

*Looks for the occurrences of the substrings matching the multiple patterns.*

---

### Syntax

```
IppStatus ippRegExpMultiFind_8u(const Ipp8u* pSrc, int srcLen,
IppRegExpMultiFind* pDstMultiFind, const IppRegExpMultiState* pState);
```

### Parameters

<code>pSrc</code>	Pointer to the source string.
<code>srcLen</code>	Number of elements in the source string.
<code>pDstMultiFind</code>	Array of pointers to the matching patterns.
<code>pState</code>	Pointer to the state structure for the multi-pattern search engine.

### Description

The function `ippRegExpMultiFind` is declared in the `ippch.h` file. This function searches through the `srcLen` elements of the source string `pSrc` for substrings that match each pattern from the database in the state structure `pState` for multiple patterns search engine. This state structure must be initialized by the functions `ippRegExpMultiInit` or `ippRegExpMultiInitAlloc` beforehand.

Results of the search are stored in the array of structures `pDstMultiFind` of type `IppRegExpMultiFind`. The structure has the following fields:

```
typedef structure {
Ipp32u regexpDoneFlag;
Ipp32u regexpID;
Ipp32s numMultiFind ;
IppStatus status;
IppRegExpFind* pFind;
```

```
Ipp32s numMultiFind
}IppRegExpMultiFind;
```

For each element of the array *pDstMultiFind*:

*numMultiFind* - input value specifies the size of the array *pFind*, output value indicates the number of substrings matching for the pattern *regexpID*.

*pFind* - specifies the offset of the pointer to the matching substring, and the *lenFind* entry of *pFind* specifies number of elements in the matching substring. If the substring matches completely, *pFind* [0] points to the whole regular expression, *pFind*[1] points to the substring that matches the first grouping, *pFind*[2] points to the substring that matches the second grouping, and so on.

The *regexpDoneFlag* is initially set to 0. When the search for the corresponding pattern *regexpID* is finished - it is set to 1.

## Return Values

<i>ippStsNoErr</i>	Indicates no error.
<i>ippStsNullPtrErr</i>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<i>ippStsSizeErr</i>	Indicates an error when <i>srcLen</i> is less than or equal to zero.
<i>ippStsNoOperation</i>	Indicates a warning when <i>pState</i> database does not contain any patterns.



## Example of Using Multipattern Functions

### Example 11-8 Using of Multipattern Regular Expression Functions

```
/*String*/
Ipp8u *srcString = (Ipp8u *)"Hello World!";
/*1st pattern*/
char *pattern1 = "Hello";
/*2nd pattern*/
char *pattern2 = "World";
/*3rd pattern*/
char *pattern3 = ".$";
/*4th pattern*/
char *pattern4 = "0 W";
IppRegExpState* pRegExpState1;
IppRegExpState* pRegExpState2;
IppRegExpState* pRegExpState3;
IppRegExpState* pRegExpState4;
```

```

/*3 patterns will be used*/
int nPatterns = 3;
int numMatch = 30;
int pErrOffset, i, k;
IppRegExpMultiFind pMultiFind[3];
IppRegExpMultiState* pState=NULL;
/*Init multi pattern search engine*/
ippsRegExpMultiInitAlloc(&pState,nPatterns);
for(i=0;i<nPatterns;i++) {
    pMultiFind[i].pFind = (IppRegExpFind*)ippMalloc(numMatch*sizeof(IppRegExpFind));
    pMultiFind[i].numMultiFind = numMatch;
}
/*Compile 1st pattern*/
ippsRegExpInitAlloc(pattern1, NULL,&pRegExpState1, &pErrOffset);
ippsRegExpSetMatchLimit(numMatch,pRegExpState1);
/*Compile 2nd pattern*/
ippsRegExpInitAlloc(pattern2, NULL,&pRegExpState2, &pErrOffset);
ippsRegExpSetMatchLimit(numMatch,pRegExpState2);
/*Compile 3rd pattern*/
ippsRegExpInitAlloc(pattern3, NULL,&pRegExpState3, &pErrOffset);
ippsRegExpSetMatchLimit(numMatch,pRegExpState3);
/*Compile 4th pattern*/
ippsRegExpInitAlloc(pattern4, NULL,&pRegExpState4, &pErrOffset);
ippsRegExpSetMatchLimit(numMatch,pRegExpState4);
/*Add 1st, 2nd, and 3rd patterns*/
ippsRegExpMultiAdd(pRegExpState1, 1, pState);
ippsRegExpMultiAdd(pRegExpState2, 2, pState);
ippsRegExpMultiAdd(pRegExpState3, 3, pState);

```

```
/*Match*/
ippsRegExpMultiFind_8u( (const Ipp8u*)srcString, 12, pState, pMultiFind);
/*Output:
    RegExpID = 1
    Matches  = 1
    Hello
    RegExpID = 2
    Matches  = 1
    World
    RegExpID = 3
    Matches  = 1
!*/
/*Replace 2nd pattern by 4th pattern*/
ippsRegExpMultiModify(pRegExpState4, 2, pState);
/*Delete 3rd pattern*/
ippsRegExpMultiDelete(3, pState);
/*Reset destination structure*/
for(i=0;i<nPatterns;i++) {
    pMultiFind[i].numMultiFind = numMatch;
}
/*Match*/
ippsRegExpMultiFind_8u( (const Ipp8u*)srcString, 13, pState, pMultiFind);
/*Output:
    RegExpID = 1
    Matches  = 1
    Hello
    RegExpID = 2
    Matches  = 1
```

```

        o W*/

/*Free memory*/
ippsRegExpMultiFree(pState);
ippsRegExpFree(pRegExpState1);
ippsRegExpFree(pRegExpState2);
ippsRegExpFree(pRegExpState3);
ippsRegExpFree(pRegExpState4);
for(i=0;i<nPatterns;i++) {
    ippFree(pMultiFind[i].pFind);
}

```

## RegExpReplaceGetSize

*Calculates the size of the state structure for the find-replace operation.*

---

### Syntax

```

IppStatus ippsRegExpReplaceGetSize(const Ipp8u* pSrcReplacement, Ipp32u*
pSize);

```

### Parameters

<i>pSrcReplacement</i>	Pointer to the source null-terminated replace pattern.
<i>pSize</i>	Pointer to the size of the state structure for the find and replace operation.

### Description

The function `ippsRegExpReplaceGetSize` is declared in the `ippch.h` file. This function calculates the size of the memory that is necessary to initialize the state structure required by the function `ippsRegExpReplaceInit`. The value of the calculated size is stored in the *pSize*.




---

**CAUTION.** Value of the parameter *pSrcReplacement* must be the same as used for the `ippsRegExpReplaceInit` function call.

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pSize</i> pointer is <code>NULL</code> .

## RegExpReplaceInit

*Initialize the state structure for the find-replace operation.*

---

### Syntax

```
IppStatus ippRegExpReplaceInit(const Ipp8u* pSrcReplacement,  
IppRegExpReplaceState* pReplaceState);
```

### Parameters

<i>pSrcReplacement</i>	Pointer to the source null-terminated replace pattern.
<i>pReplaceState</i>	Pointer to the state structure for the find and replace operation.

### Description

The function `ippRegExpReplaceInit` is declared in the `ippch.h` file. This function initializes a state structure *pState* for the find and replace operation in the external buffer. The size of this buffer must be computed previously by calling the function `ippRegExpReplaceGetSize`.



---

**CAUTION.** Value of the parameter *pSrcReplacement* must be the same as used for the `ippRegExpReplaceGetSize` function call.

---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when the <i>pReplaceState</i> or <i>pSrcReplacement</i> pointer is <code>NULL</code> .

## RegExpReplace

*Performs find and replace operation.*

---

### Syntax

```
ippStatus ippsRegExpReplace_8u(const Ipp8u* pSrc, int* pSrcLenOffset, Ipp8u*
pDst, int* pDstLen, IppRegExpFind* pFind, int* pNumFind, IppRegExpState*
pRegExpState, IppRegExpReplaceState* pReplaceState);
```

### Parameters

<i>pSrc</i>	Pointer to the source string.
<i>pSrcLenOffset</i>	Pointer to length of the <i>pSrc</i> vector on input; its used length on output.
<i>pDst</i>	Pointer to the destination string.
<i>pDstLen</i>	Pointer to length of the <i>pDst</i> vector on input; its used length on output.
<i>pFind</i>	Array of pointers to the matching substrings.
<i>pNumFind</i>	Pointer to size of the array <i>pFind</i> on input, to number of matching substrings on output.
<i>pRegExpState</i>	Pointer to the compiled pattern structure.
<i>pReplaceState</i>	Pointer to the state structure for the find and replace operation.

### Description

The function `ippsRegExpReplace` is declared in the `ippch.h` file. This function search through the *pSrcLen* elements of the source string *pSrc* for substrings that match the specified regular expression in accordance with the regular expression pattern that is stored in the structure *pRegExpState*. This state structure must be initialized by the functions [ippsRegExpInit](#) or [ippsRegExpInitAlloc](#) beforehand. All found matches are replaced according to the replacement pattern that is stored in the structure *pReplaceState*. This structure must be initialized beforehand by the function [ippsRegExpReplaceInit](#).

Initially the parameter *pNumFind* specifies the size of array *pFind*, the output parameter *pNumFind* returns the number of the matching substrings. *pFind->pFind* specifies the offset of the pointer to the matching substring, and *pFind->lenFind* - number of elements in the

matching substring. *pFind* [0] points to the substring that matches the whole regular expression, *pFind*[1] points to the substring that matches the first grouping, *pFind* [2] points to the substring that matches the second grouping, and so on.



---

**CAUTION.** The compiled regular expression pattern and/or replacement pattern can be used for different input strings in different combinations.

---

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>pSrcLen</i> or <i>pDstLen</i> is less than or equal to zero.

---

---



# Fixed-Accuracy Arithmetic Functions

12

This chapter describes Intel® IPP fixed-accuracy transcendental mathematical real and complex functions of vector arguments. These functions take an input vector as argument, compute values of the respective elementary function element-wise, and return the results in an output vector.

Function specifications comply with the common API agreement of Intel IPP, but include some new features essential to scientific arithmetic functions. The main feature is a more elaborate specification of accuracy that differs from the common definition in adding several new levels of accuracy, besides original levels introduced by single precision and double precision data formats.

Fixed-accuracy vector functions implementation supports the IEEE-754 standard in all flavors, which means that:

- All functions have a precisely determined and guaranteed level of accuracy for all argument values.
- All special value processing and exceptions handling requirements are met, which implies that when accuracy is below the standard level, the function meets the IEEE-754 requirements in all other respects.

The choice of accuracy levels should be based on practical experience and identified application demands. Available options are specified in the function name suffix and include A11, A21, or A24 for the single precision, and A21, A50, or A53 for the double precision data format. Flavors A11, A21, A26, and A50 provide approximately 3, 6, 8, and 15 exact decimal digits, respectively. For flavors A24 and A53, the maximum guaranteed error is within 1 ulp and in most cases does not exceed 0.55 ulp.

Fixed-accuracy arithmetic functions subset of Intel IPP has the similar functionality as the respective part of the Intel® Math Kernel Library (Intel® MKL).

However, Intel IPP provides lower-level transcendental functions that have separate flavors for each mode of operations and data type and are better suitable for multimedia and signal processing in real time applications. The full list of these functions is given in Table 12-1 below.

**Table 12-1. Intel IPP Fixed-Accuracy Arithmetic Functions**

Function Short Name	Description
Arithmetic Functions	
Add	Adds elements of a vector to corresponding elements of another vector.
Sub	Subtracts elements of a vector from corresponding elements of another vector.
Sqr	Computes square of each vector element.
Mul	Multiplies elements of a vector by corresponding elements of another vector.
MulByConj	Multiplies elements of a vector by conjugated corresponding elements of another vector.
Conj	Conjugates the value of each vector element.

Function Short Name	Description
<a href="#">Abs</a>	Computes the absolute value of each vector element.
<a href="#">Arg</a>	Computes an argument of vector elements.
Power and Root Functions	
<a href="#">Inv</a>	Computes inverse value of each vector element.
<a href="#">Div</a>	Divides elements of one vector by corresponding elements of another vector.
<a href="#">Sqrt</a>	Computes square root of each vector element.
<a href="#">InvSqrt</a>	Computes inverse square root of each vector element.
<a href="#">Cbrt</a>	Computes cube root of each vector element.
<a href="#">InvCbrt</a>	Computes inverse cube root of each vector element.
<a href="#">Pow2o3</a>	Computes the value of each vector element raised to the power of 2/3.
<a href="#">Pow3o2</a>	Computes the value of each vector element raised to the power of 3/2.
<a href="#">Pow</a>	Raises each element of one vector to the power of corresponding element of another vector.
<a href="#">Powx</a>	Raises each element of a vector to a constant power.
<a href="#">Hypot</a>	Computes a square root of sum of two squared elements.
Exponential and Logarithmic Functions	
<a href="#">Exp</a>	Raises $e$ to the power of each vector element.
<a href="#">Expml</a>	Computes $e$ raised to the power of each vector element and decreased by 1.
<a href="#">Ln</a>	Computes natural logarithm of each vector element.
<a href="#">Log10</a>	Computes common logarithm of each vector element.
<a href="#">Log1p</a>	Computes natural logarithm of each vector element decreased by 1.
Trigonometric Functions	
<a href="#">Cos</a>	Computes cosine of each vector element.
<a href="#">Sin</a>	Computes sine of each vector element.
<a href="#">SinCos</a>	Computes sine and cosine of each vector element.
<a href="#">CIS</a>	Computes complex exponent of each vector element.
<a href="#">Tan</a>	Computes tangent of each vector element.
<a href="#">Acos</a>	Computes inverse cosine of each vector element.
<a href="#">Asin</a>	Computes inverse sine of each vector element.
<a href="#">Atan</a>	Computes inverse tangent of each vector element.
<a href="#">Atan2</a>	Computes four-quadrant inverse tangent of elements of two vectors.
Hyperbolic Functions	

Function Short Name	Description
<a href="#">Cosh</a>	Computes hyperbolic cosine of each vector element.
<a href="#">Sinh</a>	Computes hyperbolic sine of each vector element.
<a href="#">Tanh</a>	Computes hyperbolic tangent of each vector element.
<a href="#">Acosh</a>	Computes inverse hyperbolic cosine of each vector element.
<a href="#">Asinh</a>	Computes inverse hyperbolic sine of each vector element.
<a href="#">Atanh</a>	Computes inverse hyperbolic tangent of each vector element.
Special Functions	
<a href="#">Erf</a>	Computes the error function value.
<a href="#">Erfc</a>	Computes the complementary error function value.
<a href="#">CdfNorm</a>	Computes the cumulative normal distribution function values of vector elements.
<a href="#">ErfInv</a>	Computes the inverse error function value for each vector element.
<a href="#">ErfcInv</a>	Computes the inverse complementary error function value of vector elements.
<a href="#">CdfNormInv</a>	Computes the inverse cumulative normal distribution function values of vector elements.
Rounding Functions	
<a href="#">Floor</a>	Computes integer value rounded toward minus infinity for each vector element.
<a href="#">Ceil</a>	Computes integer value rounded toward plus infinity for each vector element.
<a href="#">Trunc</a>	Computes integer value rounded toward zero for each vector element.
<a href="#">Round</a>	Computes integer value rounded to nearest for each vector element.
<a href="#">NearbyInt</a>	Computes rounded integer value in current rounding mode for each vector element.
<a href="#">Rint</a>	Computes rounded integer value in current rounding mode for each vector element with inexact result exception raised for each changed value.
<a href="#">Modf</a>	Computes truncated integer value and remaining fraction part for each vector element.



**NOTE.** Do not confuse fixed-accuracy arithmetic functions described here with [common arithmetic functions](#) that have similar functionality but follow different accuracy specifications.

Intel IPP fixed-accuracy arithmetic functions may return status codes of the specific warnings listed in the Table 12-2 below. In this case, the value returned is positive and the computation is continued.

**Table 12-2 Warning Status Codes for Fixed-Accuracy Arithmetic Functions**

	Value	Message
IppStsOverflow	12	Overflow occurred in the operation.
IppStsUnderflow	17	Underflow occurred in the operation.
IppStsSingularity	18	Singularity occurred in the operation.
IppStsDomain	19	Argument is out of the function domain.

See appendix A "[Handling of Special Cases](#)" for more information on function operation in cases when their arguments take on specific values that are outside the range of function definition.



**NOTE.** All functions described in this chapter support in-place operation.

# Arithmetic Functions

## Add

*Performs element by element addition of two vectors.*

### Syntax

```
IppStatus ippsAdd_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, Ipp32s len);

IppStatus ippsAdd_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f* pDst, Ipp32s len);

IppStatus ippsAdd_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2, Ipp32fc* pDst, Ipp32s len);

IppStatus ippsAdd_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2, Ipp64fc* pDst, Ipp32s len);
```

## Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsAdd` is declared in the `ippvm.h` file. This function performs element by element addition of the vectors *pSrc1* and *pSrc2*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavors `ippsAdd_32f_A24` and `ippsAdd_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAdd_64f_A53` and `ippsAdd_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc1[n]) + (pSrc2[n]), 0 \leq n < len.$$

Example 12-1 below shows how to use the function `ippsAdd`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc1</i> , <i>pSrc2</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-1. Using ippsAdd Function

```
IppStatus ippsAdd_32f_A24_sample(void)
{
    const Ipp32f x1[4] = {+4.885, -0.543, -3.809, -4.953};
    const Ipp32f x2[4] = {-0.543, -3.809, -4.953, +4.885};
    Ipp32f y[4];

    IppStatus st = ippsAdd_32f_A24( x1, x2, y, 4 );
    printf(" ippsAdd_32f_A24:\n");

    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsAdd_32f_A24:
x1 = +4.885 -0.543 -3.809 -4.953
x2 = -0.543 -3.809 -4.953 +4.885
y  = +4.342 -4.352 -8.762 -0.068
```

## Sub

*Performs element by element subtraction of one vector from another.*

---

### Syntax

```
IppStatus ippsSub_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsSub_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsSub_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);
```

---

```
IppStatus ippsSub_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsSub` is declared in the `ippvm.h` file. This function performs element by element subtraction of the vector *pSrc2* from the vector *pSrc1*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavors `ippsSub_32f_A24` and `ippsSub_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSub_64f_A53` and `ippsSub_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc1[n]) - (pSrc2[n]), 0 \leq n < len.$$

Example 12-2 below shows how to use the function `ippsSub`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc1</i> , <i>pSrc2</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-2. Using `ippsSub` Function

```

IppStatus ippsSub_32f_A24_sample(void)
{
    const Ipp32f x1[4] = {+4.885, -0.543, -3.809, -4.953};
    const Ipp32f x2[4] = {-0.543, -3.809, -4.953, +4.885};
    Ipp32f y[4];

    IppStatus st = ippsSub_32f_A24( x1, x2, y, 4 );
    printf(" ippsSub_32f_A24:\n");

    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsSub_32f_A24:
x1 = +4.885 -0.543 -3.809 -4.953
x2 = -0.543 -3.809 -4.953 +4.885
y  = +5.428 +3.266 +1.144 -9.838

```

## Sqr

*Performs element by element squaring of the vector.*

---

### Syntax

```

IppStatus ippsSqr_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSqr_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

### Parameters

*pSrc*                                      Pointer to the source vector.



<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

Description

The function `ippsSqr` is declared in the `ippvm.h` file. This function performs element by element squaring of the vector *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsSqr_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsSqr_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc[n])^2, 0 \leq n < len.$$

Example 12-3 below shows how to use the function `ippsSqr`.

Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-3. Using ippsSqr Function

```

IppStatus ippsSqr_32f_A24_sample(void)
{
    const Ipp32f x[4] = {+4.885, -0.543, -3.809, -4.953};
    Ipp32f y[4];
    IppStatus st = ippsSqr_32f_A24( x, y, 4 );
    printf(" ippsSqr_32f_A24:\n");
    printf(" x = %+.3f %+.3f %+.3f %+.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %+.3f %+.3f %+.3f %+.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsSqr_32f_A24:
x = +4.885 -0.543 -3.809 -4.953
y = +23.863 +0.295 +14.508 +24.532

```

## Mul

*Performs element by element multiplication of two vectors.*

---

### Syntax

```

IppStatus ippsMul_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsMul_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsMul_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsMul_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsMul_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

```

```

IppStatus ippsMul_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

IppStatus ippsMul_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

IppStatus ippsMul_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

```

## Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsMul` is declared in the `ippvm.h` file. This function performs element by element multiplication of the vectors *pSrc1* and *pSrc2*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsMul_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsMul_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsMul_32f_A24` and `ippsMul_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsMul_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsMul_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsMul_64f_A53` and `ippsMul_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = (pSrc1[n]) \times (pSrc2[n]), 0 \leq n < len.$

Example 12-4 below shows how to use the function `ippsMul`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> , <code>pSrc2</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 12-4. Using `ippsMul` Function

```

IppStatus ippsMul_32f_A24_sample(void)
{
    const Ipp32f x1[4] = {+4.885, -0.543, -3.809, -4.953};
    const Ipp32f x2[4] = {-0.543, -3.809, -4.953, +4.885};
    Ipp32f y[4];

    IppStatus st = ippsMul_32f_A24( x1, x2, y, 4 );
    printf(" ippsMul_32f_A24:\n");

    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsMul_32f_A24:
x1 = +4.885 -0.543 -3.809 -4.953
x2 = -0.543 -3.809 -4.953 +4.885
y  = -2.653 +2.068 +18.866 -24.195

```

## MulByConj

*Performs element by element multiplication of a vector **a** element and a conjugated vector **b** element.*

---

### Syntax

```
IppStatus ippsMulByConj_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsMulByConj_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsMulByConj_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsMulByConj_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

IppStatus ippsMulByConj_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

IppStatus ippsMulByConj_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsMulByConj` is declared in the `ippvm.h` file. This function performs element by element multiplication of the vector *pSrc1* and the conjugated vector *pSrc2*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsMulByConj_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsMulByConj_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsMulByConj_32fc_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsMulByConj_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsMulByConj_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsMulByConj_64fc_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = pSrc1[n] \times CONJ(pSrc2[n]), 0 \leq n < len.$$

Example 12-5 below shows how to use the function `ippsMulByConj`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> , <code>pSrc2</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 12-5. Using `ippsMulByConj` Function

```

IppStatus ippsMulByConj_32fc_A24_sample(void)
{
    const Ipp32fc x1[4] = {{+2.885,-1.809}, {-0.543,-2.809}};
    const Ipp32fc x2[4] = {{-0.543,-2.809}, {-1.809,-2.809}};
    Ipp32fc y[2];

    IppStatus st = ippsMulByConj_32fc_A24( x1, x2, y, 2 );
    printf(" ippsMulByConj_32fc_A24:\n");
}

```

```

printf(" x1 = %+.3f%+.3f*i   %+.3f%+.3f*i \n", x1[0].re, x1[0].im, x1[1].re, x1[1].im);
printf(" x2 = %+.3f%+.3f*i   %+.3f%+.3f*i \n", x2[0].re, x2[0].im, x2[1].re, x2[1].im);
printf(" y  = %+.3f%+.3f*i   %+.3f%+.3f*i \n", y[0].re, y[0].im, y[1].re, y[1].im);
return st;
}

```

Output results:

ippsMulByConj\_32fc\_A24:

```

x1 = +2.885-1.809*i   -0.543-2.809*i
x2 = -0.543-2.809*i   -1.809-2.809*i
y  = +3.515+9.086*i   +8.873+3.556*i

```

## Conj

*Performs element by element conjugation of the vector.*

---

### Syntax

```

IppStatus ippsConj_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

```

```

IppStatus ippsConj_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsConj` is declared in the `ippvm.h` file. This function performs element by element conjugation of the vector *pSrc* and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsConj_32fc_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsConj_64fc_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = CONJ(pSrc[n]), 0 \leq n < len.$

Example 12-6 below shows how to use the function `ippsConj`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> , <code>pSrc2</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 12-6. Using `ippsConj` Function

```
IppStatus ippsConj_32fc_A24_sample(void)
{
    const Ipp32fc x[2] = {{+2.885,-1.809}, {-0.543,-2.809}};
    Ipp32fc y[2];

    IppStatus st = ippsConj_32fc_A24( x, y, 2 );
    printf(" ippsConj_32fc_A24:\n");

    printf(" x = %+.3f%+.3f*i   %+.3f%+.3f*i \n", x[0].re, x[0].im, x[1].re, x[1].im);
    printf(" y = %+.3f%+.3f*i   %+.3f%+.3f*i \n", y[0].re, y[0].im, y[1].re, y[1].im);
    return st;
}
```

Output results:

```
ippsConj_32fc_A24:
x = +2.885-1.809*i   -0.543-2.809*i
y = +2.885+1.809*i   -0.543+2.809*i
```



## Abs

Computes the absolute value of vector elements.

### Syntax

```
IppStatus ippsAbs_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAbs_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAbs_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAbs_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAbs_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAbs_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAbs_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAbs_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsAbs` is declared in the `ippvm.h` file. This function computes the absolute value of the vector *pSrc* elements and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsAbs_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAbs_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAbs_32f_A24` and `ippsAbs_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAbs_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsAbs_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAbs_64f_A53` and `ippsAbs_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = |pSrc1[n]|, 0 \leq n < len.$$

Example 12-7 below shows how to use the function `ippsAbs`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> , <code>pSrc2</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 12-7. Using `ippsAbs` Function

```

IppStatus ippsAbs_32fc_A24_sample(void)
{
    const Ipp32fc x[2] = {{+2.885,-1.809}, {-0.543,-2.809}};
    Ipp32fc y[2];

    IppStatus st = ippsAbs_32fc_A24( x, y, 2 );
    printf(" ippsAbs_32fc_A24:\n");

    printf(" x = %.3f%.3f*i   %.3f%.3f*i \n", x[0].re, x[0].im, x[1].re, x[1].im);
    printf(" y = %.3f           %.3f%           \n", y[0],           y[1]);
    return st;
}

```

Output results:

```

ippsAbs_32fc_A24:
x = +2.885-1.809*i   -0.543-2.809*i
y = +3.405          +2.861

```

## Arg

Computes the argument of vector elements.

### Syntax

```
IppStatus ippsArg_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsArg_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsArg_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsArg_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsArg_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsArg_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsArg` is declared in the `ippvm.h` file. This function computes the argument of the vector *pSrc* elements and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsArg_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsArg_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsArg_32fc_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsArg_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsArg_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsArg_64fc_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \phi(pSrc1[n]), 0 \leq n < len.$$

Example 12-8 below shows how to use the function `ippsArg`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> , <code>pSrc2</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 12-8. Using `ippsArg` Function

```

IppStatus ippsArg_32fc_A24_sample(void)
{
    const Ipp32fc x[2] = {{+2.885,-1.809}, {-0.543,-2.809}};
    Ipp32fc y[2];
    IppStatus st = ippsArg_32fc_A24( x, y, 2 );
    printf(" ippsArg_32fc_A24:\n");

    printf(" x = %.3f%.3f*i   %.3f%.3f*i \n", x[0].re, x[0].im, x[1].re, x[1].im);
    printf(" y = %.3f      %.3f      \n", y[0],          y[1]);
    return st;
}

```

Output results:

```

ippsArg_32fc_A24:
x = +2.885-1.809*i   -0.543-2.809*i
y = -0.560         -1.762

```

## Power and Root Functions

### Inv

Computes inverse value of each vector element.

#### Syntax

```
IppStatus ippsInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInv_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

#### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

#### Description

The function `ippsInv` is declared in the `ippvm.h` file. This function computes the inverse value of each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInv_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = 1/(pSrc[n]), 0 \leq n < len.$

Example 12-9 shows how to use the function `ippsInv`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppsStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <code>pSrc</code> is equal to 0.

**Example 12-9. Using ippsInv Function**

```

IppStatus ippsInv_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-9.975, 1.272, -6.134, 6.175};
    Ipp32f        y[4];
    IppStatus st = ippsInv_32f_A21( x, y, 4 );
    printf(" ippsInv_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsInv_32f_A21:
x = -9.975 1.272 -6.134 6.175
y = -0.100 0.786 -0.163 0.162

```

**Div**

*Divides each element of the first vector by  
corresponding element of the second vector.*

---

**Syntax**

```

IppStatus ippsDiv_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsDiv_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsDiv_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsDiv_64f_A26 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsDiv_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

```

```

IppStatus ippsDiv_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsDiv_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsDiv_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsDiv_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsDiv_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

IppStatus ippsDiv_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

IppStatus ippsDiv_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

```

## Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsDiv` is declared in the `ippvm.h` file. This function divides each element of the vector *pSrc1* by the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsDiv_32f_A11` and `ippsDiv_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsDiv_32f_A21` and `ippsDiv_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsDiv_32f_A24` and `ippsDiv_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:



function flavors `ippsDiv_64f_A26` and `ippsDiv_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsDiv_64f_A50` and `ippsDiv_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsDiv_64f_A53` and `ippsDiv_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc1[n]) / (pSrc2[n]), 0 \leq n < len.$$

Example 12-10 shows how to use the function `ippsDiv`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc1</i> or <i>pSrc2</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppsStsSingularity</code>	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of <i>pSrc2</i> is equal to 0.

## Example 12-10. Using ippsDiv Function

```
IppStatus ippsDiv_32f_A21_sample(void)
{
    const Ipp32f x1[4] = {599.088, 735.034, 572.448, 151.640};
    const Ipp32f x2[4] = {385.297, 609.005, 361.403, 225.182};
    Ipp32f        y[4];

    IppStatus st = ippsDiv_32f_A21( x1, x2, y, 4 );
    printf(" ippsDiv_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsDiv_32f_A21:
x1 = 599.088 735.034 572.448 151.640
x2 = 385.297 609.005 361.403 225.182
y  = 1.555 1.207 1.584 0.673
```

## Sqrt

Computes square root of each vector element.

### Syntax

```
IppStatus ippsSqrt_32f_A11 ( const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSqrt_32f_A21 ( const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSqrt_32f_A24 ( const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSqrt_64f_A26 ( const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSqrt_64f_A50 ( const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSqrt_64f_A53 ( const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

```

IppStatus ippsSqrt_32fc_A11 ( const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);

IppStatus ippsSqrt_32fc_A21 ( const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);

IppStatus ippsSqrt_32fc_A24 ( const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);

IppStatus ippsSqrt_64fc_A26 ( const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

IppStatus ippsSqrt_64fc_A50 ( const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

IppStatus ippsSqrt_64fc_A53 ( const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsSqrt` is declared in the `ippvm.h` file. This function computes square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsSqrt_32f_A11` and `ippsSqrt_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsSqrt_32f_A21` and `ippsSqrt_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsSqrt_32f_A24` and `ippsSqrt_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSqrt_64f_A26` and `ippsSqrt_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsSqrt_64f_A50` and `ippsSqrt_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsSqrt_64f_A53` and `ippsSqrt_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc[n])^{1/2}, 0 \leq n < len.$$

Example 12-11 below shows how to use the function `ippsSqrt`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is less than 0.

**Example 12-11. Using ippsSqrt Function**

```

IppStatus ippsSqrt_32f_A21_sample(void)
{
    const Ipp32f x[4] = {5850.093, 4798.730, 3502.915, 8959.624};
    Ipp32f      y[4];
    IppStatus st = ippsSqrt_32f_A21( x, y, 4 );
    printf(" ippsSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsSqrt_32f_A21:
x = 5850.093 4798.730 3502.915 8959.624
y = 76.486 69.273 59.185 94.655

```

**InvSqrt**

*Computes inverse square root of each vector element.*

---

**Syntax**

```

IppStatus ippsInvSqrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInvSqrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

**Parameters**

*pSrc*                                      Pointer to the source vector.

*pDst* Pointer to the destination vector.  
*len* Number of elements in the vectors.

## Description

The function `ippsInvSqrt` is declared in the `ippvm.h` file. This function computes inverse square root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsInvSqrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInvSqrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInvSqrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInvSqrt_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsInvSqrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInvSqrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc[n])^{-1/2}, 0 \leq n < len.$$

Example 12-12 below shows how to use the function `ippsInvSqrt`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppsStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> is less than 0.

`IppStsSingularity` Indicates a warning that the argument is the singularity point, that is, at least one of the elements of `pSrc` is equal to 0.

### Example 12-12. Using `ippsInvSqrt` Function

```
IppStatus ippsInvSqrt_32f_A21_sample(void)
{
    const Ipp32f x[4] = {7105.043, 5135.398, 3040.018, 149.944};
    Ipp32f      y[4];
    IppStatus st = ippsInvSqrt_32f_A21( x, y, 4 );
    printf(" ippsInvSqrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsInvSqrt_32f_A21:
x = 7105.043 5135.398 3040.018 149.944
y = 0.012 0.014 0.018 0.082
```

## Cbrt

Computes cube root of each vector element.

### Syntax

```
IppStatus ippsCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCbrt_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsCbrt` is declared in the `ippvm.h` file. This function computes cube root of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCbrt_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc[n])^{1/3}, 0 \leq n < len.$$

Example 12-13 below shows how to use the function `ippsCbrt`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.



**Example 12-13. Using ippsCbrt Function**

```

IppStatus ippsCbrt_32f_A21_sample(void)
{
    const Ipp32f x[4] = {6456.801, 4932.096, -6517.838, 7178.869};
    Ipp32f      y[4];
    IppStatus st = ippsCbrt_32f_A21( x, y, 4 );
    printf(" ippsCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsCbrt_32f_A21:
x = 6456.801 4932.096 -6517.838 7178.869
y = 18.621 17.022 -18.680 19.291

```

**InvCbrt**

*Computes inverse cube root of each vector element.*

---

**Syntax**

```

IppStatus ippsInvCbrt_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsInvCbrt_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

**Parameters**

*pSrc*                                      Pointer to the source vector.

*pDst* Pointer to the destination vector.  
*len* Number of elements in the vectors.

## Description

The function `ippsInvCbrt` is declared in the `ippvm.h` file. This function computes inverse cube root of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsInvCbrt_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsInvCbrt_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsInvCbrt_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsInvCbrt_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsInvCbrt_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsInvCbrt_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = (pSrc[n])^{-1/3}, 0 \leq n < len.$$

Example 12-14 below shows how to use the function `ippsInvCbrt`.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when *pSrc* or *pDst* pointer is NULL.  
`ippStsSizeErr` Indicates an error when *len* is less than or equal to 0.  
`IppsStsSingularity` Indicates a warning that the argument is the singularity point, that is, at least one of the elements of *pSrc* is equal to 0.

**Example 12-14. Using ippsInvCbrt Function**

```

IppStatus ippsInvCbrt_32f_A21_sample(void)
{
    const Ipp32f x[4] = {914.120, 3644.584, 1473.214, 1659.070};
    Ipp32f        y[4];
    IppStatus st = ippsInvCbrt_32f_A21( x, y, 4 );
    printf(" ippsInvCbrt_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsInvCbrt_32f_A21:
x = 914.120 3644.584 1473.214 1659.070
y = 0.103 0.065 0.088 0.084

```

**Pow2o3**

*Computes the value of each vector element raised to the power of 2/3.*

---

**Syntax**

```

IppStatus ippsPow2o3_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow2o3_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow2o3_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow2o3_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsPow2o3_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsPow2o3_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

**Parameters**

*pSrc*                                      Pointer to the source vector.

*pDst* Pointer to the destination vector.  
*len* Number of elements in the vectors.

## Description

The function `ippsPow2o3` is declared in the `ippvm.h` file. This function computes the value of each vector element of the vector *pSrc* raised to  $2/3$  power and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsPow2o3_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsPow2o3_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsPow2o3_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsPow2o3_64f_A26` guarantees 26 correctly rounded bits of significand, or  $6.7E+7$  ulps, or approximately 8 exact decimal digits;

function flavor `ippsPow2o3_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsPow2o3_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

## Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error when *pSrc* or *pDst* pointer is NULL.  
`ippStsSizeErr` Indicates an error when *len* is less than or equal to 0.

## Pow3o2

*Computes the value of each vector element raised to the power of  $3/2$ .*

---

### Syntax

```
ippStatus ippsPow3o2_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
```

```

IppStatus ippsPow3o2_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow3o2_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsPow3o2_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsPow3o2_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsPow3o2_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsPow3o2` is declared in the `ippvm.h` file. This function computes the value of each vector element of the vector *pSrc* raised to  $3/2$  power and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsPow3o2_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsPow3o2_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsPow3o2_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsPow3o2_64f_A26` guarantees 26 correctly rounded bits of significand, or  $6.7E+7$  ulps, or approximately 8 exact decimal digits;

function flavor `ippsPow3o2_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsPow3o2_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is less than 0.

## Pow

*Raises each element of the first vector to the power of corresponding element of the second vector.*

---

### Syntax

```

IppStatus ippsPow_32f_All (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsPow_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsPow_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsPow_64f_A26 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsPow_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsPow_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsPow_32fc_All (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsPow_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsPow_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc* pSrc2,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsPow_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);

```

```
IppStatus ippsPow_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);
```

```
IppStatus ippsPow_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc* pSrc2,
Ipp64fc* pDst, Ipp32s len);
```

## Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsPow` is declared in the `ippvm.h` file. This function raises each element of vector *pSrc1* to the power of the corresponding element of the vector *pSrc2* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsPow_32f_A11` and `ippsPow_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsPow_32f_A21` and `ippsPow_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsPow_32f_A24` and `ippsPow_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsPow_64f_A26` and `ippsPow_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsPow_64f_A50` and `ippsPow_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsPow_64f_A53` and `ippsPow_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Note that for `ippsPow` complex functions there may be argument ranges where the accuracy specification does not hold.

The computation is performed as follows:

$pDst[n] = (pSrc1[n])^{pSrc2[n]}, 0 \leq n < len.$

Example 12-15 below shows how to use the function `ippsPow`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> or <code>pSrc2</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one pair of the source elements meets the following condition: element of <code>pSrc1</code> is finite, less than 0, and element of <code>pSrc2</code> is finite, non-integer.
<code>IppStsSingularity</code>	In real functions, indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of <code>pSrc1</code> is equal to 0, and element of <code>pSrc2</code> is integer and less than 0.



**Example 12-15. Using ippsPow Function**

```

IppStatus ippsPow_32f_A21_sample(void)
{
    const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};
    const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
    Ipp32f      y[4];
    IppStatus st = ippsPow_32f_A21( x1, x2, y, 4 );
    printf(" ippsPow_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0],  y[1],  y[2],  y[3]);
    return st;
}

```

Output results:

```

ippsPow_32f_A21:
x1 = 0.483 0.565 0.776 0.252
x2 = 0.823 0.991 0.411 0.692
y  = 0.549 0.568 0.901 0.386

```

**Powx**

*Raises each element of a vector to a constant power.*

---

**Syntax**

```

IppStatus ippsPowx_32f_A11 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
Ipp32f* pDst, Ipp32s len);

IppStatus ippsPowx_32f_A21 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
Ipp32f* pDst, Ipp32s len);

IppStatus ippsPowx_32f_A24 (const Ipp32f* pSrc1, const Ipp32f ConstValue,
Ipp32f* pDst, Ipp32s len);

```

```

IppStatus ippsPowx_64f_A26 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
Ipp64f* pDst, Ipp32s len);

IppStatus ippsPowx_64f_A50 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
Ipp64f* pDst, Ipp32s len);

IppStatus ippsPowx_64f_A53 (const Ipp64f* pSrc1, const Ipp64f ConstValue,
Ipp64f* pDst, Ipp32s len);

IppStatus ippsPowx_32fc_A11 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsPowx_32fc_A21 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsPowx_32fc_A24 (const Ipp32fc* pSrc1, const Ipp32fc ConstValue,
Ipp32fc* pDst, Ipp32s len);

IppStatus ippsPowx_64fc_A26 (const Ipp64fc* pSrc1, const Ipp64fc ConstValue,
Ipp64fc* pDst, Ipp32s len);

IppStatus ippsPowx_64fc_A50 (const Ipp64fc* pSrc1, const Ipp64fc ConstValue,
Ipp64fc* pDst, Ipp32s len);

IppStatus ippsPowx_64fc_A53 (const Ipp64fc* pSrc1, const Ipp64fc ConstValue,
Ipp64fc* pDst, Ipp32s len);

```

## Parameters

<i>pSrc1</i>	Pointer to the source vector.
<i>ConstValue</i>	Constant value.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsPowx` is declared in the `ippvm.h` file. This function raises each element of the vector *pSrc1* to the constant power *ConstValue* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsPowx_32f_A11` and `ippsPowx_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsPowx_32f_A21` and `ippsPowx_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsPowx_32f_A24` and `ippsPowx_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsPowx_64f_A26` and `ippsPowx_64fc_A26` guarantee 26 correctly rounded bits of significand, or  $6.7E+7$  ulps, or approximately 8 exact decimal digits;

function flavors `ippsPowx_64f_A50` and `ippsPowx_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsPowx_64f_A53` and `ippsPowx_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

Note that for `ippsPowx` complex functions there may be argument ranges where the accuracy specification does not hold.

The computation is performed as follows:

$$pDst[n] = (pSrc[n])^{ConstValue}, 0 \leq n < len.$$

Example 12-16 below shows how to use the function `ippsPowx`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>ippStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one pair of the elements meets the following condition: element of <code>pSrc1</code> is finite, less than 0, and <code>ConstValue</code> is finite, non-integer.
<code>IppStsSingularity</code>	In real functions, indicates a warning that the argument is the singularity point, that is, at least one pair of the elements is as follows: element of <code>pSrc1</code> is equal to 0, and <code>ConstValue</code> is integer and less than 0.

### Example 12-16. Using `ippsPowx` Function

```
IppStatus ippsPowx_32f_A21_sample(void)
```

```
{  
    const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252};  
  
    const Ipp32f x2 = 0.823;  
  
    Ipp32f          y[4];  
  
    IppStatus st = ippsPowx_32f_A21( x1, x2, y, 4 );  
  
    printf(" ippsPowx_32f_A21:\n");  
  
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);  
  
    printf(" x2 = %.3f \n", x2);  
  
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);  
  
    return st;  
  
}
```

Output results:

```
ippsPowx_32f_A21:  
  
x1 = 0.483 0.565 0.776 0.252  
  
x2 = 0.823  
  
y  = 0.549 0.568 0.901 0.386
```

## Hypot

*Computes a square root of sum of two squared elements.*

---

### Syntax

```
IppStatus ippsHypot_32f_A11 (const Ipp32f* pSrc1, const Ipp32f pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsHypot_32f_A21 (const Ipp32f* pSrc1, const Ipp32f pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsHypot_32f_A24 (const Ipp32f* pSrc1, const Ipp32f pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsHypot_64f_A26 (const Ipp64f* pSrc1, const Ipp64f pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsHypot_64f_A50 (const Ipp64f* pSrc1, const Ipp64f pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsHypot_64f_A53 (const Ipp64f* pSrc1, const Ipp64f pSrc2, Ipp64f*
pDst, Ipp32s len);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsHypot` is declared in the `ippvm.h` file. This function computes square of each element of the *pSrc1* and *pSrc2* vectors, sums corresponding elements, computes square roots of each sum and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsHypot_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsHypot_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippSHypot_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippSHypot_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippSHypot_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippSHypot_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = ((pSrc1[n])^2 + (pSrc2[n])^2)^{1/2}, 0 \leq n < len.$$

Example 12-17 below shows how to use the function `ippSHypot`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> or <code>pSrc2</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

**Example 12-17. Using ippsHypot Function**

```

IppStatus ippsHypot_32f_A21_sample(void)
{
    const Ipp32f x1[4] = {0.483, 0.565, 0.776, 0.252}
    const Ipp32f x2[4] = {0.823, 0.991, 0.411, 0.692};
    Ipp32f y[4];

    IppStatus st = ippsHypot_32f_A21( x1, x2, y, 4 );
    printf(" ippsHypot_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsHypot_32f_A21:
x1 = 0.483 0.565 0.776 0.252
x2 = 0.823 0.991 0.411 0.692
y  = 0.954 1.141 0.878 0.736

```

## Exponential and Logarithmic Functions

### Exp

Raises  $e$  to the power of each vector element.

**Syntax**

```

IppStatus ippsExp_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExp_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExp_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);

```

```

IppStatus ippsExp_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExp_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExp_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExp_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsExp_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsExp_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsExp_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsExp_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsExp_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsExp` is declared in the `ippvm.h` file. This function raises *e* to the power of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsExp_32f_A11` and `ippsExp_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsExp_32f_A21` and `ippsExp_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsExp_32f_A24` and `ippsExp_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsExp_64f_A26` and `ippsExp_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsExp_64f_A50` and `ippsExp_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;



function flavors `ippsExp_64f_A53` and `ippsExp_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = e^{pSrc[n]}, 0 \leq n < len.$$

Example 12-18 below shows how to use the function `ippsExp`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppsStsOverflow</code>	In real functions, indicates a warning that the function overflows, that is, at least one of elements of <i>pSrc</i> is greater than $\text{Ln}(\text{FPMAX})$ , where <code>FPMAX</code> is the maximum representable floating-point number.
<code>IppsStsUnderflow</code>	In real functions, indicates a warning that the function underflows, that is, at least one of elements of <i>pSrc</i> is less than $\text{Ln}(\text{FPMIN})$ , where <code>FPMIN</code> is the minimum positive floating-point value.

## Example 12-18. Using ippsExp Function

```

IppStatus ippsExp_32f_A21_sample(void)
{
    const Ipp32f x[4] = {4.885, -0.543, -3.809, -4.953};
    Ipp32f      y[4];
    IppStatus st = ippsExp_32f_A21( x, y, 4 );
    printf(" ippsExp_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsExp_32f_A21:
x = 4.885 -0.543 -3.809 -4.953
y = 132.324 0.581 0.022 0.007

```

## Expm1

*Computes  $e$  raised to the power of each vector element and decreased by 1.*

---

### Syntax

```

IppStatus ippsExpml_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExpml_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExpml_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsExpml_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExpml_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsExpml_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

### Parameters

*pSrc*                                      Pointer to the source vector.

<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsExpml` is declared in the `ippvm.h` file. This function computes  $e$  raised to the power of each vector element of *pSrc* and decreased by 1, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsExpml_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsExpml_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsExpml_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsExpml_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsExpml_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsExpml_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppsStsOverflow</code>	Indicates a warning that the function overflows, that is, at least one of elements of <i>pSrc</i> is greater than $\text{Ln}(\text{FPMAX})$ , where <code>FPMAX</code> is the maximum representable floating-point number.

## Ln

*Computes natural logarithm of each vector element.*

---

### Syntax

```

IppStatus ippsLn_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLn_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLn_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLn_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLn_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLn_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLn_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLn_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLn_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsLn_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsLn_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsLn_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);

```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsLn` is declared in the `ippvm.h` file. This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsLn_32f_A11` and `ippsLn_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsLn_32f_A21` and `ippsLn_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsLn_32f_A24` and `ippsLn_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsLn_64f_A26` and `ippsLn_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsLn_64f_A50` and `ippsLn_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsLn_64f_A53` and `ippsLn_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \log_e(pSrc[n]), 0 \leq n < len.$

Example 12-19 below shows how to use the function `ippsLn`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is less than 0.
<code>IppStsSingularity</code>	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of <code>pSrc</code> is equal to 0.

## Example 12-19. Using ippsLn Function

```
IppStatus ippsLn_32f_A21_sample(void)
{
    const Ipp32f x[4] = {0.188, 3.841, 5.363, 5.755};
    Ipp32f        y[4];
    IppStatus st = ippsLn_32f_A21( x, y, 4 );
    printf(" ippsLn_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

ippsLn\_32f\_A21:

x = 0.188 3.841 5.363 5.755

y = -1.670 1.346 1.680 1.750

## Log10

*Computes common logarithm of each vector element.*

---

### Syntax

```
IppStatus ippsLog10_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog10_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog10_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog10_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog10_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog10_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog10_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
```

```

IppStatus ippsLog10_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);

IppStatus ippsLog10_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);

IppStatus ippsLog10_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

IppStatus ippsLog10_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

IppStatus ippsLog10_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsLog10` is declared in the `ippvm.h` file. This function computes a natural logarithm of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsLog10_32f_A11` and `ippsLog10_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsLog10_32f_A21` and `ippsLog10_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsLog10_32f_A24` and `ippsLog10_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsLog10_64f_A26` and `ippsLog10_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsLog10_64f_A50` and `ippsLog10_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsLog10_64f_A53` and `ippsLog10_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \log_{10}(pSrc[n]), 0 \leq n < len.$$

Example 12-20 below shows how to use the function `ippsLog10`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is less than 0.
<code>IppStsSingularity</code>	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of <code>pSrc</code> is equal to 0.

## Example 12-20 . Using `ippsLog10` Function

```

IppStatus ippsLog10_32f_A21_sample(void)
{
    const Ipp32f x[4] = {6.057, 6.111, 1.746, 6.664};
    Ipp32f        y[4];

    IppStatus st = ippsLog10_32f_A21( x, y, 4 );
    printf(" ippsLog10_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsLog10_32f_A21:
x = 6.057 6.111 1.746 6.664
y = 0.782 0.786 0.242 0.824

```



## Log1p

*Computes natural logarithm of each vector element decreased by 1.*

---

### Syntax

```
IppStatus ippsLog1p_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog1p_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog1p_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsLog1p_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog1p_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsLog1p_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsLog1p` is declared in the `ippvm.h` file. This function computes a natural logarithm of each vector element of *pSrc* decreased by 1, and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsLog1p_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsLog1p_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsLog1p_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsLog1p_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsLog1p_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsLog1p_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is less than -1.
<code>IppStsSingularity</code>	Indicates a warning that the argument is the singularity point, that is, at least one of the elements of <code>pSrc</code> is equal to -1.

# Trigonometric Functions

## Cos

*Computes cosine of each vector element.*

---

### Syntax

```

IppStatus ippsCos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCos_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCos_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCos_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCos_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);

```

```
IppStatus ippsCos_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCos_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCos_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsCos` is declared in the `ippvm.h` file. This function computes a cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsCos_32f_A11` and `ippsCos_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsCos_32f_A21` and `ippsCos_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsCos_32f_A24` and `ippsCos_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsCos_64f_A26` and `ippsCos_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsCos_64f_A50` and `ippsCos_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsCos_64f_A53` and `ippsCos_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \cos(pSrc[n]), 0 \leq n < len.$$

Example 12-21 below shows how to use the function `ippsCos`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> is equal to $\pm \text{INF}$ .

## Example 12-21. Using `ippsCos` Function

```
IppStatus ippsCos_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-984.222, -2957.549, -8859.218, 2153.691};
    Ipp32f        y[4];
    IppStatus st = ippsCos_32f_A21( x, y, 4 );
    printf(" ippsCos_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsCos_32f_A21:
x = -984.222 -2957.549 -8859.218 2153.691
y = -0.619 -0.258 0.997 0.129
```

## Sin

*Computes sine of each vector element.*

---

### Syntax

```
IppStatus ippsSin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
```

```

IppStatus ippsSin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSin_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSin_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSin_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSin_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSin_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsSin_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsSin_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsSin` is declared in the `ippvm.h` file. This function computes a sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsSin_32f_A11` and `ippsSin_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsSin_32f_A21` and `ippsSin_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsSin_32f_A24` and `ippsSin_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSin_64f_A26` and `ippsSin_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsSin_64f_A50` and `ippsSin_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsSin_64f_A53` and `ippsSin_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \sin(pSrc[n]), 0 \leq n < len.$$

Example 12-22 below shows how to use the function `ippsSin`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppsStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is equal to $\pm \text{INF}$ .

## Example 12-22. Using `ippsSin` Function

```
IppStatus ippsSin_32f_A21_sample(void)
{
    const Ipp32f x[4] = {5666.372, 6052.125, 397.656, -3960.997};
    Ipp32f        y[4];

    IppStatus st = ippsSin_32f_A21( x, y, 4 );
    printf(" ippsSin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsSin_32f_A21:
x = 5666.372 6052.125 397.656 -3960.997
y = -0.873 0.988 0.970 -0.524
```

## SinCos

Computes sine and cosine of each vector element.

### Syntax

```
IppStatus ippsSinCos_32f_A11 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
pDst2, Ipp32s len);
```

```
IppStatus ippsSinCos_32f_A21 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
pDst2, Ipp32s len);
```

```
IppStatus ippsSinCos_32f_A24 (const Ipp32f* pSrc, const Ipp32f* pDst1, Ipp32f*
pDst2, Ipp32s len);
```

```
IppStatus ippsSinCos_64f_A26 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
pDst2, Ipp32s len);
```

```
IppStatus ippsSinCos_64f_A50 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
pDst2, Ipp32s len);
```

```
IppStatus ippsSinCos_64f_A53 (const Ipp64f* pSrc, const Ipp64f* pDst1, Ipp64f*
pDst2, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the first source vector.
<i>pDst1</i>	Pointer to the destination vector for sine values.
<i>pDst2</i>	Pointer to the destination vector for cosine values.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsSinCos` is declared in the `ippvm.h` file. This function computes sine of each element of *pSrc* and stores the result in the corresponding element of *pDst1*; computes cosine of each element of *pSrc* and stores the result in the corresponding element of *pDst2*.

For single precision data:

function flavor `ippsSinCos_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsSinCos_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsSinCos_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsSinCos_64f_A26` guarantees 26 correctly rounded bits of significand, or  $6.7E+7$  ulps, or approximately 8 exact decimal digits;

function flavor `ippsSinCos_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsSinCos_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst1[n] = \sin(pSrc[n]), \quad pDst2[n] = \cos(pSrc[n]), \quad 0 \leq n < len.$$

Example 12-23 below shows how to use the function `ippsSinCos`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pDst1</i> or <i>pDst2</i> or <i>pSrc</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the <i>pSrc</i> elements is equal to $\pm \text{INF}$ .



**Example 12-23. Using ippsSinCos Function**

```

IppStatus ippsSinCos_32f_A21_sample(void)
{
    const Ipp32f x[4] = {3857.845, -3939.024, -1468.856, -8592.486};
    Ipp32f      y1[4];
    Ipp32f      y2[4];

    IppStatus st = ippsSinCos_32f_A21( x, y1, y2, 4 );
    printf(" ippsSinCos_32f_A21:\n");
    printf(" x  = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
    return st;
}

```

Output results:

```

ippsSinCos_32f_A21:
x  = 3857.845 -3939.024 -1468.856 -8592.486
y1 = -0.031 0.508 0.987 0.228
y2 = 1.000 0.861 0.161 -0.974

```

**CIS**

*Computes complex exponent of each vector element.*

---

**Syntax**

```

IppStatus ippsCIS_32fc_A11 (const Ipp32f* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCIS_32fc_A21 (const Ipp32f* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCIS_32fc_A24 (const Ipp32f* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCIS_64fc_A26 (const Ipp64f* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCIS_64fc_A50 (const Ipp64f* pSrc, Ipp64fc* pDst, Ipp32s len);

```

```
IppStatus ippsCIS_64fc_A53 (const Ipp64f* pSrc, Ipp64fc* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsCIS` is declared in the `ippvm.h` file. This function computes a complex exponent of each vector element of *pSrc* and stores the result in the corresponding element of the vector *pDst*.

For single precision data:

function flavor `ippsCIS_32fc_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCIS_32fc_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCIS_32fc_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCIS_64fc_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsCIS_64fc_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCIS_64fc_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of the <i>pSrc</i> elements is equal to $\pm\text{INF}$ .

## Tan

Computes tangent of each vector element.

### Syntax

```
IppStatus ippsTan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTan_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTan_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTan_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTan_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTan_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsTan_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsTan_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsTan` is declared in the `ippvm.h` file. This function computes the tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsTan_32f_A11` and `ippsTan_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsTan_32f_A21` and `ippsTan_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippSTan_32f_A24` and `ippSTan_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippSTan_64f_A26` and `ippSTan_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippSTan_64f_A50` and `ippSTan_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippSTan_64f_A53` and `ippSTan_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \tan(pSrc[n]), 0 \leq n < len.$$

Example 12-24 below shows how to use the function `ippSTan`.

## Return Values

<code>ippSTsNoErr</code>	Indicates no error.
<code>ippSTsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippSTsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppSTsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is equal to $\pm \text{INF}$ .

**Example 12-24. Using ippsTan Function**

```

IppStatus ippsTan_32f_A21_sample(void)
{
    const Ipp32f x[4] = {7519.456, 4533.524, 9118.015, 8514.359};
    Ipp32f        y[4];
    IppStatus st = ippsTan_32f_A21( x, y, 4 );
    printf(" ippsTan_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsTan_32f_A21:
x = 7519.456 4533.524 9118.015 8514.359
y = -18.656 0.209 2.028 0.750

```

**Acos**

Computes inverse cosine of each vector element.

**Syntax**

```

IppStatus ippsAcos_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcos_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcos_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcos_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcos_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcos_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcos_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAcos_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);

```

```
IppStatus ippsAcos_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAcos_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAcos_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAcos_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsAcos` is declared in the `ippvm.h` file. This function computes the inverse cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAcos_32f_A11` and `ippsAcos_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAcos_32f_A21` and `ippsAcos_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAcos_32f_A24` and `ippsAcos_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAcos_64f_A26` and `ippsAcos_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAcos_64f_A50` and `ippsAcos_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAcos_64f_A53` and `ippsAcos_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \text{acos}(pSrc[n]), 0 \leq n < len.$$

Example 12-25 below shows how to use the function `ippsAcos`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> has an absolute value greater than 1.

## Example 12-25. Using `ippsAcos_32f_A21` Function

```
IppStatus ippsAcos_32f_A21_sample(void)
{
    const Ipp32f x[4] = {0.079, -0.715, -0.076, -0.529};

    Ipp32f      y[4];

    IppStatus st = ippsAcos_32f_A21( x, y, 4 );

    printf(" ippsAcos_32f_A21:\n");

    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);

    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);

    return st;
}
```

Output results:

```
ippsAcos_32f_A21:

x = 0.079 -0.715 -0.076 -0.529

y = 1.492  2.368  1.647  2.129
```

## Asin

Computes inverse sine of each vector element.

### Syntax

```
IppStatus ippsAsin_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsin_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsin_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsin_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsin_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsin_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsin_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsin_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsin_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAsin_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAsin_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAsin_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsAsin` is declared in the `ippvm.h` file. This function computes the inverse sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAsin_32f_A11` and `ippsAsin_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAsin_32f_A21` and `ippsAsin_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;



function flavors `ippsAsin_32f_A24` and `ippsAsin_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAsin_64f_A26` and `ippsAsin_64fc_A26` guarantee 26 correctly rounded bits of significand, or  $6.7E+7$  ulps, or approximately 8 exact decimal digits;

function flavors `ippsAsin_64f_A50` and `ippsAsin_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAsin_64f_A53` and `ippsAsin_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

```
pDst[n] = asin(pSrc[n]), 0 ≤ n < len.
```

Example 12-26 below shows how to use the function `ippsAsin`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <i>pSrc</i> has an absolute value greater than 1.

## Example 12-26. Using ippsAsin Function

```

IppStatus ippsAsin_32f_A21_sample(void)
{
    const Ipp32f x[4] = {0.724, -0.581, 0.559, 0.687};
    Ipp32f      y[4];
    IppStatus st = ippsAsin_32f_A21( x, y, 4 );
    printf(" ippsAsin_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsAsin_32f_A21:
x = 0.724 -0.581 0.559 0.687
y = 0.810 -0.620 0.594 0.758

```

## Atan

Computes inverse tangent of each vector element.

### Syntax

```

IppStatus ippsAtan_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtan_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtan_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtan_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtan_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtan_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtan_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtan_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtan_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtan_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);

```

```
IppStatus ippsAtan_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAtan_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsAtan` is declared in the `ippvm.h` file. This function computes the inverse tangent of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAtan_32f_A11` and `ippsAtan_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAtan_32f_A21` and `ippsAtan_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAtan_32f_A24` and `ippsAtan_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAtan_64f_A26` and `ippsAtan_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAtan_64f_A50` and `ippsAtan_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAtan_64f_A53` and `ippsAtan_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \text{atan}(pSrc[n]), 0 \leq n < len.$$

Example 12-27 below shows how to use the function `ippsAtan`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.

`ippStsSizeErr` Indicates an error when `len` is less than or equal to 0.

## Example 12-27. Using `ippsAtan` Function

```
IppStatus ippsAtan_32f_A21_sample(void)

{
    const Ipp32f x[4] = {0.994, 0.999, 0.223, -0.215};

    Ipp32f      y[4];

    IppStatus st = ippsAtan_32f_A21( x, y, 4 );

    printf(" ippsAtan_32f_A21:\n");

    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);

    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);

    return st;
}
```

Output results:

```
ippsAtan_32f_A21:

x = 0.994 0.999 0.223 -0.215

y = 0.782 0.785 0.219 -0.212
```

## Atan2

*Computes four-quadrant inverse tangent of elements of two vectors.*

---

### Syntax

```
IppStatus ippsAtan2_32f_A11 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsAtan2_32f_A21 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsAtan2_32f_A24 (const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f*
pDst, Ipp32s len);

IppStatus ippsAtan2_64f_A26 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsAtan2_64f_A50 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);

IppStatus ippsAtan2_64f_A53 (const Ipp64f* pSrc1, const Ipp64f* pSrc2, Ipp64f*
pDst, Ipp32s len);
```

### Parameters

<i>pSrc1</i>	Pointer to the first source vector.
<i>pSrc2</i>	Pointer to the second source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsAtan2` is declared in the `ippvm.h` file. This function computes the angle between the  $x$  axis and the line from the origin to the point  $(x, y)$ , for each element of *pSrc1* as a  $y$  (the ordinate) and corresponding element of *pSrc2* as an  $x$  (the abscissa), and stores the result in the corresponding element of *pDst*. The result angle varies from  $-\pi$  to  $+\pi$ .

For single precision data:

function flavor `ippsAtan2_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsAtan2_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsAtan2_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsAtan2_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsAtan2_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsAtan2_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \text{atan2}(pSrc1[n], pSrc2[n]), 0 \leq n < len.$$

Example 12-28 below shows how to use the function `ippsAtan2`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc1</code> , <code>pSrc2</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

**Example 12-28. Using ippsAtan2 Function**

```

IppStatus ippsAtan2_32f_A21_sample(void)
{
    const Ipp32f x1[4] = {1.492, 1.700, 1.147, 1.142};
    const Ipp32f x2[4] = {1.064, 1.505, 1.950, 1.905};
    Ipp32f        y[4];
    IppStatus st = ippsAtan2_32f_A21( x1, x2, y, 4 );
    printf(" ippsAtan2_32f_A21:\n");
    printf(" x1 = %.3f %.3f %.3f %.3f \n", x1[0], x1[1], x1[2], x1[3]);
    printf(" x2 = %.3f %.3f %.3f %.3f \n", x2[0], x2[1], x2[2], x2[3]);
    printf(" y  = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsAtan2_32f_A21:
x1 = 1.492 1.700 1.147 1.142
x2 = 1.064 1.505 1.950 1.905
y  = 0.951 0.846 0.532 0.540

```

## Hyperbolic Functions

### Cosh

*Computes hyperbolic cosine of each vector element.*

---

**Syntax**

```

IppStatus ippsCosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);

```

```

IppStatus ippsCosh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCosh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCosh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCosh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsCosh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCosh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsCosh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsCosh` is declared in the `ippvm.h` file. This function computes the hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsCosh_32f_A11` and `ippsCosh_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsCosh_32f_A21` and `ippsCosh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsCosh_32f_A24` and `ippsCosh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsCosh_64f_A26` and `ippsCosh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsCosh_64f_A50` and `ippsCosh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;



function flavors `ippsCosh_64f_A53` and `ippsCosh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \cosh(pSrc[n]), 0 \leq n < len.$$

Example 12-29 below shows how to use the function `ippsCosh`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppsStsOverflow</code>	In real functions, indicates a warning that the function overflows, that is, at least one of elements of <code>pSrc</code> has the absolute value greater than $\text{Ln}(\text{FPMAX}) + \text{Ln}(2)$ , where <code>FPMAX</code> is the maximum representable floating-point number.

### Example 12-29. Using `ippsCosh` Function

```

IppStatus ippsCosh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-4.676, -4.054, 6.803, -9.525};
    Ipp32f        y[4];
    IppStatus st = ippsCosh_32f_A21( x, y, 4 );
    printf(" ippsCosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsCosh_32f_A21:
x = -4.676 -4.054 6.803 -9.525
y = 53.661 28.833 450.219 6849.870

```

## Sinh

Computes hyperbolic sine of each vector element.

### Syntax

```
IppStatus ippsSinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsSinh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsSinh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSinh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSinh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsSinh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsSinh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsSinh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsSinh` is declared in the `ippvm.h` file. This function computes the hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsSinh_32f_A11` and `ippsSinh_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsSinh_32f_A21` and `ippsSinh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsSinh_32f_A24` and `ippsSinh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsSinh_64f_A26` and `ippsSinh_64fc_A26` guarantee 26 correctly rounded bits of significand, or  $6.7E+7$  ulps, or approximately 8 exact decimal digits;

function flavors `ippsSinh_64f_A50` and `ippsSinh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsSinh_64f_A53` and `ippsSinh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \sinh(pSrc[n]), 0 \leq n < len.$

Example 12-30 below shows how to use the function `ippsSinh`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsOverflow</code>	In real functions, indicates a warning that the function overflows, that is, at least one of elements of <code>pSrc</code> has the absolute value greater than $\text{Ln}(\text{FPMAX}) + \text{Ln}(2)$ , where <code>FPMAX</code> is the maximum representable floating-point number.

## Example 12-30. Using ippsSinh Function

```

IppStatus ippsSinh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-2.483, -8.148, 3.544, -8.876};
    Ipp32f      y[4];
    IppStatus st = ippsSinh_32f_A21( x, y, 4 );
    printf(" ippsSinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsSinh_32f_A21:
x = -2.483 -8.148 3.544 -8.876
y = -5.945 -1727.412 17.290 -3577.970

```

## Tanh

*Computes hyperbolic tangent of each vector element.*

---

### Syntax

```

IppStatus ippsTanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTanh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsTanh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTanh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);

```

```

IppStatus ippsTanh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsTanh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsTanh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsTanh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsTanh` is declared in the `ippvm.h` file. This function computes the hyperbolic tangent of each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsTanh_32f_A11` and `ippsTanh_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsTanh_32f_A21` and `ippsTanh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsTanh_32f_A24` and `ippsTanh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsTanh_64f_A26` and `ippsTanh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsTanh_64f_A50` and `ippsTanh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsTanh_64f_A53` and `ippsTanh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \tanh(pSrc[n]), 0 \leq n < len.$$

Example 12-31 below shows how to use the function `ippsTanh`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-31. Using `ippsTanh` Function

```

IppStatus ippsTanh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
    Ipp32f        y[4];

    IppStatus st = ippsTanh_32f_A21( x, y, 4 );

    printf(" ippsTanh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);

    return st;
}

```

Output results:

```

ippsTanh_32f_A21:
x = -0.982 0.838 -0.448 -0.454
y = -0.754 0.685 -0.420 -0.425

```

## Acosh

*Computes inverse hyperbolic cosine of each vector element.*

---

### Syntax

```

IppStatus ippsAcosh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcosh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcosh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAcosh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

```

IppStatus ippsAcosh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcosh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAcosh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);
IppStatus ippsAcosh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);
IppStatus ippsAcosh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);
IppStatus ippsAcosh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);
IppStatus ippsAcosh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);
IppStatus ippsAcosh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsAcosh` is declared in the `ippvm.h` file. This function computes the inverse (nonnegative) hyperbolic cosine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAcosh_32f_A11` and `ippsAcosh_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAcosh_32f_A21` and `ippsAcosh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAcosh_32f_A24` and `ippsAcosh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAcosh_64f_A26` and `ippsAcosh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAcosh_64f_A50` and `ippsAcosh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAcosh_64f_A53` and `ippsAcosh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \text{acosh}(pSrc[n]), 0 \leq n < len.$$

Example 12-32 below shows how to use the function `ippsAcosh`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> is less than 1.



**Example 12-32. Using ippsAcosh Function**

```

IppStatus ippsAcosh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {588.321, 691.492, 837.773, 726.767};
    Ipp32f      y[4];
    IppStatus st = ippsAcosh_32f_A21( x, y, 4 );
    printf(" ippsAcosh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsAcosh_32f_A21:
x = 588.321 691.492 837.773 726.767
y = 7.070 7.232 7.424 7.282

```

**Asinh**

*Computes inverse hyperbolic sine of each vector element.*

---

**Syntax**

```

IppStatus ippsAsinh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsinh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsinh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAsinh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsinh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsinh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAsinh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);

```

```

IppStatus ippsAsinh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);

IppStatus ippsAsinh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s
len);

IppStatus ippsAsinh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

IppStatus ippsAsinh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

IppStatus ippsAsinh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s
len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsAsinh` is declared in the `ippvm.h` file. This function computes the inverse hyperbolic sine of each element of *pSrc*, and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavors `ippsAsinh_32f_A11` and `ippsAsinh_32cf_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAsinh_32f_A21` and `ippsAsinh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAsinh_32f_A24` and `ippsAsinh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAsinh_64f_A26` and `ippsAsinh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAsinh_64f_A50` and `ippsAsinh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAsinh_64f_A53` and `ippsAsinh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$$pDst[n] = \operatorname{asinh}(pSrc[n]), 0 \leq n < len.$$

Example 12-33 below shows how to use the function `ippsAsinh`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

### Example 12-33. Using `ippsAsinh` Function

```
IppStatus ippsAsinh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-30.122, -589.282, 487.472, -63.082};
    Ipp32f        y[4];
    IppStatus st = ippsAsinh_32f_A21( x, y, 4 );
    printf(" ippsAsinh_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsAsinh_32f_A21:
x = -30.122 -589.282 487.472 -63.082
y = -4.099 -7.072 6.882 -4.838
```

## Atanh

*Computes inverse hyperbolic tangent of each vector element.*

---

### Syntax

```
IppStatus ippsAtanh_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtanh_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtanh_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsAtanh_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtanh_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtanh_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsAtanh_32fc_A11 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtanh_32fc_A21 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtanh_32fc_A24 (const Ipp32fc* pSrc, Ipp32fc* pDst, Ipp32s len);
IppStatus ippsAtanh_64fc_A26 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAtanh_64fc_A50 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
IppStatus ippsAtanh_64fc_A53 (const Ipp64fc* pSrc, Ipp64fc* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsAtanh` is declared in the `ippvm.h` file. This function computes the inverse hyperbolic tangent of each element of `pSrc`, and stores the result in the corresponding element of `pDst`.

For single precision data:

function flavors `ippsAtanh_32f_A11` and `ippsAtanh_32fc_A11` guarantee 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavors `ippsAtanh_32f_A21` and `ippsAtanh_32fc_A21` guarantee 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavors `ippsAtanh_32f_A24` and `ippsAtanh_32fc_A24` guarantee 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavors `ippsAtanh_64f_A26` and `ippsAtanh_64fc_A26` guarantee 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavors `ippsAtanh_64f_A50` and `ippsAtanh_64fc_A50` guarantee 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavors `ippsAtanh_64f_A53` and `ippsAtanh_64fc_A53` guarantee 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

```
pDst[n] = atanh(pSrc[n]), 0 ≤ n < len.
```

Example 12-34 below shows how to use the function `ippsAtanh`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.
<code>IppStsDomain</code>	In real functions, indicates a warning that the argument is out of the function domain, that is, at least one of the elements of <code>pSrc</code> has absolute value greater than 1.
<code>IppStsSingularity</code>	In real functions, indicates a warning that the argument is the singularity point, that is, at least one of the elements of <code>pSrc</code> has absolute value equal to 1.

## Example 12-34. Using ippsAtanh Function

```
IppStatus ippsAtanh_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.076, 0.808, 0.440, -0.705};
    Ipp32f        y[4];
    IppStatus st = ippsAtanh_32f_A21( x, y, 4 );
    printf(" ippsAtanh_32f_A21:\n");

    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsAtanh_32f_A21:
x = -0.076 0.808 0.440 -0.705
y = -0.076 1.123 0.472 -0.877
```

## Special Functions

### Erf

*Computes the error function value.*

---

#### Syntax

```
IppStatus ippsErf_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErf_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErf_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErf_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErf_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErf_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsErf` is declared in the `ippvm.h` file. This function computes the error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErf_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErf_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErf_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErf_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsErf_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErf_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{erf}(pSrc[n]), 0 \leq n < len$ , where

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Example 12-35 below shows how to use the function `ippsErf`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-35. Using `ippsErf` Function

```

IppStatus ippsErf_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
    Ipp32f        y[4];

    IppStatus st = ippsErf_32f_A21( x, y, 4 );
    printf(" ippsErf_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsErf_32f_A21:
x = -0.982 0.838 -0.448 -0.454
y = -0.835 0.764 -0.474 -0.479

```

## Erfc

Computes the complementary error function value.

### Syntax

```

IppStatus ippsErfc_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfc_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfc_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfc_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```



```
IppStatus ippsErfc_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfc_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsErfc` is declared in the `ippvm.h` file. This function computes the complementary error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErfc_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErfc_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErfc_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErfc_64f_A26` guarantees 26 correctly rounded bits of significand, or  $6.7E+7$  ulps, or approximately 8 exact decimal digits;

function flavor `ippsErfc_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErfc_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{erfc}(pSrc[n]), 0 \leq n < len$ , where

$$\operatorname{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Example 12-36 below shows how to use the function `ippSErfc`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsUnderflow</code>	Indicates a warning that the function underflows, that is, at least one element of <i>pSrc</i> is less than some threshold value, where the function result is less than the minimum positive floating-point value in target precision.

**Example 12-36. Using ippsErfc Function**

```

IppStatus ippsErfc_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.982, 0.838, -0.448, -0.454};
    Ipp32f        y[4];
    IppStatus st = ippsErfc_32f_A21( x, y, 4 );
    printf(" ippsErfc_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsErfc_32f_A21:
x = -0.982 0.838 -0.448 -0.454
y = -0.754 0.685 -0.420 -0.425

```

**CdfNorm**

*Computes the cumulative normal distribution function values of vector element.*

---

**Syntax**

```

IppStatus ippsCdfNorm_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsCdfNorm_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

**Parameters**

*pSrc*                                      Pointer to the source vector.

*pDst* Pointer to the destination vector.  
*len* Number of elements in the vectors.

## Description

The function `ippsCdfNorm` is declared in the `ippvm.h` file. This function computes the cumulative normal distribution function values of *pSrc* element and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsCdfNorm_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCdfNorm_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCdfNorm_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCdfNorm_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsCdfNorm_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCdfNorm_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = CdfNorm(pSrc[n]), 0 \leq n < len$ , where

$$CdfNorm(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt .$$

Example 12-37 below shows how to use the function `ippsCdfNorm`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>IppStsUnderflow</code>	Indicates a warning that the function underflows, that is, at least one element of <i>pSrc</i> is less than some threshold value, where the function result is less than the minimum positive floating-point value in the target precision.

## Example 12-37. Using `ippsCdfNorm` Function

```
IppStatus ippsCdfNorm_32f_A24_sample(void)
{
    const Ipp32f x[4] = {+4.885, -0.543, -3.809, -4.953};
    Ipp32f        y[4];
    IppStatus st = ippsCdfNorm_32f_A24( x, y, 4 );
    printf(" ippsCdfNorm_32f_A24:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsCdfNorm_32f_A24:
x = +4.885 -0.543 -3.809 -4.953
y = +1.000 +0.294 +0.000 +0.000
```

## ErfInv

*Computes the inverse error function value.*

### Syntax

```
IppStatus ippsErfInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
```

```

IppStatus ippsErfInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfInv_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsErfInv` is declared in the `ippvm.h` file. This function computes the inverse error function value for each element of *pSrc* and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErfInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErfInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErfInv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErfInv_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsErfInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErfInv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{erfinv}(pSrc[n]), 0 \leq n < len$ , where  $\text{erfinv}(x) = \text{erf}^{-1}(x)$ , and  $\text{erf}(x)$  denotes the error function defined as given by:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Example 12-38 below shows how to use the function `ippSErfcInv`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of <i>pSrc</i> elements has the absolute value greater than 1.
<code>ippStsSingularity</code>	Indicates a warning that the argument is a singularity point, that is, at least one of the elements of <i>pSrc</i> has the absolute value equal to 1.

## Example 12-38. Using ippsErfInv Function

```

IppStatus ippsErfInv_32f_A21_sample(void)
{
    const Ipp32f x[4] = {-0.842, 0.638, -0.345, -0.774};
    Ipp32f      y[4];
    IppStatus st = ippsErfInv_32f_A21( x, y, 4 );
    printf(" ippsErfInv_32f_A21:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsErfInv_32f_A21:
x = -0.842 0.638 -0.345 -0.774
y = -0.998 0.645 -0.316 -0.856

```

## ErfcInv

*Computes the inverse complementary error function value of vector element.*

---

### Syntax

```

IppStatus ippsErfcInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfcInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfcInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsErfcInv_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfcInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
IppStatus ippsErfcInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

### Parameters

*pSrc*                                      Pointer to the source vector.



*pDst*                      Pointer to the destination vector.  
*len*                        Number of elements in the vectors.

## Description

The function `ippsErfcInv` is declared in the `ippvm.h` file. This function computes the inverse complementary error function value of each *pSrc* vector element and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsErfcInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsErfcInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsErfcInv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsErfcInv_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsErfcInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsErfcInv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{erfcinv}(pSrc[n]), 0 \leq n < len$ , where  $\text{erfcinv}(x) = \text{erfinv}(1 - x)$ , and  $\text{erfinv}(x)$  denotes the error function defined as given by:

$$\text{erfinv}(x) = \text{erf}^{-1}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt,$$

Example 12-39 below shows how to use the function `ippsErfcInv`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of <i>pSrc</i> elements is outside the function domain [0; 2].
<code>ippStsSingularity</code>	Indicates a warning that the argument is a singularity point, that is, at least one of the elements of <i>pSrc</i> is equal to 0 or 2.

**Example 12-39. Using ippsErfcInv Function**

```

IppStatus ippsErfcInv_32f_A24_sample(void)
{
    const Ipp32f x[4] = {+0.885, +0.543, +1.809, +0.953};
    Ipp32f      y[4];
    IppStatus st = ippsErfcInv_32f_A24( x, y, 4 );
    printf(" ippsErfcInv_32f_A24:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsErfcInv_32f_A24:
x = +0.885 +0.543 +1.809 +0.953
y = +0.102 +0.430 -0.925 +0.042

```

**CdfNormInv**

*Computes the inverse cumulative normal distribution function values of vector elements.*

---

**Syntax**

```

IppStatus ippsCdfNormInv_32f_A11 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s
len);

IppStatus ippsCdfNormInv_32f_A21 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s
len);

IppStatus ippsCdfNormInv_32f_A24 (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s
len);

IppStatus ippsCdfNormInv_64f_A26 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s
len);

IppStatus ippsCdfNormInv_64f_A50 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s
len);

```

```
IppStatus ippsCdfNormInv_64f_A53 (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsCdfNormInv` is declared in the `ippvm.h` file. This function computes the inverse cumulative normal distribution function values of *pSrc* vector elements and stores the result in the corresponding element of *pDst*.

For single precision data:

function flavor `ippsCdfNormInv_32f_A11` guarantees 11 correctly rounded bits of significand, or at least 3 exact decimal digits;

function flavor `ippsCdfNormInv_32f_A21` guarantees 21 correctly rounded bits of significand, or 4 ulps, or about 6 exact decimal digits;

function flavor `ippsCdfNormInv_32f_A24` guarantees 24 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

For double precision data:

function flavor `ippsCdfNormInv_64f_A26` guarantees 26 correctly rounded bits of significand, or 6.7E+7 ulps, or approximately 8 exact decimal digits;

function flavor `ippsCdfNormInv_64f_A50` guarantees 50 correctly rounded bits of significand, or 4 ulps, or approximately 15 exact decimal digits;

function flavor `ippsCdfNormInv_64f_A53` guarantees 53 correctly rounded bits of significand, including the implied bit, with the maximum guaranteed error within 1 ulp.

The computation is performed as follows:

$pDst[n] = \text{CdfNormInv}(pSrc[n]), 0 \leq n < len$ , where  $\text{CdfNormInv}(x) = \text{CdfNorm}^{-1}(x)$ , and  $\text{CdfNorm}(x)$  denotes the cumulative normal distribution function:

$$\text{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt .$$

Example 12-40 below shows how to use the function `ippSCdfNormInv`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.
<code>ippStsDomain</code>	Indicates a warning that the argument is out of the function domain, that is, at least one of <i>pSrc</i> elements is outside the function domain <code>[0; 1]</code> .
<code>ippStsSingularity</code>	Indicates a warning that the argument is a singularity point, that is, at least one of the elements of <i>pSrc</i> is equal to 0 or 1.

## Example 12-40. Using ippsCdfNormInv Function

```

IppStatus ippsCdfNormInv_32f_A24_sample(void)
{
    const Ipp32f x[4] = {+0.085, +0.543, +1.809, +0.953};
    Ipp32f      y[4];
    IppStatus st = ippsCdfNormInv_32f_A24( x, y, 4 );
    printf(" ippsCdfNormInv_32f_A24:\n");
    printf(" x = %+.3f %+.3f %+.3f %+.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %+.3f %+.3f %+.3f %+.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

```

ippsCdfNormInv_32f_A24:
x = +0.085 +0.543 +1.809 +0.953
y = -1.372 +0.108 +0.874 +1.675

```

## Rounding Functions

### Floor

*Computes integer value rounded toward minus infinity for each vector element.*

---

#### Syntax

```

IppStatus ippsFloor_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsFloor_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

#### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.

*len*                      Number of elements in the vectors.

### Description

The function `ippsFloor` is declared in the `ippvm.h` file. This function computes an integer value rounded towards minus infinity for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Example 12-41 below shows how to use the function `ippsFloor`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

### Example 12-41. Using `ippsFloor` Function

```
IppStatus ippsFloor_32f_sample(void)
{
    const Ipp32f x[4] = {-0.883, -0.265, 0.176, 0.752};
    Ipp32f y[4];
    IppStatus st = ippsFloor_32f ( x, y, 4 );
    printf(" ippsFloor_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

`ippsFloor_32f:`

`x = -0.883 -0.265 0.176 0.752`

`y = -1.000 -1.000 0.000 0.000`

## Ceil

*Computes integer value rounded toward plus infinity for each vector element.*

---

### Syntax

```
IppStatus ippsCeil_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsCeil_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsCeil` is declared in the `ippvm.h` file. This function computes an integer value rounded towards plus infinity for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Example 12-42 below shows how to use the function `ippsCeil`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.



**Example 12-42. Using ippsCeil Function**

```

IppStatus ippsCeil_32f_sample(void)
{
    const Ipp32f x[4] = {-0.883, -0.265, 0.176, 0.752};
    Ipp32f y[4];
    IppStatus st = ippsCeil_32f ( x, y, 4 );
    printf(" ippsCeil_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}

```

Output results:

ippsCeil\_32f:

x = -0.883 -0.265 0.176 0.752

y = 0.000 0.000 1.000 1.000

**Trunc**

*Computes integer value rounded toward zero for each vector element.*

---

**Syntax**

```

IppStatus ippsTrunc_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsTrunc_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);

```

**Parameters**

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsTrunc` is declared in the `ippvm.h` file. This function computes an integer value rounded towards zero for each element of the vector `pSrc`, and stores the result in the corresponding element of the vector `pDst`.

Example 12-43 below shows how to use the function `ippsTrunc`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <code>pSrc</code> or <code>pDst</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <code>len</code> is less than or equal to 0.

## Example 12-43. Using `ippsTrunc` Function

```
IppStatus ippsTrunc_32f_sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y[4];
    IppStatus st = ippsTrunc_32f ( x, y, 4 );
    printf(" ippsTrunc_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

```
ippsTrunc_32f:
x = -1.883 -0.265 0.176 1.752
y = -1.000 0.000 0.000 1.000
```

## Round

*Computes integer value rounded to nearest for each vector element.*

---

### Syntax

```
IppStatus ippsRound_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);  
IppStatus ippsRound_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsRound` is declared in the `ippvm.h` file. This function computes a rounded to the nearest integer value for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*. Halfway values, that is, 0.5, -1.5, and the like, are rounded off away from zero, that is, 0.5 -> 1, -1.5 -> -2, and so on.

Example 12-44 below shows how to use the function `ippsRound`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-44. Using `ippsRound` Function

```
IppStatus ippsRound_32f_sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y[4];
    IppStatus st = ippsRound_32f ( x, y, 4 );
    printf(" ippsRound_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y = %.3f %.3f %.3f %.3f \n", y[0], y[1], y[2], y[3]);
    return st;
}
```

Output results:

`ippsRound_32f:`

`x = -1.883 -0.265 0.176 1.752`

`y = -2.000 0.000 0.000 2.000`

## NearbyInt

*Computes rounded integer value in current rounding mode for each vector element.*

---

### Syntax

```
IppStatus ippsNearbyInt_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsNearbyInt_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

### Description

The function `ippsNearbyInt` is declared in the `ippvm.h` file. This function computes a rounded integer value in a current rounding mode for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst*.

Example 12-45 below shows how to use the `ippsNearbyInt` function.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-45. Using ippsNearbyInt Function

```
#include <fenv.h>

void ippsNearbyInt_32f_sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4], y2[4];
    fesetround(FE_TONEAREST);
    ippsNearbyInt_32f ( x, y2, 4 );
    fesetround(FE_TOWARDZERO);
    ippsNearbyInt_32f ( x, y2, 4 );
    printf(" ippsNearbyInt_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
}
```

Output results:

```
ippsNearInt_32f:
x = -1.883 -0.265 0.176 1.752
y = -2.000 0.000 0.000 2.000
y = -1.000 0.000 0.000 1.000
```

## Rint

*Computes rounded integer value in current rounding mode for each vector element with inexact result exception raised for each changed value.*

---

### Syntax

```
IppStatus ippsRint_32f (const Ipp32f* pSrc, Ipp32f* pDst, Ipp32s len);
IppStatus ippsRint_64f (const Ipp64f* pSrc, Ipp64f* pDst, Ipp32s len);
```

## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsRint` is declared in the `ippvm.h` file. This function computes a rounded integer value in a current rounding mode for each element of the vector *pSrc*, and stores the result in the corresponding element of the vector *pDst* raising inexact result exception if the value has changed.

Example 12-46 below shows how to use the `ippsRint` function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-46. Using ippsRint Function

```
#include <fenv.h>

void ippsRint_32f_sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4], y2[4];
    fesetround(FE_TONEAREST);
    ippsRint_32f ( x, y2, 4 );
    fesetround(FE_TOWARDZERO);
    ippsRint_32f ( x, y2, 4 );
    printf(" ippsRint_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
}
```

Output results:

```
ippsRint_32f:
x = -1.883 -0.265 0.176 1.752
y = -2.000 0.000 0.000 2.000
y = -1.000 0.000 0.000 1.000
```

## Modf

*Computes truncated integer value and remaining fraction part for each vector element.*

---

### Syntax

```
IppStatus ippsModf_32f (const Ipp32f* pSrc, Ipp32f* pDst1, Ipp32f* pDst2,
Ipp32s len);

IppStatus ippsModf_64f (const Ipp64f* pSrc, Ipp64f* pDst1, Ipp64f* pDst2,
Ipp32s len);
```



## Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst1</i>	Pointer to the first destination vector.
<i>pDst2</i>	Pointer to the second destination vector.
<i>len</i>	Number of elements in the vectors.

## Description

The function `ippsModf` is declared in the `ippvm.h` file. This function computes a truncated value and a remainder of each element of the vector *pSrc*. The truncated integer value is stored in the corresponding element of the *pDst1* vector and the remainder is stored in the corresponding element of the *pDst2* vector.

Example 12-47 below shows how to use the function `ippsModf`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error when <i>pSrc</i> or <i>pDst1</i> or <i>pDst2</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error when <i>len</i> is less than or equal to 0.

## Example 12-47. Using `ippsModf_32f` Function

```
IppStatus ippsModf_32f_sample(void)
{
    const Ipp32f x[4] = {-1.883, -0.265, 0.176, 1.752};
    Ipp32f y1[4] y2[4];
    IppStatus st = ippsModf_32f ( x, y1, y2, 4 );
    printf(" ippsModf_32f:\n");
    printf(" x = %.3f %.3f %.3f %.3f \n", x[0], x[1], x[2], x[3]);
    printf(" y1 = %.3f %.3f %.3f %.3f \n", y1[0], y1[1], y1[2], y1[3]);
    printf(" y2 = %.3f %.3f %.3f %.3f \n", y2[0], y2[1], y2[2], y2[3]);
    return st;
}
```

Output results:

`ippsModf_32f:`

`x = -1.883 -0.265 0.176 1.752`

`y1 = -1.000 0.000 0.000 1.000`

`y2 = -0.883 -0.265 0.176 0.752`

# Data Compression Functions

This chapter describes the Intel® IPP functions for data compression that support a number of different compression methods: Huffman and variable-length coding, dictionary-based coding methods (including support of ZLIB compression), and methods based on Burrows-Wheeler Transform.

These functions are grouped into the following sections:

[VLC and Huffman Coding Functions](#)

[Dictionary-Based Compression Functions](#)

[BWT-Based Compression Functions](#)

The full lists of this functions with the brief description of operations that these functions perform are given in the tables below. The functions are described in more detail later in the corresponding sections.

**Table 13-1. Intel IPP VLC and Huffman Coding Functions**

Function Short Name	Description
VLC Functions	
<a href="#">VLC Encode Init Alloc</a>	Allocates memory and initializes the VLC encoder structure.
<a href="#">VLC Encode Free</a>	Frees memory allocated for the VLC encoder structure.
<a href="#">VLC Encode Init</a>	Initializes the VLC encoder structure.
<a href="#">VLC Encode Get Size</a>	Computes the size of the VLC encode structure.
<a href="#">VLC Encode Block</a>	Performs VLC encoding of a block of data.
<a href="#">VLC Encode One</a>	Performs VLC encoding of a single element.
<a href="#">VLC Count Bits</a>	Computes a number of bits required to encode the source block.
<a href="#">VLC Decode Init Alloc</a>	Allocates memory and initializes the VLC decoder structure.
<a href="#">VLC Decode Free</a>	Frees memory allocated for the VLC decoder structure.
<a href="#">VLC Decode Init</a>	Initializes the VLC decoder structure.
<a href="#">VLC Decode Get Size</a>	Computes the size of the VLC decoder structure.
<a href="#">VLC Decode Block</a>	Decodes a block of VLC encoded data.
<a href="#">VLC Decode One</a>	Decodes a single VLC encoded element.
<a href="#">VLC Decode UTuple Init Alloc</a>	Allocates memory and initializes the VLCDecodeUTuple structure based on the input parameters.
<a href="#">VLC Decode UTuple Free</a>	Frees memory allocated for the VLCDecodeUTuple structure.
<a href="#">VLC Decode UTuple Init</a>	Initializes the VLCDecodeUTuple structure based on the input parameters.

Function Short Name	Description
<a href="#">VLCDecodeUTupleGetSize</a>	Computes the size of the VLCDecodUTuple structure.
<a href="#">VLCDecodeUTupleBlock</a>	Decodes a block of VLC encoded data.
<a href="#">VLCDecodeUTupleOne</a>	Decodes a single VLC encoded tuple.
Huffman Encoding and Decoding Functions	
<a href="#">EncodeHuffInitAlloc</a>	Allocates memory and initializes the Huffman encoding structure.
<a href="#">HuffFree</a>	Frees memory allocated for the Huffman encoding and decoding structures.
<a href="#">EncodeHuffInit</a>	Initializes the Huffman encoding structure.
<a href="#">HuffGetSize</a>	Computes size of the external buffer for the Huffman encoding structure.
<a href="#">EncodeHuffOne</a>	Performs Huffman encoding of a single element.
<a href="#">EncodeHuff</a>	Performs Huffman encoding.
<a href="#">EncodeHuffFinal</a>	Performs Huffman encoding of the remainder.
<a href="#">HuffGetLenCodeTable</a>	Returns the table containing lengths of Huffman codes.
<a href="#">DecodeHuffInitAlloc</a>	Allocates memory and initializes the Huffman decoding structure.
<a href="#">DecodeHuffInit</a>	Initializes the Huffman decoding structure.
<a href="#">DecodeHuffOne</a>	Performs Huffman decoding of a single code word.
<a href="#">DecodeHuff</a>	Performs Huffman decoding.
<a href="#">HuffGetDstBuffSize</a>	Computes the size of a destination buffer for Huffman encoding and decoding by blocks
<a href="#">HuffLenCodeTablePack</a>	Packs the table containing lengths of Huffman codes.
<a href="#">HuffLenCodeTableUnpack</a>	Unpacks the table containing lengths of Huffman codes.

**Table 13-2. Intel IPP Dictionary-Based Data Compression Functions**

Function Short Name	Description
LZSS Coding	
<a href="#">EncodeLZSSInitAlloc</a>	Allocates memory and initializes the LZSS state structure.
<a href="#">LZSSFree</a>	Frees memory allocated for the LZSS state structure.
<a href="#">EncodeLZSSInit</a>	Initializes the LZSS state structure.
<a href="#">LZSSGetSize</a>	Computes the size of the LZSS state structure.
<a href="#">EncodeLZSS</a>	Performs LZSS encoding.
<a href="#">EncodeLZSSFlush</a>	Flushes the last few bits in the bitstream and aligns the output data on a byte boundary.
<a href="#">DecodeLZSSInitAlloc</a>	Allocates memory and initializes the LZSS state structure.
<a href="#">DecodeLZSSInit</a>	Initializes the LZSS state structure.

Function Short Name	Description
<a href="#">DecodeLZSS</a>	Performs LZSS decoding
ZLIB Coding	
<a href="#">EncodeLZ77Init</a>	Initializes the encoding LZ77 state structure.
<a href="#">EncodeLZ77GetSize</a>	Computes the size of the encoding LZ77 state structure.
<a href="#">EncodeLZ77InitAlloc</a>	Allocates memory and initializes the encoding LZ77 state structure.
<a href="#">LZ77Free</a>	Frees memory allocated for the encoding or decoding LZ77 state structures.
<a href="#">EncodeLZ77</a>	Performs LZ77 encoding.
<a href="#">EncodeLZ77SelectHuffMode</a>	Determines the optimal encoding mode.
<a href="#">EncodeLZ77FixedHuff</a>	Performs encoding with the fixed Huffman codes.
<a href="#">EncodeLZ77DynamicHuff</a>	Performs encoding with the dynamic Huffman codes.
<a href="#">EncodeLZ77StoredBlock</a>	Stores the data block without compression.
<a href="#">EncodeLZ77Flush</a>	Writes the checksum and total length of the input data to the end of the output stream.
<a href="#">EncodeLZ77GetPairs</a>	Retrieves pair data from the encoding state structure.
<a href="#">EncodeLZ77SetPairs</a>	Sets pair data to the encoding state structure.
<a href="#">EncodeLZ77GetStatus</a>	Reads the deflate status value from the encoding state structure.
<a href="#">EncodeLZ77SetStatus</a>	Writes the deflate status value to the encoding state structure.
<a href="#">EncodeLZ77Reset</a>	Resets the internal encoding state structure to the initial state.
<a href="#">DecodeLZ77Init</a>	Initializes the LZ77 decoding structure.
<a href="#">DecodeLZ77GetSize</a>	Computes the size of the decoding LZ77 state structure.
<a href="#">DecodeLZ77InitAlloc</a>	Allocates memory and initializes the decoding LZ77 state structure.
<a href="#">DecodeLZ77</a>	Performs LZ77 decoding.
<a href="#">DecodeLZ77GetBlockType</a>	Determines the type of encoded data block.
<a href="#">DecodeLZ77FixedHuff</a>	Performs fixed Huffman decoding.
<a href="#">DecodeLZ77DynamicHuff</a>	Performs dynamic Huffman decoding.
<a href="#">DecodeLZ77StoredBlock</a>	Performs decoding of the block stored without compression.
<a href="#">DecodeLZ77GetPairs</a>	Retrieves pair data from the decoding state structure.
<a href="#">DecodeLZ77SetPairs</a>	Sets pair data to the decoding state structure.
<a href="#">DecodeLZ77GetStatus</a>	Reads the inflate status value from the LZ77 decoding state structure.

Function Short Name	Description
<a href="#">DecodeLZ77SetStatus</a>	Sets the inflate status to the desired value in the LZ77 decoding state structure
<a href="#">DecodeLZ77Reset</a>	Resets the internal decoding state structure to the initial state.
<a href="#">Adler32</a>	Computes the Adler32 checksum for the source data buffer.
<a href="#">CRC32, CRC32C</a>	Computes the CRC32 checksum for the source data buffer.
<a href="#">DeflateLZ77</a>	Performs LZ77 encoding according to the specified compression level.
<a href="#">DeflateDictionarySet</a>	Presets the user's dictionary for LZ77 encoding.
<a href="#">DeflateHuff</a>	Performs Huffman encoding.
<a href="#">InflateBuildHuffTable</a>	Builds the Huffman code table for the data encoded in the "deflate" format.
<a href="#">Inflate</a>	Decodes the data encoded in the "deflate" format.
<a href="#">LZO Coding</a>	
<a href="#">EncodeLZOGetSize</a>	Calculates the size of LZO encoding structure.
<a href="#">EncodeLZOInit</a>	Initializes LZO encoding structure.
<a href="#">EncodeLZO</a>	Compresses input data, returns the length of the compressed data.
<a href="#">DecodeLZO</a>	Decompresses input data, returns the length of the decompressed data.
<a href="#">DecodeLZOSafe</a>	Decompresses input data with constantly checking integrity of output.

**Table 13-3. Intel IPP BWT-Based Compression Functions**

Function Short Name	Description
BWT Functions	
<a href="#">BWTFwdGetSize</a>	Computes the size of the external buffer for the forward BWT transform.
<a href="#">BWTFwd</a>	Performs the forward BWT transform.
<a href="#">BWTInvGetSize</a>	Computes the size of the external buffer for the inverse BWT transform.
<a href="#">BWTInv</a>	Performs the inverse BWT transform.
<a href="#">BWTGetSize_SmalBlock</a>	Computes the size of the external buffer for the BWT transforms for small data block.
<a href="#">BWTFwd_SmalBlock</a>	Performs the forward BWT transform for small data block.
<a href="#">BWTInv_SmalBlock</a>	Performs the inverse BWT transform for small data block.
GIT Functions	

Function Short Name	Description
<a href="#">EncodeGITInitAlloc</a>	Allocates memory and initializes the GIT encoding state structure.
<a href="#">GITFree</a>	Frees memory allocated by the function <a href="#">EncodeGITInitAlloc</a> or <a href="#">DecodeGITInitAlloc</a> .
<a href="#">EncodeGITInit</a>	Initializes the GIT encoding state structure.
<a href="#">EncodeGITGetSize</a>	Computes the size of the GIT encoding state structure.
<a href="#">EncodeGIT</a>	Performs GIT encoding.
<a href="#">DecodeGITInitAlloc</a>	Allocates memory and initializes the GIT decoding state structure.
<a href="#">DecodeGITInit</a>	Initializes the GIT decoding state structure.
<a href="#">DecodeGITGetSize</a>	Computes the size of the GIT decoding state structure.
<a href="#">DecodeGIT</a>	Performs GIT decoding.
<b>MTF Functions</b>	
<a href="#">MTFInitAlloc</a>	Allocates memory and initializes the MTF structure.
<a href="#">MTFFree</a>	Frees memory allocated for the MTF structure.
<a href="#">MTFInit</a>	Initializes the MTF structure.
<a href="#">MTFGetSize</a>	Computes the size of the MTF structure.
<a href="#">MTFFwd</a>	Performs the forward MTF transform.
<a href="#">MTFInv</a>	Performs the inverse MTF transform.
<b>RLE Functions</b>	
<a href="#">EncodeRLE</a>	Performs RLE encoding.
<a href="#">DecodeRLE</a>	Performs RLE decoding.
<b>bzip2 Compatible Functions</b>	
<a href="#">EncodeRLEInitAlloc_BZ2</a>	Allocates memory and initializes the bzip2-specific RLE structure.
<a href="#">RLEFree_BZ2</a>	Frees memory allocated for the bzip2-specific RLE structure.
<a href="#">EncodeRLEInit_BZ2</a>	Initializes the bzip2-specific RLE structure.
<a href="#">RLEGetSize_BZ2</a>	Compute the size of the state structure for the bzip2-specific RLE.
<a href="#">EncodeRLE_BZ2</a>	Performs the bzip2-specific RLE.
<a href="#">EncodeRLEFlush_BZ2</a>	Flushes the remaining data after RLE.
<a href="#">RLEGetInUseTable</a>	Gets the pointer to the <code>inUse</code> vector from the RLE state structure.
<a href="#">DecodeRLE_BZ2</a>	Performs the bzip2-specific RLE.
<a href="#">EncodeZ1Z2_BZ2</a>	Performs the bzip2-specific Z1Z2 encoding.
<a href="#">DecodeZ1Z2_BZ2</a>	Performs the bzip2-specific Z1Z2 decoding.
<a href="#">ReduceDictionary</a>	Performs the dictionary reducing.

Function Short Name	Description
<a href="#">ExpandDictionary</a>	Performs the dictionary expanding.
<a href="#">CRC32_BZ2</a>	Computes the CRC32 checksum for the source data buffer.
<a href="#">EncodeHuffInitAlloc_BZ2</a>	Allocated memory an initializes the elements of the bzip2-specific internal state for Huffman encoding.
<a href="#">EncodeHuffFree_BZ2</a>	Frees memory allocated for the bzip2-specific Huffman encoding structure.
<a href="#">EncodeHuffInit_BZ2</a>	Initializes the elements of the bzip2-specific internal state for Huffman encoding.
<a href="#">EncodeHuffGetSize_BZ2</a>	Computes the size of the internal state for bzip2-specific Huffman encoding.
<a href="#">PackHuffContext_BZ2</a>	Performs the bzip2-specific encoding of Huffman context.
<a href="#">EncodeHuff_BZ2</a>	Performs the bzip2-specific Huffman encoding.
<a href="#">DecodeHuffInitAlloc_BZ2</a>	Allocated memory an initializes the elements of the bzip2-specific internal state for Huffman decoding.
<a href="#">DecodeHuffFree_BZ2</a>	Frees memory allocated for the bzip2-specific Huffman decoding structure.
<a href="#">DecodeHuffInit_BZ2</a>	Initializes the elements of the bzip2-specific internal state for Huffman decoding.
<a href="#">DecodeHuffGetSize_BZ2</a>	Computes the size of the internal state for bzip2-specific Huffman decoding.
<a href="#">UnpackHuffContext_BZ2</a>	Performs the bzip2-specific decoding of Huffman context.
<a href="#">DecodeHuff_BZ2</a>	Performs the bzip2-specific Huffman decoding.

## Application Notes

- The functions in this domain can be divided into two types: the functions that actually compress data, and transformation functions. The latters do not compress data but only modify them and prepare for further compression. The examples of such transformation are the *Burrows-Weller Transform*, or *MoveToFront* algorithm. To do data compression efficient, you should develop the proper consequence of functions of different type that will transform data and then compress them.
- Compression ratio depends on the statistics of input data. For some types of input data no compression could be achieved at all.
- The size of memory required for the output of data compression functions typically is not obvious. As a rule encoding functions uses less memory for output than the size of the input buffer, on the contrary decoding functions use more memory for output than the size of the input buffer. You should account these issues and allocate the proper quantity of output memory using the techniques provided by functions in this domain. For example, you can



use a double-pointer technique for automatic shifting the user submitted pointer. For some other functions it is possible to compute the upper limit of the size of the required output buffer.

## VLC and Huffman Coding Functions

This section describes the Intel IPP functions for Huffman and variable-length coding.

The full list of these functions is given in [Table 13-1](#).

### Data Compression VLC functions

Variable length coding (VLC) is a data compression method that uses statistical modeling to define which values occur more frequently than others to build tables for subsequent encoding and decoding operations. Data in the bitstream is encoded with VLC table so that the shortest codes correspond to the most frequent values and the longer codes correspond to the less frequent values. Intel IPP functions use the structure `IppsVLCTable` for storing the VLC table:

```
struct VLCTable_32s
{
    Ipp32s    value,
    Ipp32s    code,
    Ipp32s    length;
}
typedef struct VLCTable_32s IppsVLCTable_32s
```

Here

`value` - the current code,

`code` - the actual bit code for the `value`,

`length` - the bit length of the `value`.

The `value` can be both positive and negative. The `code` denotes the `length` least significant bits in binary representation of the code value. Thus, valid lengths are within the range of 0 to 32. Note that you are responsible for specifying correct tables with prefix codes for values specified in the `value` field of the `IppsVLCTable` structure.

## VLCEncodeInitAlloc

*Allocates memory and initializes the VLC encoder structure.*

---

### Syntax

```
ippsVLCEncodeInitAlloc_32s (const IppsVLCTable_32s* pInputTable, int
inputTableSize, IppsVLCEncodeSpec_32s** ppVLCSpec);
```

### Parameters

<i>pInputTable</i>	Pointer to the VLC table.
<i>inputTableSize</i>	Size of the VLC table.
<i>ppVLCSpec</i>	Double pointer to the VLC encoder specification structure.

### Description

The function `ippsVLCEncodeInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the VLC encoder specification structure *ppVLCSpec* that uses the VLC table *pInputTable*. The VLC encoder specification structure is used by the functions [ippsVLCEncodeBlock](#) and [ippsVLCEncodeOne](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.
<code>ippStsVLCInputDataErr</code>	Indicates an error if <i>inputTableSize</i> is less than or equal to 0.

## VLCEncodeFree

*Frees memory allocated for the VLC encoder structure.*

---

### Syntax

```
void ippsVLCEncodeFree_32s (IppsVLCEncodeSpec_32s* pVLCSpec);
```

## Parameters

*pVLCSpec* Pointer to the VLC encoder specification structure.

## Description

The function `ippsVLCDecodeFree` is declared in the `ippdc.h` file. This function frees memory allocated for the VLC encode structure by the function `ippsVLCDecodeInitAlloc`.

## VLCDecodeInit

*Initializes the VLC encoder structure.*

### Syntax

```
ippsVLCDecodeInit_32s (const IppsVLCTable_32s* pInputTable, int
inputTableSize, IppsVLCDecodeSpec_32s* pVLCSpec);
```

## Parameters

*pInputTable* Pointer to the VLC table  
*inputTableSize* Size of the VLC table  
*pVLCSpec* Pointer to the VLC encoder specification structure.

## Description

The function `ippsVLCDecodeInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the VLC encoder specification structure *pVLCSpec* that uses the VLC table *pInputTable*. The size of the external buffer should be computed previously by calling the function `ippsVLCDecodeGetSize`. Alternatively, the VLC encoder structure can be initialized by the function `ippsVLCDecodeInitAlloc`.

The VLC encoder specification structure is used by the functions `ippsVLCDecodeBlock` and `ippsVLCDecodeOne`.



**NOTE.** The parameters *pInputTable* and *inputTableSize* of the `ippsVLCDecodeInit` and `ippsVLCDecodeGetSize` functions must have equal respective values to avoid an error in subsequent encoding functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCInputDataErr</code>	Indicates an error if <i>inputTableSize</i> is less than or equal to 0.

## VLCEncodeGetSize

Computes the size of the VLC encoder structure.

### Syntax

```
ippVLCEncodeGetSize_32s (const IppsVLCTable_32s* pInputTable, int
inputTableSize, Ipp32s* pSize);
```

### Parameters

<i>pInputTable</i>	Pointer to the VLC table
<i>inputTableSize</i>	Size of the input table
<i>pSize</i>	Pointer to the size of the VLC encoder specification structure.

### Description

The function `ippVLCEncodeGetSize` is declared in the `ippdc.h` file. This function computes the size of the VLC encoder structure that uses the VLC table *pInputTable*. Its size is stored in the *pSize*. This function should be called prior to the function `ippVLCEncodeInit`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCInputDataErr</code>	Indicates an error if <i>inputTableSize</i> is less than or equal to 0.

## VLCEncodeBlock

*Performs VLC encoding of a block of data.*

---

### Syntax

```
ippsVLCEncodeBlock_16s1u (const Ipp16s* pSrc, int srcLen, Ipp8u** ppDst,
int* pDstBitsOffset, const IppsVLCEncodeSpec_32s* pVLCSpec);
```

### Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Number of elements in the <i>pSrc</i> .
<i>ppDst</i>	Double pointer to the destination bitstream.
<i>pDstBitsOffset</i>	Pointer to the input/output bit position in the <i>pDst</i> .
<i>pVLCSpec</i>	Pointer to the VLC encoder specification structure.

### Description

The function `ippsVLCEncodeBlock` is declared in the `ippdc.h` file. This function encodes *srcLen* elements of the source vector *pSrc* and stores the result in the destination buffer *pDst*. After encoding the function shifts the *pDst* pointer by the number of successfully written bytes. The function updates *pDstBitsOffset* setting it to the actual bit position in the destination buffer.

The `ippsVLCEncodeBlock` function uses the VLC table from the encoder specification structure *pVLCSpec* that must be initialized by the functions `ippsVLCEncodeInitAlloc` or `ippsVLCEncodeInit` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used.

## VLCEncodeOne

*Performs VLC encoding of a single element.*

---

### Syntax

```
ippsVLCEncodeOne_16s1u (Ipp16s src, Ipp8u** ppDst, int* pDstBitsOffset, const
IppsVLCEncodeSpec_32s* pVLCSpec);
```

### Parameters

<i>src</i>	Source value.
<i>ppDst</i>	Double pointer to the destination bitstream.
<i>pDstBitsOffset</i>	Pointer to the input/output bit position in the <i>ppDst</i> .
<i>pVLCSpec</i>	Pointer to the VLC encoder specification structure.

### Description

The function `ippsVLCEncodeOne` is declared in the `ippdc.h` file. This function encodes a single element *src* and stores the result in the destination buffer *ppDst*. After encoding the function shifts the *ppDst* pointer by the number of successfully written bytes. The function updates *pDstBitsOffset* setting it to the actual bit position in the destination buffer.

The function `ippsVLCEncodeOne` uses the VLC table from the encoder specification structure *pVLCSpec* that must be initialized by the functions `ippsVLCEncodeInitAlloc` or `ippsVLCEncodeInit` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used.

## VLCCountBits

*Computes a number of bits required to encode the source block.*

---

### Syntax

```
ippVLCCountBits_16s32s (const Ipp16s* pSrc, int srcLen, Ipp32s* pCountBits,
const IppsVLCDecodeSpec_32s* pVLCSpec);
```

### Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Number of elements in the <i>pSrc</i> .
<i>pCounBits</i>	Pointer to the computed length in bits required to encode <i>pSrc</i> .
<i>pVLCSpec</i>	Pointer to the VLC encoder specification structure.

### Description

The function `ippVLCCountBits` is declared in the `ippdc.h` file. This function computes a number of bits required to encode the source data *pSrc* using the VLC table from the encoder specification structure *pVLCSpec* that must be initialized by the functions [ippsVLCDecodeInitAlloc](#) or [ippsVLCDecodeInit](#) beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .

## VLCDecodeInitAlloc

*Allocates memory and initializes the VLC decoder structure.*

---

### Syntax

```
ippsVLCDecodeInitAlloc_32s (const IppsVLCTable_32s* pInputTable, int
inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables,
IppsVLCDecodeSpec_32s** ppVLCSpec);
```

## Parameters

<i>pInputTable</i>	Pointer to the VLC table.
<i>inputTableSize</i>	Size of the VLC table.
<i>pSubTablesSizes</i>	Pointer to the sizes of the subtables.
<i>numSubTables</i>	Number of the subtables.
<i>ppVLCSpec</i>	Double pointer to the VLC decoder specification structure.

## Description

The function `ippVLCDecodeInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the VLC decoder structure *ppVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

The VLC decoder specification structure is used by the functions `ippVLCDecodeBlock` and `ippVLCDecodeOne`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.
<code>ippStsVLCUsTblCodeLengthErr</code>	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in <i>pSubTablesSizes</i> is less than 0; or if the sum of these values is less than the maximum code length in the input table.

## VLCDecodeFree

*Frees memory allocated for the VLC decoder structure.*

---

### Syntax

```
void ippVLCDecodeFree_32s (IppsVLCDecodeSpec_32s* pVLCSpec);
```



## Parameters

*pVLCSpec* Pointer to the VLC decoder specification structure.

## Description

The function `ippsVLCDecodeFree` is declared in the `ippdc.h` file. This function frees memory allocated for the VLC decoder structure by the function `ippsVLCDecodeInitAlloc`.

## VLCDecodeInit

*Initializes the VLC decoder structure.*

---

### Syntax

```
ippsVLCDecodeInit_32s (const IppsVLCTable_32s* pInputTable, int
inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables,
IppsVLCDecodeSpec_32s* pVLCSpec);
```

### Parameters

*pInputTable* Pointer to the VLC table.  
*inputTableSize* Size of the VLC table.  
*pSubTablesSizes* Pointer to the sizes of the subtables.  
*numSubTables* Number of the subtables.  
*pVLCSpec* Pointer to the VLC decoder specification structure.

### Description

The function `ippsVLCDecodeInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the VLC decoder structure *pVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table. The size of the external buffer must be computed previously by calling the function `ippsVLCDecodeGetSize`. Alternatively, the VLC decoder structure can be initialized by the function `ippsVLCDecodeInitAlloc`.

The VLC decoder specification structure is used by the functions `ippsVLCDecodeBlock` and `ippsVLCDecodeOne`.



**NOTE.** The parameters *pInputTable* and *inputTableSize* of the `ippsVLCDecodeInit` and `ippsVLCDecodeGetSize` functions must have equal respective values to avoid an error in subsequent decoding fu

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCUsTblCodeLengthErr</code>	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in <i>pSubTablesSizes</i> is less than 1; or if the sum of these values is less than the maximum code length in the input table.

## VLCDecodeGetSize

*Computes the size of the VLC decoder structure.*

### Syntax

```
ippsVLCDecodeGetSize_32s (const IppsVLCTable_32s* pInputTable, int
inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables, Ipp32s* pSize);
```

### Parameters

<i>pInputTable</i>	Pointer to the input table.
<i>inputTableSize</i>	Size of the input table.
<i>pSubTablesSizes</i>	Pointer to the sizes of the subtables.
<i>numSubTables</i>	Number of the subtables.
<i>pVLCSpec</i>	Pointer to the size of the VLC decoder specification structure.

### Description

The function `ippsVLCDecodeGetSize` is declared in the `ippdc.h` file. This function computes the size of the VLC decoder structure *ppVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*. Size values for the subtables must be greater

than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table. This function must be called prior to the function `ippVLCDe-  
codeInit`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCUsrTblCodeLengthErr</code>	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in <code>pSubTablesSizes</code> is less than 0; or if the sum of these values is less than the maximum code length in the input table.

## VLCDecodeBlock

*Decodes a block of VLC encoded data.*

### Syntax

```
ippVLCDecodeBlock_lu16s (const Ipp8u** ppSrc, int* pSrcBitsOffset, Ipp16s*  
pDst, int dstLen, const IppsVLCDecodeSpec_32s* pVLCSpec);
```

### Parameters

<code>ppSrc</code>	Double pointer to the source bitstream.
<code>pSrcBitsOffset</code>	Pointer to the input/output bit position.
<code>pDst</code>	Pointer to the destination buffer.
<code>dstLen</code>	Number of elements in the <code>pDst</code> .
<code>pVLCSpec</code>	Pointer to the VLC decoder specification structure.

### Description

The function `ippVLCDecodeBlock` is declared in the `ippdc.h` file. This function decodes `dstLen` VLC encoded elements in the source bitstream starting from the `pSrcBitsOffset` bit position. The function stores the result in the destination buffer `pDst`. After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates `pSrcBitsOffset` setting it to the actual bit position in the source buffer.

The function `ippVLCDecodeBlock` uses the VLC table from the decoder specification structure `pVLCSpec` that must be initialized by the functions `ippVLCDecodeInitAlloc` or `ippVLCDe-  
codeInit` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used. It can indicate that bitstream contains code that is not specified in the table.
<code>ippStsBitOffsetErr</code>	Indicates an error if value of the <code>pSrcBitsOffset</code> is less than 0 or greater than 7.
<code>ippStsContextMatchErr</code>	Indicates an error if the <code>pVLCSpec</code> structure is not initialized by the function <code>ippsVLCDecodeInitAlloc</code> or <code>ippsVLCDecodeInit</code> .

## VLCDecodeOne

*Decodes a single VLC encoded element.*

---

### Syntax

```
ippsVLCDecodeOne_lu16s (const Ipp8u** ppSrc, int* pSrcBitsOffset, Ipp16s* pDst, const IppsVLCDecodeSpec_32s* pVLCSpec);
```

### Parameters

<code>ppSrc</code>	Double pointer to the source bitstream.
<code>pSrcBitsOffset</code>	Pointer to the input/output bit position.
<code>pDst</code>	Pointer to the decoded value.
<code>pVLCSpec</code>	Pointer to the VLC decoder specification structure.

### Description

The function `ippsVLCDecodeOne` is declared in the `ippdc.h` file. This function decodes a single VLC encoded element in the `ppSrc` on position `pSrcBitsOffset`. The decoded element is stored in the destination buffer `pDst`.

After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates `pSrcBitsOffset` setting it to the actual bit position in the source buffer.

The function `ippsVLCDecodeOne` uses the VLC table from the decoder specification structure `pVLCSpec` that must be initialized by the functions `ippsVLCDecodeInitAlloc` or `ippsVLCDecodeInit` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used. For decoding functions it can indicate that bitstream contains code that is not specified in the table.
<code>ippStsBitOffsetErr</code>	Indicates an error if value of the <code>pSrcBitsOffset</code> is less than 0 or greater than 7.
<code>ippStsContextMatchErr</code>	Indicates an error if the <code>pVLCSpec</code> structure is not initialized by the function <code>ippsVLCDecodeInitAlloc</code> or <code>ippsVLCDecodeInit</code> .

## Example of Using VLC Coding Functions

The functions `ippsVLCCountBits` and `ippsVLCEncodeBlock` can be simply used in the wide range of different encoders (for example, for audio data). For example, if you need to quantize some vectors and encode them using no more bits than specified, these functions can be used as follows:

- previously the table `inputTable` of the size `inputTableSize` must be created; this table is of the following format:

```
static IppsVLCTable_32s inputTable[]=
{
    {value0, code0, length0};
    {value1, code1, length1};
    ...
    {valueN, codeN, lengthN};
}
```

- initialization step:

```
ippsVLCEncodeGetSize_32s(InputTable, inputTableSize, &Size);
pVLCSpec = ippsMalloc_8u(Size); /* memory allocation */
ippsVLCEncodeInit_32s(pInputTable, inputTableSize, pVLCSpec);
...
```

- counting bits step:

```
Do {
//A some quantization algorithm should be here. Let pSrc is output of this algorithm.
...

```

```
ippsVLCCountBits_16s32s(pSrc, srcLen, &bitNumber, pVLCSpec);
} while (bitNumber > allowedBitNumber);
```

- encoding bit stream step:

```
ippsVLCEncodeBlock_16slu (pSrc, srcLen, ppBitStream, pBitStreamBitsOffset, pVLCSpec);
```

## VLCDecodeUTupleInitAlloc

*Allocates memory and initializes the VLCDecodeUTuple structure based on the input parameters.*

---

### Syntax

```
IppStatus ippsVLCDecodeUTupleInitAlloc_32s(const IppsVLCTable_32s*
pInputTable, int inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables,
int numElements, int numValueBit, IppsVLCDecodeUTupleSpec_32s** ppVLCSpec);
```

### Parameters

<i>pInputTable</i>	Pointer to the input VLC table.
<i>inputTableSize</i>	Size of the input VLC table.
<i>pSubTablesSizes</i>	Pointer to the sizes of the subtables.
<i>numSubTables</i>	Number of the subtables.
<i>numElements</i>	Number of the elements in the tuple.
<i>numValueBit</i>	Size of one element in bits.
<i>ppVLCSpec</i>	Double pointer to the VLC decoder specification structure.

## Description

The function `ippsVLCDecodeUTupleInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the VLC decoder structure `ppVLCSpec` that is specified by the following parameters: the VLC table `pInputTable` of size `inputTableSize`, the number of internal subtables `numSubTables` and their sizes `pSubTablesSizes`, number of elements in tuple `numElements`, and size of one element `numValueBit`.

Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

The VLC decoder specification structure is used by the functions `ippsVLCDecodeUTupleBlock` and `ippsVLCDecodeUTupleOne`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.
<code>ippStsVLCUsrTblCodeLengthErr</code>	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in <code>pSubTablesSizes</code> is less than 0; or if the sum of these values is less than the maximum code length in the input table.

## VLCDecodeUTupleFree

*Frees memory allocated for the VLC decoder structure.*

---

### Syntax

```
void ippsVLCDecodeUTupleFree_32s(IppsVLCDecodeUTupleSpec_32s* pVLCSpec);
```

### Parameters

`pVLCSpec`                      Pointer to the VLC decoder specification structure.

### Description

The function `ippsVLCDecodeUTupleFree` is declared in the `ippdc.h` file. This function frees memory allocated for the VLC decoder structure by the function `ippsVLCDecodeUTupleInitAlloc`.

## VLCDecodeUTupleInit

*Initializes the VLCDecodeUTuple structure.*

---

### Syntax

```

IppStatus ippsVLCDecodeUTupleInit_32s(const IppsVLCTable_32s* pInputTable,
int inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables, int
numElements, int numValueBit, IppsVLCDecodeUTupleSpec_32s* pVLCSpec);

```

### Parameters

<i>pInputTable</i>	Pointer to the input VLC table.
<i>inputTableSize</i>	Size of the input VLC table.
<i>pSubTablesSizes</i>	Pointer to the sizes of the subtables.
<i>numSubTables</i>	Number of the subtables.
<i>numElements</i>	Number of the elements in the tuple.
<i>numValueBit</i>	Size of one element in bits.
<i>pVLCSpec</i>	Pointer to the VLC decoder specification structure.

### Description

The function `ippsVLCDecodeUTupleInit` is declared in the `ippdc.h` file. This function initializes the VLC decoder structure with U-tuple *pVLCSpec* that is specified by the following parameters: the VLC table *pInputTable* of size *inputTableSize*, the number of internal subtables *numSubTables* and their sizes *pSubTablesSizes*, number of elements in tuple *numElements*, and size of one element *numValueBit*.

Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

The VLC decoder specification structure is used by the functions `ippsVLCDecodeUTupleBlock` and `ippsVLCDecodeUTupleOne`.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCUsTblCodeLengthErr</code>	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in <code>pSubTablesSizes</code> is less than 0; or if the sum of these values is less than the maximum code length in the input table.

## VLCDecodeUTupleGetSize

Computes the size of the `VLCDecodUTuple` structure.

---

### Syntax

```
IPPStatus ippSVLCDecodeUTupleGetSize_32s(const IppsVLCTable_32s* pInputTable,
int inputTableSize, Ipp32s* pSubTablesSizes, int numSubTables, int
numElements, int numValueBit, Ipp32s* pSize);
```

### Parameters

<code>pInputTable</code>	Pointer to the input VLC table.
<code>inputTableSize</code>	Size of the input VLC table.
<code>pSubTablesSizes</code>	Pointer to the sizes of the subtables.
<code>numSubTables</code>	Number of the subtables.
<code>numElements</code>	Number of the elements in the tuple.
<code>numValueBit</code>	Size of one element in bits.
<code>pSize</code>	Pointer to the size of VLC decoder specification structure.

### Description

The function `ippSVLCDecodeUTupleGetSize` is declared in the `ippdc.h` file. This function computes the size of the VLC decoder structure with U-tuple that is specified by the following parameters: the VLC table `pInputTable` of size `inputTableSize`, the number of internal subtables `numSubTables` and their sizes `pSubTablesSizes`, number of elements in tuple `numElements`, and size of one element `numValueBit`.

Size values for the subtables must be greater than 0. The sum of these values must be greater than or preferably equal to the maximum code length in the VLC table.

This function must be called prior to the function `ippsVLCDecodeUTupleInit`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCUsrTblCodeLengthErr</code>	Indicates an error if the maximum code length in the input table exceeds 32; or if any size value for the subtables in <code>pSubTablesSizes</code> is less than 0; or if the sum of these values is less than the maximum code length in the input table.

## VLCDecodeUTupleBlock

*Decodes a block of VLC encoded data.*

---

### Syntax

```
IppStatus ippsVLCDecodeUTupleBlock_lu16s(Ipp8u** ppSrc, int* pSrcBitsOffset,
Ipp16s* pDst, int dstLen, const IppsVLCDecodeUTupleSpec_32s* pVLCSpec);
```

### Parameters

<code>ppSrc</code>	Double pointer to the source bitstream.
<code>pSrcBitsOffset</code>	Pointer to the input/output bit position.
<code>pDst</code>	Pointer to the destination buffer.
<code>dstLen</code>	Number of elements in the <code>pDst</code> .
<code>pVLCSpec</code>	Pointer to the VLC decoder specification structure.

### Description

The function `ippsVLCDecodeUTupleBlock` is declared in the `ippdc.h` file. This function decodes `dstLen` VLC encoded elements in the source bitstream starting from the `pSrcBitsOffset` bit position. The function stores the result in the destination buffer `pDst`. After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates `pSrcBitsOffset` setting it to the actual bit position in the source buffer.

The function `ippsVLCDecodeUTupleBlock` uses the VLC table from the decoder specification structure `pVLCSpec` that must be initialized by the functions `ippsVLCDecodeUTupleInitAlloc` or `ippsVLCDecodeUTupleInit` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used. It can indicate that bitstream contains code that is not specified in the table.
<code>ippStsBitOffsetErr</code>	Indicates an error if value of the <code>pSrcBitsOffset</code> is less than 0 or greater than 7.
<code>ippStsContextMatchErr</code>	Indicates an error if the <code>pVLCSpec</code> structure is not initialized by the function <code>ippsVLCDecodeUTupleInitAlloc</code> or <code>ippsVLCDecodeUTupleInit</code> .

## VLCDecodeUTupleOne

*Decodes a single VLC encoded tuple.*

### Syntax

```
IppStatus ippsVLCDecodeUTupleOne_lu16s(Ipp8u** ppSrc, int* pSrcBitsOffset,
Ipp16s* pDst, const IppsVLCDecodeUTupleSpec_32s* pVLCSpec);
```

### Parameters

<code>ppSrc</code>	Double pointer to the source bitstream.
<code>pSrcBitsOffset</code>	Pointer to the input/output bit position.
<code>pDst</code>	Pointer to the decoded value.
<code>pVLCSpec</code>	Pointer to the VLC decoder specification structure.

### Description

The function `ippsVLCDecodeUTupleOne` is declared in the `ippdc.h` file. This function decodes a single VLC encoded element (containing `numElements` samples) in the `ppSrc` on position `pSrcBitsOffset`. The decoded element is stored in the destination buffer `pDst`.

After decoding the function shifts the source pointer by the number of successfully read and processed bytes. The function updates *pSrcBitsOffset* setting it to the actual bit position in the source buffer.

The function `ippVLCDecodeUTupleOne` uses the VLC table from the decoder specification structure *pVLCSpec* that must be initialized by the functions `ippsVLCDecodeUTupleInit` or `ippsVLCDecodeUTupleInitAlloc` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsVLCInputDataErr</code>	Indicates an error if incorrect input is used. For decoding functions it can indicate that bitstream contains code that is not specified in the table.
<code>ippStsBitOffsetErr</code>	Indicates an error if value of the <i>pSrcBitsOffset</i> is less than 0 or greater than 7.
<code>ippStsContextMatchErr</code>	Indicates an error if the <i>pVLCSpec</i> structure is not initialized by the function <code>ippsVLCDecodeUTupleInitAlloc</code> or <code>ippsVLCDecodeUTupleInit</code> .

## Huffman Coding

This section describes functions implementing the Huffman coding algorithm. This algorithm performs so called *entropy* encoding. The main idea of this method is to substitute the code words (Huffman codes) instead of symbols. The least length codes correspond to the most often occurring symbols allowing to perform compression. For example, the sequence "aaaaaaaa" contains only one symbol 'a' that corresponds to the Huffman code with size 1 bit; after encoding the output sequence will have size 1 byte - thus compression ratio will be 8.

### Limitations and conventions.

Intel IPP functions implement the Huffman static encoding. This means that encoding is a two-passes operation. In the first pass the probability structure of the source data is determined, then in the second pass the encoding itself is performed. The table of Huffman Code lengths must be passed to decoder (this table can be effectively compressed).

Maximum length of the Huffman codes in the Intel IPP functions for Huffman coding is 32 bits.

The functions for Huffman encoding/decoding initialize a correspondent specification structure and perform the encoding and decoding operations.

To use the functions for **encoding**, follow this general scheme:

- Call the function `ippsEncodeHuffInitAlloc` to allocate memory and initialize the Huffman structure for encoding with specified frequency table. Or call the function `ippsEncodeHuffInit` to initialize the Huffman structure for encoding in the previously created external buffer. The size of this buffer can be computed by calling the function `ippsHuffGetSize`.
- If encoding is performed **by blocks**, call the function `ippsHuffGetDstBuffSize` to compute the size of the destination buffer.
- Call `ippsEncodeHuffOne` to encode a single element, or call `ippsEncodeHuff` to encode the source sequence (vector). (Alternatively the sequence of N element can be encoded by calling `ippsEncodeHuffOne` N times.) Then call `ippsEncodeHuffFinal` to finalize the encoding procedure. Note that the size of the input sequence (vector) must be passed to the decoding functions.
- Call `ippsHuffGetLenCodeTable` to receive the table containing the lengths of Huffman codes. This table must be passed to the decoding functions. To increase the effectiveness of encoding it may be compressed by calling the function `ippsHuffLenCodeTablePack`.
- Call `ippsHuffFree` to free dynamic memory allocated with the Huffman structure for encoding by the function `ippsEncodeHuffInitAlloc`.

To use these functions for **decoding**, follow this general scheme:

- Call the function `ippsDecodeHuffInitAlloc` to allocate memory and initialize the Huffman structure for decoding with specified table containing lengths of Huffman codes. Or call the function `ippsDecodeHuffInit` to initialize the Huffman structure for encoding in the previously created external buffer. The size of this buffer can be computed by calling the function `ippsHuffGetSize`.
- If decoding is performed by blocks, call the function `ippsHuffGetDstBuffSize` to compute the size of the destination buffer.
- Call `ippsDecodeHuffOne` to encode a single element, or call `ippsDecodeHuff` to encode the source sequence (vector). The size of the source non-encoded sequence (vector) must be passed from the encoding functions as well as the table containing the lengths of Huffman codes. If this table is in the packed form, it may be restored by calling the function `ippsHuffLenCodeTableUnpack`.
- Call `ippsHuffFree` to free dynamic memory allocated with the Huffman structure for decoding by the function `ippsDecodeHuffInitAlloc`.

## EncodeHuffInitAlloc

*Allocates memory and initializes the structure for Huffman encoding.*

### Syntax

```
IppStatus ippsEncodeHuffInitAlloc_8u(const int freqTable[256],
IppHuffState_8u** ppHuffState);
```

## Parameters

<i>freqTable</i>	Table of frequencies of the symbols.
<i>ppHuffState</i>	Double pointer to the Huffman encoding structure.

## Description

The function `ippsEncodeHuffInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the structure for Huffman encoding *pHuffState*. This structure is specifying by the table of symbol's frequencies *freqTable*. The Huffman encoding structure is used by the functions `ippsEncodeHuffOne` and `ippsEncodeHuff`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pHuffState</i> pointer is NULL.
<code>ippStsFreqTableErr</code>	Indicates an error if <i>freqTable</i> is invalid.
<code>ippStsMaxLenHuffCodeErr</code>	Indicates an error if the length of the Huffman code is more than expected one.
<code>ippMemAllocErr</code>	Indicates an error if memory allocation fails.

## HuffFree

*Frees memory allocated for the Huffman encoding and decoding structures.*

---

## Syntax

```
void ippsHuffFree_8u(IppHuffState_8u* pHuffState);
```

## Parameters

<i>pHuffState</i>	Pointer to the Huffman coding structure.
-------------------	--

## Description

The function `ippsHuffFree` is declared in the `ippdc.h` file. This function frees memory allocated for the Huffman encoding and decoding structures by the function `ippsEncodeHuffInitAlloc` and `ippsDecodeHuffInitAlloc` respectively.

## EncodeHuffInit

*Initializes the Huffman encoding structure.*

---

### Syntax

```
IppStatus ippsEncodeHuffInit_8u (const int freqTable[256], IppHuffState_8u* pHuffState);
```

### Parameters

<i>freqTable</i>	Table of frequencies of the symbols.
<i>pHuffState</i>	Pointer to the Huffman encoding structure.

### Description

The function `ippsEncodeHuffInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the structure `pHuffState` that is required for Huffman encoding. This structure is specifying by the table of symbol's frequencies `freqTable`. The size of the external buffer must be computed previously by calling the function `ippsHuffGetSize`. Alternatively, the Huffman encoder structure can be initialized by the function `ippsEncodeHuffInitAlloc`.

The Huffman encoding structure is used by the functions `ippsEncodeHuffOne` and `ippsEncodeHuff`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <code>pHuffState</code> pointer is <code>NULL</code> .
<code>ippStsFreqTableErr</code>	Indicates an error if <code>freqTable</code> is invalid.
<code>ippStsMaxLenHuffCodeErr</code>	Indicates an error if the length of the Huffman code is more than expected one.

## HuffGetSize

*Computes size of the external buffer for the Huffman encoding structure.*

---

### Syntax

```
IppStatus ippsHuffGetSize_8u(int* pHuffStateSize);
```

## Parameters

*pHuffStateSize*                      Pointer to the computed size of the Huffman coding structure.

## Description

The function `ippsHuffGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) for the external buffer required for the Huffman encoding structure. This function must be called prior to the function `ippsEncodeHuffInit`.

## Return Values

`ippsStsNoErr`                      Indicates no error.  
`ippsStsNullPtrErr`                Indicates an error if *pHuffStateSize* pointer is NULL.

## EncodeHuffOne

Performs Huffman encoding of a single element.

## Syntax

```
IppStatus ippsEncodeHuffOne_8u(Ipp8u src, Ipp8u* pDst, int dstOffsetBits,
IppHuffState_8u* pHuffState);
```

## Parameters

*src*                                      Source single element.  
*pDst*                                      Pointer to the destination buffer.  
*dstOffsetBits*                          Offset (in bits) of the position of code word in the destination buffer starting from the high bit.  
*pHuffState*                              Pointer to the Huffman encoding structure.

## Description

The function `ippsEncodeHuffOne` is declared in the `ippdc.h` file. This function encodes the single source element *src* using the Huffman encoding structure *pHuffState*. The result is placed in the destination buffer *pDst* in position specified by the value (in bits) of offset from the high bit *dstOffsetBits*. The Huffman encoding structure must be initialized by calling the functions `ippsEncodeHuffInitAlloc` or `ippsEncodeHuffInit` beforehand.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .

## EncodeHuff

*Performs Huffman encoding.*

---

### Syntax

```
IppStatus ippEncodeHuff_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int* pDstLen, IppHuffState_8u* pHuffState);
```

### Parameters

<code>pSrc</code>	Pointer to the source buffer.
<code>srcLen</code>	Number of elements in the source buffer.
<code>pDst</code>	Pointer to the destination buffer.
<code>pDstLen</code>	Pointer to the number of elements in the destination vector after encoding.
<code>pHuffState</code>	Pointer to the Huffman encoding structure.

### Description

The function `ippEncodeHuff` is declared in the `ippdc.h` file. This function encodes data in the source buffer `pSrc` using the Huffman encoding structure `pHuffState`. The encoded data are placed in the destination buffer `pDst`. The actual number of elements in the destination buffer is written in `pDstLen`. The Huffman encoding structure must be initialized by calling the functions `ippEncodeHuffInitAlloc` or `ippEncodeHuffInit` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>srcLen</code> is less than or equal to 0.

## EncodeHuffFinal

*Performs Huffman encoding of the remainder.*

---

### Syntax

```
IppStatus ippEncodeHuffFinal_8u(Ipp8u* pDst, int* pDstLen, IppHuffState_8u* pHuffState);
```

### Parameters

<i>pDst</i>	Pointer to the destination vector.
<i>pDstLen</i>	Pointer to the number of elements in the destination vector after encoding.
<i>pHuffState</i>	Pointer to the Huffman coding structure.

### Description

The function `ippEncodeHuffFinal` is declared in the `ippdc.h` file. This function performs the final stage of the Huffman encoding - flushes the remainder part of the encoded data to the destination vector *pDst* with *pDstLen* elements using the encoding structure *pHuffState*. The function must be run just after `ippEncodeHuff`, while the Huffman encoding structure *pHuffState* is still initialized.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .

## HuffGetLenCodeTable

*Returns the table containing lengths of Huffman codes.*

---

### Syntax

```
IppStatus ippHuffGetLenCodeTable_8u(int codeLenTable[256], IppHuffState_8u* pHuffState);
```

### Parameters

<i>codeLenTable</i>	Destination table containing lengths of Huffman codes.
---------------------	--

*pHuffState*                      Pointer to the pointer to the Huffman decoding structure.

### Description

The function `ippsHuffGetLenCodeTable` is declared in the `ippdc.h` file. This function creates the table *codeLenTable* containing the lengths of the used Huffman codes. Their value are taken from the Huffman encoding structure *pHuffState*, which must be initialized beforehand.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error if the *pHuffState* pointer is `NULL`.

## DecodeHuffInitAlloc

*Allocates memory and initializes the Huffman decoding structure.*

---

### Syntax

```
IppStatus ippsDecodeHuffInitAlloc_8u(const int codeLenTable[256],
IppHuffState_8u** ppHuffState);
```

### Parameters

*codeLenTable*                      Table containing lengths of Huffman codes.  
*ppHuffState*                        Double pointer to the Huffman decoding structure.

### Description

The function `ippsDecodeHuffInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the Huffman decoding structure *pHuffState*. The structure requires the table of lengths of Huffman codes *codeLenTable* that was used in the encoding operation. This table can be obtained by calling the function `ippsHuffGetLenCodeTable` or `ippsHuffLenCodeTableUnpack`. The Huffman decoding structure is used by the functions `ippsDecodeHuffOne` and `ippsDecodeHuff`.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error if *pHuffState* pointer is `NULL`.

`ippMemAllocErr` Indicates an error if memory allocation fails.

## DecodeHuffInit

*Initialized the Huffman decoding structure.*

---

### Syntax

```
IppStatus ippDecodeHuffInit_8u(const int codeLenTable[256], IppHuffState_8u* pHuffState);
```

### Parameters

*codeLenTable* Table containing lengths of Huffman codes.  
*pHuffState* Pointer to the Huffman decoding structure.

### Description

The function `ippDecodeHuffInit` is declared in the `ippdc.h` file. This function initializes in the external buffer the Huffman decoding structure *pHuffState*. The structure requires the table of lengths of Huffman codes *codeLenTable* that was used in the encoding operation. This table can be obtained by calling the function `ippHuffGetLenCodeTable` or `ippHuffLenCodeTableUnpack`. The size of the external buffer must be computed previously by calling the function `ippHuffGetSize`. Alternatively, the Huffman decoding structure can be initialized by the function `ippDecodeHuffInitAlloc`.

The Huffman decoding structure is used by the functions `ippDecodeHuffOne` and `ippDecodeHuff`.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error if *pHuffState* pointer is NULL.

## DecodeHuffOne

*Decodes a single code word.*

---

### Syntax

```
IppStatus ippDecodeHuffOne_8u(const Ipp8u* pSrc, int srcOffsetBits, Ipp8u* pDst, IppHuffState_8u* pHuffState);
```

## Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcOffsetBits</i>	Offset (bits) in the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pHuffState</i>	Pointer to the Huffman decoding structure.

## Description

The function `ippsDecodeHuffOne` is declared in the `ippdc.h` file. This function performs Huffman decoding of the single code word in the source buffer *pSrc*. Its position is specified by the value (in bits) of offset from the high bit *srcOffsetBits*. The decoded element is stored in the destination buffer *pDst*. The function uses the Huffman decoding structure *pHuffState* that must be initialized by calling the functions `ippsDecodeHuffInitAlloc` or `ippsDecodeHuffInit` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .

# DecodeHuff

*Performs Huffman decoding.*

---

## Syntax

```
IppStatus ippsDecodeHuff_8u(const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int* pDstLen, IppHuffState_8u* pHuffState);
```

## Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Number of elements in the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the expected number of elements in the destination buffer; after decoding to the actual number of elements in the destination buffer.
<i>pHuffState</i>	Pointer to the Huffman coding structure.

## Description

The function `ippsDecodeHuff` is declared in the `ippdc.h` file. This function decodes the data in the source buffer `pSrc` containing `srcLen` code words. The `pDstLen` decoded elements are stored in the destination vector `pDst`. The function uses the Huffman decoding structure `pHuffState` that must be initialized by calling the functions `ippsDecodeHuffInitAlloc` or `ippsDecodeHuffInit` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>srcLen</code> is less than or equal to 0.

## HuffGetDstBuffSize

*Computes the size of a destination buffer for Huffman encoding/decoding by blocks.*

---

### Syntax

```
IppStatus ippsHuffGetDstBuffSize_8u(const int codeLenTable[256], int srcLen,
int* pEncDstBuffSize, int* pDecDstBuffSize);
```

### Parameters

<code>codeLenTable</code>	Table with lengths of Huffman codes.
<code>srcLen</code>	Number of elements in the source buffer.
<code>pEncDstBuffSize</code>	Pointer to the computed size of the destination buffer for Huffman encoding.
<code>pDecDstBuffSize</code>	Pointer to the computed size of the destination buffer for Huffman decoding.

## Description

The function `ippsHuffGetDstBuffSize` is declared in the `ippdc.h` file. This function computes the size (in bytes) of destination buffer for Huffman encoding and decoding by blocks. It must be called prior to the encoding/decoding operations.

The function computes the size of destination buffers for both operations. To exclude one of them the correspondent pointer must be set to `NULL`.

## Application Notes

The following formula can be used to estimate maximum sizes of the destination buffer for the Huffman encoding and decoding by blocks:

*encDstBuffSize* = *srcLen*\*4,

*decDstBuffSize* = *srcLen*\*8

You can use these values without calling the function `ippsHuffGetDstBuffSize`. Nevertheless the function `ippsHuffGetDstBuffSize` allows to minimize the sizes of the destination buffers.

## Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error if the *codeLenTable* pointer is NULL.

## HuffLenCodeTablePack

*Packs the table containing lengths of Huffman codes.*

---

### Syntax

```
ippStatus ippsHuffLenCodeTablePack_8u(const int codeLenTable[256], Ipp8u*
pDst, int* pDstLen);
```

### Parameters

<i>codeLenTable</i>	Table containing lengths of Huffman codes.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the number of elements in the destination buffer; after operation to the actual number of elements in the destination buffer.

### Description

The function `ippsHuffLenCodeTablePack` is declared in the `ippdc.h` file. This function compresses the table *codeLenTable* containing lengths of Huffman codes and that is required for decoding. The table may be retrieved by calling the function `ippsHuffGetLenCodeTable`. To restore the table from the compressed form, call the function `ippsHuffLenCodeTableUnpack`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>pDstLen</i> is less than or equal to 0; or if <i>pDstLen</i> is less than actual size of the output.

## HuffLenCodeTableUnpack

*Unpacks the table containing lengths of Huffman codes.*

---

### Syntax

```
IppStatus ippHuffLenCodeTableUnpack_8u(const Ipp8u* pSrc, int* pSrcLen, int
codeLenTable[256]);
```

### Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the number of elements in the source buffer; after operation pointer to the actual number of elements in the source buffer.
<i>codeLenTable</i>	Table containing lengths of Huffman codes.

### Description

The function `ippHuffLenCodeTableUnpack` is declared in the `ippdc.h` file. This function restores the table *codeLenTable* containing the lengths of Huffman codes from the packed form *pSrc* created by the function [HuffLenCodeTablePack](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>pSrcLen</i> is less than or equal to 0; or if <i>pSrcLen</i> is less than actual size.



## Dictionary-Based Compression Functions

This section describes the Intel IPP functions that use different dictionary-based compression methods.

The full list of these functions is given in [Table 13-2](#).

### LZSS Compression Functions

These functions implement the LZSS (Lempel-Ziv-Storer-Szymanski) compression algorithm [[Storer82](#)]. The functions perform LZSS coding with a vocabulary size of 32KB and 256-byte maximum match string length.

### EncodeLZSSInitAlloc

*Allocates memory and initializes the LZSS encoder state structure.*

---

#### Syntax

```
IppStatus ippEncodeLZSSInitAlloc_8u (IppLZSSState_8u** ppLZSSState);
```

#### Parameters

*ppLZSSState*                      Double pointer to the LZSS encoder state structure.

#### Description

The function `ippEncodeLZSSInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZSS encoder state structure *ppLZSSState* required for the encoder functions [ippEncodeLZSS](#) and [ippEncodeLZSSFlush](#).

#### Return Values

<code>ippStsNoErr</code>	Indicates no errors.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>ppLZSSState</i> pointer is <code>NULL</code> .
<code>StsMemAllocErr</code>	Indicates an error if memory allocation fails.

## LZSSFree

*Frees memory allocated for the LZSS state structure.*

---

### Syntax

```
void ippLZSSFree_8u (IppLZSSState_8u* pLZSSState);
```

### Parameters

*pLZSSState*                      Pointer to the LZSS encoder state structure.

### Description

The function `ippEncodeLZSSFree` is declared in the `ippdc.h` file. This function frees memory allocated for the LZSS state structure for encoding and decoding by the function `ippEncodeLZSSInitAlloc` or `ippDecodeLZSSInitAlloc` respectively.

## EncodeLZSSInit

*Initializes the LZSS encoder state structure.*

---

### Syntax

```
IppStatus ippEncodeLZSSInit_8u (IppLZSSState_8u* pLZSSState);
```

### Parameters

*pLZSSState*                      Pointer to the LZSS encoder state structure.

### Description

The function `ippEncodeLZSSInit` is declared in the `ippdc.h` file. This function initializes the LZSS state structure *pLZSSState* in the external buffer. Its size must be computed previously by calling the function `ippLZSSGetSize`. Alternatively, this state structure can be initialized by the function `ippEncodeLZSSInitAlloc`.

The LZSS encoder state structure is required for the encoder functions `ippEncodeLZSS` and `ippEncodeLZSSFlush`.

### Return Values

`ippStsNoErr`                      Indicates no error.

`ippStsNullPtrErr` Indicates an error if the pointer `pLZSSState` is NULL.

## LZSSGetSize

*Computes the size of the LZSS state structure.*

---

### Syntax

```
IppStatus ippLZSSGetSize_8u (int* pLZSSStateSize);
```

### Parameters

`pLZSSStateSize` Pointer to the size of the LZSS state structure.

### Description

The function `ippLZSSGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the LZSS state structure for encoding and decoding and stores it to an integer pointed to by `pLZSSStateSize`. The function must be called prior to the function [ippEncodeLZSSInit](#) or [ippDecodeLZSSInit](#).

### Return Values

`ippStsNoErr` Indicates no error.

`ippStsNullPtrErr` Indicates an error if the pointer `pLZSSStateSize` is NULL.

## EncodeLZSS

*Performs LZSS encoding.*

---

### Syntax

```
IppStatus ippEncodeLZSS_8u (Ipp8u** ppSrc, int* pSrcLen, Ipp8u** p pDst,
int* pDstLen, IppLZSSState_8u* pLZSSState);
```

### Parameters

`ppSrc` Double pointer to the source buffer.

`pSrcLen` Pointer to the number of elements in the source buffer; it is updated after encoding.

`ppDst` Double pointer to the destination buffer.

<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>pLZSSState</i>	Pointer to the LZSS encoding state structure.

## Description

The function `ippsEncodeLZSS` is declared in the `ippdc.h` file. This function performs LZSS encoding of data in the source buffer *ppSrc* with *pSrcLen* length and stores the result in the destination vector *pDst*. The length of the destination vector is stored in *pDstLen*. The LZSS encoder state structure *pLZSSState* must be initialized by the functions `ippsEncodeLZSSInit` or `ippsEncodeLZSSInitAlloc` beforehand.

After encoding the function returns the pointers to source and destination buffers shifted by the number of successfully read and encoded bytes, respectively. The function updates *pSrcLen* so it is equal to the actual number of elements in the source buffer.

Example 13-1 shows how to use the function `ippsEncodeLZSS_8u` and supporting functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>srcLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

## EncodeLZSSFlush

*Encodes the last few bits in the bitstream and aligns the output data on the byte boundary.*

---

### Syntax

```
IppStatus ippsEncodeLZSSFlush_8u (Ipp8u** ppDst, int* pDstLen,
IppLZSSState_8u* pLZSSState);
```

### Parameters

<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of destination buffer.
<i>pLZSSState</i>	Pointer to the LZSS encoder state structure.

## Description

The function `ippsEncodeLZSSFlush` is declared in the `ippdc.h` file. This function encodes the last few bits (remainder) in the bitstream, writes them to `ppDst`, and aligns the output data on a byte boundary.

Example 13-1 below shows how to use the function `ippsEncodeLZSSFlush_8u` and supporting functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>pDstLen</code> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

## Example 13-1. Using the Function `ippsEncodeLZSS_8u` and Supporting Functions

```
#define INBLOCKSIZE 65536
#define OUTBLOCKSIZE 16384

.....

#include <stdio.h>
#include "ippdc.h"
.....

FILE                *inFile, *outFile;
Ipp8u               *pSrc = NULL, *pDst = NULL, *startpSrc, *startpDst;
int                 srcLen, dstLen = 0;
int                 commonDstLen;
IppLZSSState_8u     *pLZSSState;

IppStatus            status;
```

```

.....
/*****
/* Memory allocation for input data and output code buffers, */
/* and for the internal encoding state structure:          */
/*****
startpSrc = ippsMalloc_8u( (INBLOCKSIZE) * sizeof(char) );
startpDst = ippsMalloc_8u( (OUTBLOCKSIZE) * sizeof(char) );
ippsEncodeLZSSInitAlloc_8u(&pLZSSState);
/*****
/* Initializing the arguments, reading srcLen bytes of      */
/* input data from input data file to startpSrc:            */
/*****
commonDstLen = (OUTBLOCKSIZE);
pDst = startpDst;
dstLen = commonDstLen;
srcLen = fread( startpSrc, sizeof(Ipp8u), INBLOCKSIZE, inFile );
if(srcLen <= 0)
    return -1;
pSrc = startpSrc;
/*****
/* The main loop. In every iteration program calls the      */
/* ippsEncodeLZSS_8u. ippsEncodeLZSS_8u changes the values  */
/* of pSrc, srcLen, pDst and dstLen. ippsEncodeLZSS_8u      */
/* returns ippsStsDstSizeLessExpected if there is no        */
/* room in the destination buffer pDst to continue encoding. */
/* In this case dstLen == 0. Such case is handled by         */
/* flushing the pDst to the output file. If ippsEncodeLZSS_8u */
/* returns ippsStsNoErr, then the program flushes the current */

```

```
/* pDst of length of (commonDstLen - dstLen) bytes to */
/* the output file and reads next srcLen bytes of input data */
/* to startpSrc. */
/*****/
for( ; ; )
{
    status = ippsEncodeLZSS_8u(&pSrc, &srcLen, &pDst, &dstLen, pLZSSState);
    if( status == ippStsDstSizeLessExpected )
    {
        fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
        dstLen = commonDstLen;
        pDst = startpDst;
    }
    else if( status == ippStsNoErr )
    {
        fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
        pDst = startpDst;
        dstLen = commonDstLen;
        srcLen = fread( startpSrc, sizeof(Ipp8u), INBLOCKSIZE, inFile );
        if( srcLen <= 0 )
        {
            break;
        }
        pSrc = startpSrc;
    }
}
```

```

        else
            return (-1);
    } /* for */

    /*****
    /* The call of ippsEncodeLZSSFlush_8u flushes the last few
    /* bits of code to the destination buffer:
    *****/

    if( ippsEncodeLZSSFlush_8u(&pDst, &dstLen, pLZSSState) == ippsStsNoErr )
    {

        fwrite( startpDst, sizeof(Ipp8u), (commonDstLen - dstLen), outFile );
        ippsLZSSFree_8u(pLZSSState);
        ippsFree( startpSrc );
        ippsFree( startpDst );
        _fcloseall();
    }
    else
        return (-1);

    .....

```

## DecodeLZSSInitAlloc

*Allocates memory and initializes the LZSS decoder state structure.*

---

### Syntax

```
IppStatus ippsDecodeLZSSInitAlloc_8u (IppLZSSState_8u** ppLZSSState);
```

### Parameters

*ppLZSSState*                      Double pointer to the LZSS decoder state structure.



## Description

The function `ippsDecodeLZSSInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZSS decoder state structure. That structure is used by the function `ippsDecodeLZSS`.

## Return Values

<code>ippStsNoErr</code>	Indicates no errors.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pLZSSState</code> is NULL.
<code>ippStsMemAllocErr</code>	Indicates an error if memory allocation fails.

## DecodeLZSSInit

*Initializes the LZSS decoder state structure.*

---

### Syntax

```
IppStatus ippsDecodeLZSSInit_8u (IppLZSSState_8u* pLZSSState);
```

### Parameters

<code>pLZSSState</code>	Pointer to the LZSS decoder state structure.
-------------------------	--

### Description

The function `ippsDecodeLZSSInit` is declared in the `ippdc.h` file. This function initializes the LZSS decoder state structure in the external buffer, the size of which must be computed previously by calling the function `ippsLZSSGetSize`. Alternatively, this state structure can be initialized by the function `ippsDecodeLZSSInitAlloc`.

The LZSS decoder state structure is required for the function `ippsDecodeLZSS`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pLZSSState</code> is NULL.

## DecodeLZSS

*Performs LZSS decoding.*

---

### Syntax

```
IppStatus ippDecodeLZSS_8u (Ipp8u* ppSrc, int* pSrcLen, Ipp8u*8 ppDst, int*
pDstLen, IppLZSSState_8u* pLZSSState);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>pLZSSState</i>	Pointer to the LZSS decoding state structure.

### Description

The function `ippDecodeLZSS` is declared in the `ippdc.h` file. This function performs LZSS decoding of the *pSrcLen* elements of the *ppSrc* source buffer and stores the result in the *pDst* destination vector. The length of the destination vector is stored in *pDstLen*. The LZSS decoder state structure *pLZSSState* must be initialized by the functions [ippDecodeLZSSInitAlloc](#) or [ippDecodeLZSSInit](#) beforehand.

After decoding the function returns the pointers to source and destination buffers shifted by the number of successfully read and decoded bytes respectively. The function updates *pSrcLen* so it is equal to the actual number of elements in the source buffer.

Example 13-2 below shows how to use the `ippDecodeLZSS_8u` function and supporting functions.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>pSrcLen</i> or <i>pDstLen</i> is negative.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

**Example 13-2 Using the Function `ippsDecodeLZSS` and Supporting functions**

```
#define INBLOCKSIZE 65536
#define OUTBLOCKSIZE 16384

.....
#include <stdio.h>
#include "ippdc.h"
.....
FILE                *inFile, *outFile;
Ipp8u                *pSrc = NULL, *pDst = NULL, *startpSrc, *startpDst;
int                  srcLen, dstLen = 0;
IppLZSSState_8u      *pLZSSState;

IppStatus            status;
```

```

.....
/*****
/* Memory allocation for input code and output restored data */
/* buffers, and for the internal decoding state structure: */
/*****
startpSrc = ippsMalloc_8u( OUTBLOCKSIZE * sizeof(Ipp8u) );
startpDst = ippsMalloc_8u( INBLOCKSIZE * sizeof(Ipp8u) );
ippsDecodeLZSSInitAlloc_8u( &pLZSSState );
pSrc = startpSrc;
pDst = startpDst;
/*****
/* The main loop. In every iteration program calls the ippsDecodeLZSS_8u. */
/* ippsDecodeLZSS_8u changes the values of pSrc, srcLen, pDst and dstLen. */
/* ippsDecodeLZSS_8u* returns ippsStsDstSizeLessExpected if there is no */
/* room in the destination buffer pDst to continue decoding. In this case */
/* dstLen == 0. Such case is handled by flushing the pDst to the output */
/* file. If ippsDecodeLZSS_8u returns ippsStsNoErr, then program flushes */
/* the current pDst of length of (INBLOCKSIZE - dstLen) bytes to the */
/* output file and reads next srcLen bytes of input code to startSrc */
/*****
for( ; ; )

{
    status = ippsDecodeLZSS_8u( &pSrc, &srcLen, &pDst, &dstLen, pLZSSState);

```

---

```

    if( status == ippStsNoErr )
    {
        fwrite(startpDst, sizeof(Ipp8u), (INBLOCKSIZE - dstLen), outFile);

        pDst = startpDst;
        dstLen = INBLOCKSIZE;

        srcLen = fread( startpSrc, sizeof(Ipp8u), OUTBLOCKSIZE, inFile );

        pSrc = startpSrc;
        if(srcLen == 0)
            break;
    }
    else if( status == ippStsDstSizeLessExpected )
    {
        fwrite(startpDst, sizeof(Ipp8u), (INBLOCKSIZE - dstLen), outFile);

        pDst = startpDst;
        dstLen = INBLOCKSIZE;
    }
} /* for */

/*****
/* Free the memory allocated for input code and output      */
/* restored data buffers and for the internal decoding state */
/* structure:                                                */
*****/

ippsLZSSFree_8u( pLZSSState );
ippsFree( startpSrc );
ippsFree( startpDst );
_fcloseall();
.....

```

## ZLIB Coding Functions

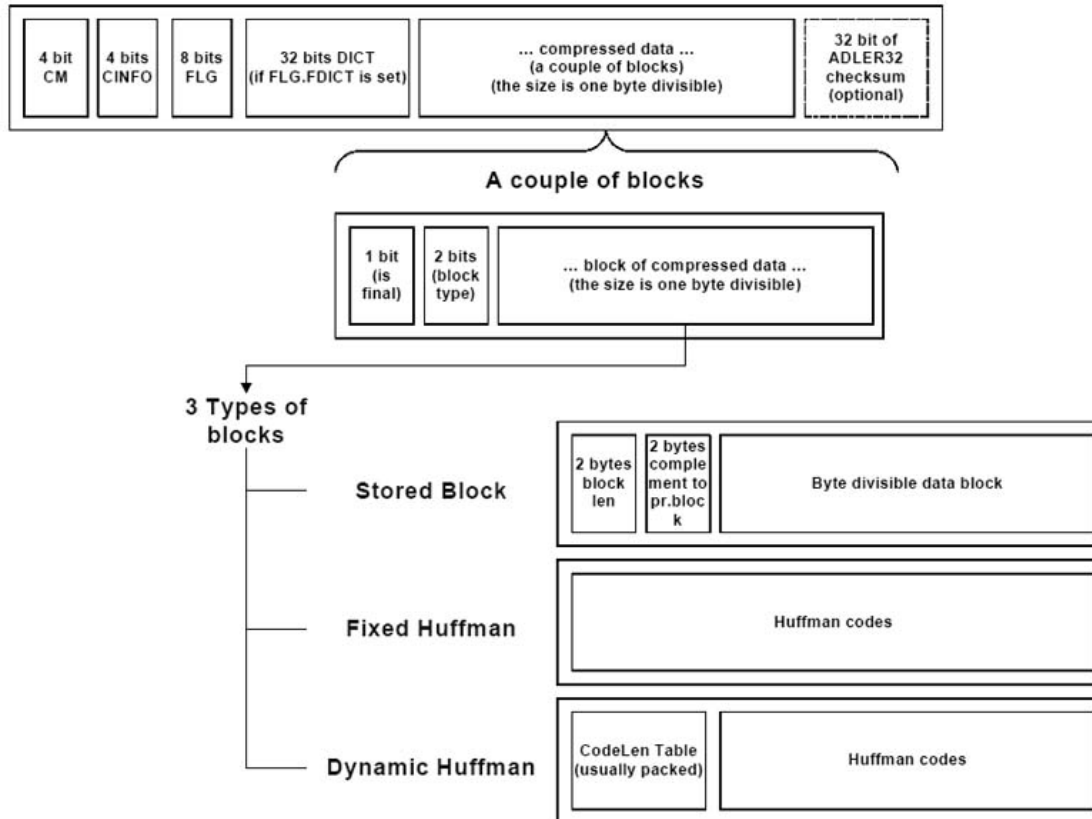
This section describes Intel IPP data compression functions that implement compression methods and data formats defined by the following specifications: [\[RFC1950\]](#), [\[RFC1951\]](#), and [\[RFC1952\]](#). These formats are also known as ZLIB, DEFLATE, and GZIP, respectively.

A basic algorithm for these data compression methods is based on the Lempel-Ziv (LZ77) [\[Ziv77\]](#) dictionary-based compression.

The structure of ZLIB data is schematically shown in [Figure 13-1](#).

Figure 13-2 and Figure 13-3 show the schemes of applying the Intel IPP ZLIB functions for encoding and decoding, respectively. The functions which names are bolded are the mandatory functions for the specified stage.

**Figure 13-1 ZLIB Data Structure**



**Figure 13-2 Encoding Data Block with Intel IPP ZLIB Functions**

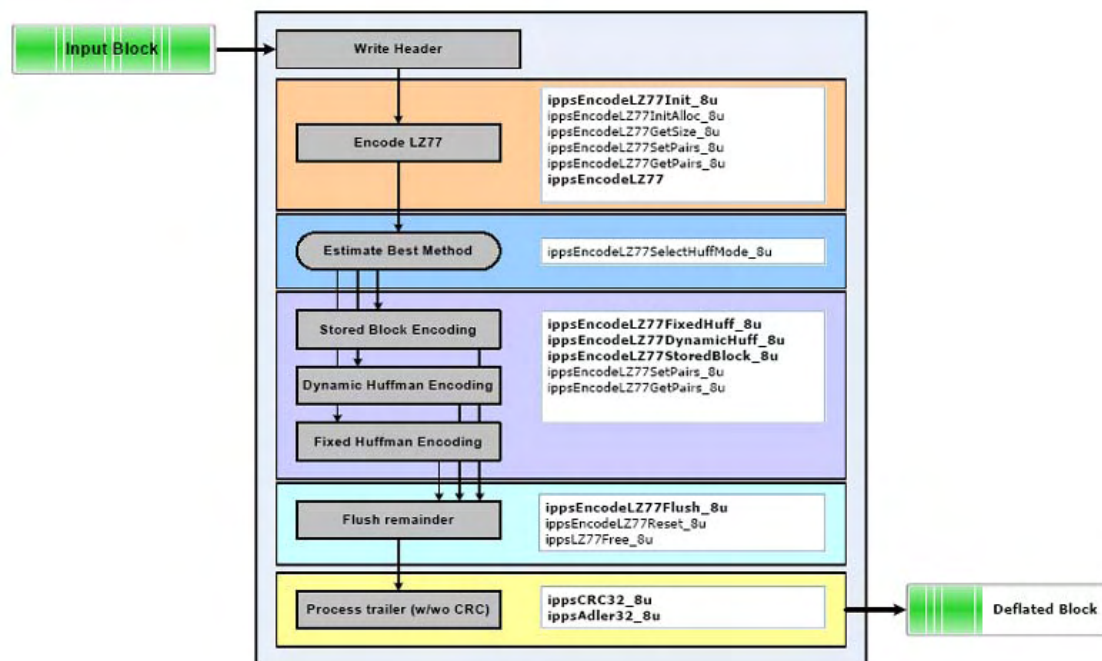
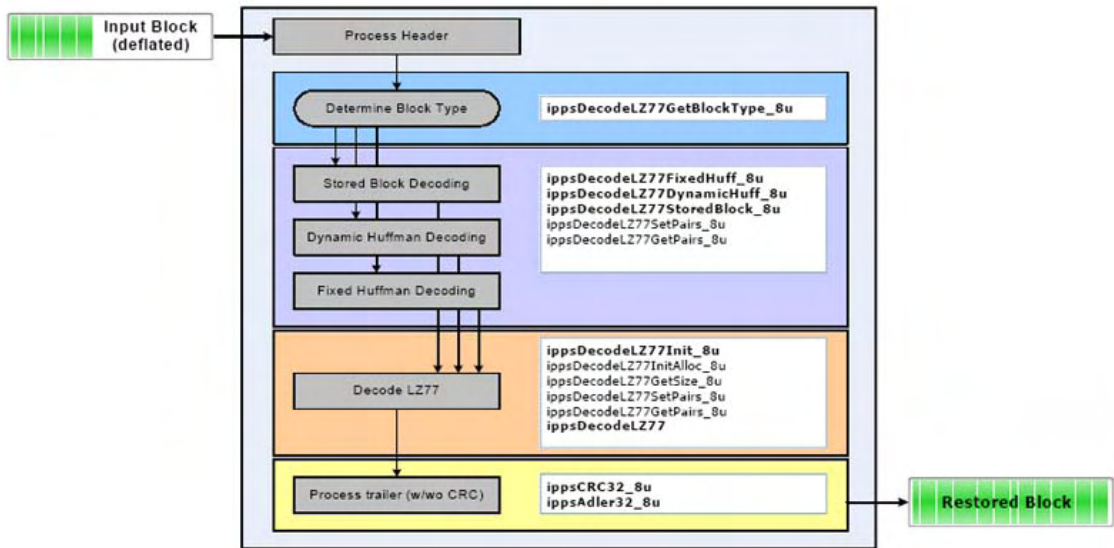




Figure 13-3 Decoding Data Block with Intel IPP ZLIB Functions



The full version of the `zlib` library based on the Intel IPP is presented as the Intel IPP Sample downloadable form  
<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm>.

Special Parameters

The ZLIB coding functions have several special parameters.

The `comprLevel` parameter specifies the level of compression rate and compression ratio. Table 13-5 below lists the possible values of the `comprLevel` parameter and their meanings.

Table 13-5. Parameter `comprLevel` for ZLIB Functions

Value	Descriptions
<code>IppLZ77FastCompr</code>	Fast compression, maximum compression rate, and below average compression ratio
<code>IppLZ77AverageCompr</code>	Average compression rate, average compression ratio
<code>IppLZ77BestCompr</code>	Slow compression, maximum compression ratio

The `checksum` parameter specifies what algorithm is used to compute checksum for input data. Table 13-6 below lists the possible values of the `checksum` parameter and their meanings.

**Table 13-6. Parameter *checksum* for ZLIB Functions**

Value	Description
IppLZ77NoChcksm	Checksum is not calculated.
IppLZ77Adler32	Checksum is calculated using Adler32 algorithm.
IppLZ77CRC32	Checksum is calculated using the CRC32 algorithm.

The *flush* parameter specifies the encoding mode for data block encoding. Table 13-7 below lists the possible values of the *flush* parameter and their meanings.

**Table 13-7. Parameter *flush* for ZLIB Functions**

Value	Descriptions
IppLZ77NoFlush	The end of the block is aligned to a byte boundary.
IppLZ77SyncFlush	The end of the block is aligned to a byte boundary, and 4-byte marker is written to pDst.
IppLZ77FullFlush	The end of the block is aligned to a byte boundary, 4-byte marker is written to pDst, sliding dictionary is zeroed.
IppLZ77FinishFlush	The end of the block is aligned to a byte boundary and the function returns the <code>ippStsStreamEnd</code> status.

The *deflateStatus* parameter specifies the encoding status to ensure the compatibility with the [RFC1951](#) specification. This parameter is used by Intel IPP ZLIB encoding functions. Table 13-8 below lists the possible values of the *deflateStatus* parameter and their meanings.

**Table 13-8. Parameter *deflateStatus* for ZLIB Encoding Functions**

Value	Descriptions
IppLZ77StatusInit	Specified the deflate implementation of encoding functions, must be used before stream encoding.
IppLZ77StatusLZ77Process	Call the deflate implementation of encoding function.
IppLZ77StatusHuffProcess	Call the deflate implementation of the encoding function with the fixed Huffman codes.
IppLZ77StatusFinal	Specified the last block in the stream.

The *inflateStatus* parameter specifies the decoding status to ensure the compatibility with the [RFC1951](#) specification. This parameter is used by Intel IPP ZLIB decoding functions. Table 13-9 below lists the possible values of the *inflateStatus* parameter and their meanings.

**Table 13-9. Parameter *inflateStatus* for ZLIB Decoding Functions**

Value	Descriptions
IppLZ77inflateStatusInit	Specified the deflate implementation of encoding functions, must be used before stream encoding.
IppLZ77InflateStatusHuffProcess	Call the deflate implementation of the encoding function with the fixed Huffman codes.

Value	Descriptions
IppLZ77InflateStatusLZ77Process	Call the deflate implementation of encoding function.
IppLZ77InflateStatusFinal	Specified the last block in the stream.

## EncodeLZ77Init

*Initializes the LZ77 encoding state structure.*

### Syntax

```
IppStatus ippsEncodeLZ77Init_8u (IppLZ77comprLevel comprLevel, ipplZ77chcksm
checksum, IppLZ77_8u* pLZ77State);
```

### Parameters

<i>comprLevel</i>	Specifies desired compression level (see <a href="#">comprLevel Parameter</a> ).
<i>checksum</i>	Specifies an algorithm to compute the checksum for input data (see <a href="#">checkSum Parameter</a> ).
<i>ppLZ77State</i>	Pointer to the LZ77 encoding state structure.

### Description

The function `ippsEncodeLZ77Init` is declared in the `ippdc.h` file. This function initializes the LZ77 encoding state structure in the external buffer. Its size must be previously computed by calling the function `ippsEncodeLZ77GetSize`. The function implements different algorithms to compute the checksum for input data in accordance with the value of the parameter `checksum` and different compression levels that are specified by the parameter `comprLevel`. The encoding state structure is used by different ZLIB encoder functions.

Alternatively, this state structure can be initialized by the function `ippsEncodeLZ77InitAlloc`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> pointer is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error if the <code>checksum</code> or <code>comprLevel</code> parameter has an illegal value.

## EncodeLZ77GetSize

*Computes the size of the LZ77 encoding state structure.*

---

### Syntax

```
IppStatus ippsEncodeLZ77GetSize_8u (int* pLZ77VLCStateSize);
```

### Parameters

*pLZ77StateSize*                      Pointer to the size of the LZ77 encoding state structure.

### Description

The function `ippsEncodeLZ77GetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the LZ77 encoding state structure. The function should be called prior to the function `ippsEncodeLZ77Init`.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error if the pointer *pLZ77StateSize* is NULL.

## EncodeLZ77InitAlloc

*Allocates memory and initializes the encoding state structure.*

---

### Syntax

```
IppStatus ippsEncodeLZ77InitAlloc_8u (IppLZ77comprLevel comprLevel,  
IppLZ77chcksm checksum, IppLZ77_8u** ppLZ77State);
```

### Parameters

*comprLevel*                      Specifies desired compression level (see [comprLevel Parameter](#)).  
*checksum*                        Specifies an algorithm to compute the checksum for input data (see [checksum Parameter](#)).  
*ppLZ77State*                    Double pointer to the LZ77 encoding state structure.

## Description

The function `ippsEncodeLZ77InitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the LZ77 encoding state structure `ppLZ77State`. The function implements different algorithms to compute the checksum for input data in accordance with the value of the parameter `checksum` and different compression levels that are specified by the parameter `comprLevel`. The encoding state structure is used by different ZLIB encoder functions.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>ppLZ77State</code> pointer is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error if the <code>checksum</code> or <code>comprLevel</code> parameter has an illegal value.
<code>ippStsMemAlloc</code>	Indicates an error if memory allocation fails.

## LZ77Free

*Frees memory allocated for the LZ77 encoding and decoding structures.*

---

## Syntax

```
void ippsLZ77Free_8u (IppLZ77State_8u* pLZ77State);
```

## Parameters

<code>pLZ77State</code>	Pointer to the LZ77 encoding or decoding state structure.
-------------------------	---

## Description

The function `ippsLZ77Free` is declared in the `ippdc.h` file. This function frees memory allocated for the LZ77 encoding or decoding state structures.

## EncodeLZ77

*Performs LZ77 encoding.*

---

### Syntax

```
IppStatus ippsEncodeLZ77_8u (Ipp8u** ppSrc, int* pSrcLen, IppLZ77Pair**
ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>flush</i>	Specifies the encoding mode for data blocks (see <a href="#">flush Parameter</a> ).
<i>pLZ77State</i>	Pointer to the LZ77 encoding state structure.

### Description

The function `ippsEncodeLZ77` is declared in the `ippdc.h` file. This function performs ZLIB encoding [ Ziv77 ] using the sliding window technique in accordance with the [RFC1951](#) specification.

The function uses the LZ77 encoding state structure `pLZ77State` that must be previously initialized by the function `ippsEncodeLZ77InitAlloc` or `ippsEncodeLZ77Init`.

The output data are of `IppLZ77Pair` type that is defined as follows:

```
typedef struct IppLZ77Pairs {
    Ipp16u length;
    Ipp16u offset;
} IppLZ77Pair;
```

where `length` is a match length, and `offset` is a distance between the far-left position of the look-ahead buffer and the first literal of the string match in the sliding window. If the match is not found, the `offset` field contains 0 and the `length` field contains a single-literal code.

[Code example 13-3](#) demonstrates how to use the function `ippsEncodeLZ77`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>dstLen</code> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

## EncodeLZ77SelectHuffMode

*Determines the optimal encoding mode.*

---

### Syntax

```
IppStatus ippEncodeLZ77SelectHuffMode_8u(IppLZ77Pair* pSrc, int srcLen,
IppLZ77HuffMode* pHuffMode, IppLZ77State_8u* pLZ77State);
```

### Parameters

<code>pSrc</code>	Pointer to the source buffer.
<code>srcLen</code>	Length of the source buffer.
<code>pHuffMode</code>	Pointer to the value of the parameter that specified the optimal encoding mode. Possible values are: <code>IppLZ77UseFixed</code> Encoding with fixed Huffman codes. <code>IppLZ77UseDynamic</code> Encoding with dynamic Huffman codes.
<code>pLZ77State</code>	Pointer to the LZ77 encoding state structure.

### Description

The function `ippEncodeLZ77SelectHuffMode` is declared in the `ippdc.h` file. This function analyzes the statistics of the source data stream `pSrc` of the `IppLZ77Pair` type and determines the optimal encoding mode - with dynamic Huffman codes or with fixed Huffman codes. The function returns the corresponding value of the parameter `pHuffMode`. Source data are produced by the function [ippEncodeLZ77](#).

This optimal mode is recommended, but you can use your own methods to encode the source data.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> or <code>pPairs</code> pointer is NULL.

## EncodeLZ77FixedHuff

*Performs fixed Huffman encoding.*

---

### Syntax

```
IppStatus ippsEncodeLZ77FixedHuff_8u (IppLZ77Pair** ppSrc, int* pSrcLen,
Ipp8u** ppDst, int* pDstLen, IppLZ77flush flush, IppLZ77_8u* pLZ77State);
```

### Parameters

<code>ppSrc</code>	Double pointer to the source buffer.
<code>pSrcLen</code>	Pointer to the length of the source buffer.
<code>ppDst</code>	Double pointer to the destination buffer.
<code>pDstLen</code>	Pointer to the length of the destination buffer.
<code>flush</code>	Specifies the encoding mode for data blocks (see <a href="#">flush Parameter</a> ).
<code>pLZ77State</code>	Pointer to the LZ77 encoding state structure.

### Description

The function `ippsEncodeLZ77FixedHuff` is declared in the `ippdc.h` file. This function performs encoding with the fixed Huffman codes (see [RFC1951](#) specification) of the input data stream of the `IppLZ77Pair` type that is produced by the function `ippsEncodeLZ77`.

The function `ippsEncodeLZ77FixedHuff` writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter `flush`.

If the function returns one of the `ippStsNoErr`, `ippStsDstSizeLessExpected`, or `ippStsStreamEnd` statuses, the values of the pointers to the source and destination data buffers and to their lengths are changed and their new values are returned.

The function uses the LZ77 encoding state structure `pLZ77State` that must be previously initialized by the function `ippsEncodeLZ77InitAlloc` or `ippsEncodeLZ77Init`.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.
<code>ippStsStreamEnd</code>	Indicates a warning that the end of the stream occurs. The function returns this status if the parameter <i>flush</i> is set to <code>IppLZ77FinishFlush</code> .

## EncodeLZ77DynamicHuff

*Performs dynamic Huffman encoding.*

---

### Syntax

```
IppStatus ippEncodeLZ77DynamicHuff_8u(IppLZ77Pair** ppSrc, int* pSrcLen,
Ipp8u** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>flush</i>	Specifies the encoding mode for data blocks (see <a href="#">flush Parameter</a> ).
<i>pLZ77State</i>	Pointer to the LZ77 encoding state structure.

### Description

The function `ippEncodeLZ77DynamicHuff` is declared in the `ippdc.h` file. This function performs dynamic Huffman encoding (see [RFC1951](#) specification) of the input data stream of the `IppLZ77Pair` type that is produced by the function `ippEncodeLZ77`.

The function `ippEncodeLZ77DynamicHuff` writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter *flush*.

If the function returns one of the `ippStsNoErr`, `ippStsDstSizeLessExpected`, or `ippStsStreamEnd` statuses, the values of the pointers to the source and destination data buffers and to their lengths are changed and their new values are returned.

The function uses the LZ77 encoding state structure `pLZ77State` that must be previously initialized by the function `ippsEncodeLZ77InitAlloc` or `ippsEncodeLZ77Init`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>DstLen</code> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.
<code>ippStsStreamEnd</code>	Indicates a warning that the end of the stream occurs. The function returns this status if the parameter <code>flush</code> is set to <code>IppLZ77FinishFlush</code> .

## EncodeLZ77StoredBlock

*Stores the data block without compression.*

---

### Syntax

```
IppStatus ippsEncodeLZ77StoredBlock_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

### Parameters

<code>ppSrc</code>	Double pointer to the source buffer.
<code>pSrcLen</code>	Pointer to the length of the source buffer.
<code>ppDst</code>	Double pointer to the destination buffer.
<code>pDstLen</code>	Pointer to the length of the destination buffer.
<code>flush</code>	Specifies the encoding mode for data blocks (see <a href="#">flush Parameter</a> ).
<code>pLZ77State</code>	Pointer to the LZ77 encoding state structure.

## Description

The `ippsEncodeLZ77StoredBlock` function is declared in the `ippdc.h` file. This function copies source data *ppSrc* data to the destination buffer *ppDst* without Huffman encoding.

The function `ippsEncodeLZ77StoredBlock` writes the marker of the end of the block to the output stream and aligns the output data in accordance with the specified parameter *flush*.

The function uses the LZ77 encoding state structure *pLZ77State* that must be previously initialized by the function `ippsEncodeLZ77InitAlloc` or `ippsEncodeLZ77Init`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

## EncodeLZ77Flush

*Writes the checksum and total length of the input data to the end of the stream.*

---

### Syntax

```
IppStatus ippsEncodeLZ77Flush_8u (Ipp8u** ppDst, int* pDstLen,
IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of destination buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.

### Description

The `ippsEncodeLZ77Flush` function is declared in the `ippdc.h` file. This function writes the computed checksum and total length of the input data stream to the end of the output stream that is encoded by the function `ippsEncodeLZ77SelectHuffMode`. This function is used to ensure compatibility with the [RFC1950](#), [RFC1951](#), and [RFC1952](#) specifications.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the size of the destination buffer is insufficient for completing the operation.

## EncodeLZ77GetPairs

*Retrieves pair data from the LZ77 encoding state structure.*

---

### Syntax

```
IppStatus ippEncodeLZ77GetPairs_8u (IppLZ77Pair** ppPairs, int* pPairsInd,
int* pPairsLen, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>ppPairs</i>	Double pointer to the pair buffer.
<i>pPairsInd</i>	Pointer to the current index in the pair buffer.
<i>pPairsLen</i>	Pointer to the length of the pair buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.

### Description

The `ippEncodeLZ77GetPairs` function is declared in the `ippdc.h` file. This function returns the following pointers: *ppPairs* to the pair buffer, *pPairsLen* to the length of the pair buffer, and *pPairsInd* to the current index. These pointers are retrieved from the initialized LZ77 encoding state structure *pLZ77State*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>ppPairs</i> pointer is <code>NULL</code> .

## EncodeLZ77SetPairs

Sets pair data in the LZ77 encoding state structure.

### Syntax

```
IppStatus ippEncodeLZ77SetPairs_8u (IppLZ77Pair* pPairs, int pairsInd, int pairsLen, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>pPairs</i>	Pointer to the pair buffer.
<i>pairsInd</i>	Current index in the pair buffer.
<i>pairsLen</i>	Length of the pair buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.

### Description

The `ippEncodeLZ77SetPairs` function is declared in the `ippdc.h` file. This function sets the pointer to the pair buffer *pPairs*, the length of the pair buffer *pairsLen*, and the current index *pairsInd* in the initialized LZ77 encoding state structure *pLZ77State*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>pPairs</i> pointer is NULL.

## EncodeLZ77GetStatus

Reads the deflate status value from the LZ77 encoding state structure.

### Syntax

```
IppStatus ippEncodeLZ77GetStatus_8u (IppLZ77VLCDeflateStatus* pDeflateStatus, IppLZ77_8u* pLZ77State);
```

### Parameters

<i>pDeflateStatus</i>	Pointer to the deflate status parameter (see <a href="#">deflateStatus Parameter</a> ).
-----------------------	---

*pLZ77State* Pointer to the initialized LZ77 encoding state structure.

## Description

The `ippsEncodeLZ77GetStatus` function is declared in the `ippdc.h` file. This function reads the current encoding status value *pDeflateStatus* from the initialized LZ77 state structure *pLZ77State*.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>pDeflateStatus</i> pointer is NULL.

## EncodeLZ77SetStatus

*Sets the deflate status to the desired value in the LZ77 encoding state structure.*

---

## Syntax

```
IppStatus ippsEncodeLZ77SetStatus_8u (IppLZ77DeflateStatus deflateStatus,
IppLZ77_8u* pLZ77Stat);
```

## Parameters

<i>pLZ77State</i>	Pointer to the initialized LZ77 encoding state structure.
<i>pDeflateStatus</i>	Pointer to the deflate status parameter (see <a href="#">deflateStatus Parameter</a> ).

## Description

The `ippsEncodeLZ77SetStatus` function is declared in the `ippdc.h` file. This function sets the parameter *deflateStatus* to the specified value in the initialized LZ77 encoding state structure *pLZ77State*.

This parameter must be specified only if the compatibility with the [RFC1950](#), [RFC1951](#), and [RFC1952](#) specifications is required.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> pointer is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error if the <code>deflateStatus</code> parameter has an illegal value.

## EncodeLZ77Reset

*Resets the LZ77 encoding state structure.*

---

### Syntax

```
IppStatus ippEncodeLZ77Reset_8u (IppLZ77State_8u* pLZ77State);
```

### Parameters

`pLZ77State` Pointer to the LZ77 encoding state structure.

### Description

The function `ippEncodeLZ77Reset` is declared in the `ippdc.h` file. This function resets the LZ77 encoding state structure to the initial state.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pLZ77State</code> pointer is <code>NULL</code> .

## DecodeLZ77Init

*Initializes the LZ77 decoding structure.*

---

### Syntax

```
IppStatus ippDecodeLZ77Init_8u(IppLZ77Chcksm checksum, IppLZ77State_8u* pLZ77State);
```

### Parameters

<code>checksum</code>	Specifies an algorithm to compute the checksum for input data (see <a href="#">checksum Parameter</a> ).
<code>pLZ77State</code>	Pointer to the initialized LZ77 decoding structure.

## Description

The function `ippsDecodeLZ77Init` is declared in the `ippdc.h` file. This function initializes the LZ77 decoding structure in the external buffer whose size must be previously computed by calling the function `ippsDecodeLZ77GetSize`. The function uses different algorithms to compute the checksum for input data in accordance with the value of the parameter `checksum`. The decoding state structure is used by the function `ippsDecodeLZ77`.

Alternatively, this structure can be initialized by the function `ippsDecodeLZ77InitAlloc`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pLZ77State</code> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the <code>checksum</code> parameter has an illiegal value.

## DecodeLZ77GetSize

Computes the size of the LZ77 decoding structure.

### Syntax

```
ippStatus ippsDecodeLZ77GetSize_8u(int* pLZ77StateSize);
```

### Parameters

<code>pLZ77StateSize</code>	Pointer to the size of the LZ77 decoding structure.
-----------------------------	---

### Description

The function `ippsDecodeLZ77GetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the LZ77 decoding state structure. The function must be called prior to the function `ippsDecodeLZ77Init`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pLZ77StateSize</code> is NULL.



## DecodeLZ77InitAlloc

*Allocates memory and initializes the LZ77 decoding structure.*

---

### Syntax

```
IppStatus ippsDecodeLZ77InitAlloc_8u(IppLZ77Chcksm checksum, IppLZ77State_8u** ppLZ77State);
```

### Parameters

<i>checksum</i>	Specifies an algorithm to compute the checksum for input data (see <a href="#">checksum Parameter</a> ).
<i>ppLZ77State</i>	Double pointer to the LZ77 decoding structure.

### Description

The function `ippsDecodeLZ77InitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the decoding state structure *ppLZ77State*. The function uses different algorithms to compute the checksum for input data in accordance with the value of the parameter *checksum*. The decoding state structure is used by the function [ippsDecodeLZ77](#).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>ppLZ77State</i> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the <i>checksum</i> parameter has an illegal value.
<code>ippStsMemAlloc</code>	Indicates an error if memory allocation fails.

## DecodeLZ77

*Performs LZ77 decoding.*

---

### Syntax

```
IppStatus ippsDecodeLZ77_8u(IppLZ77Pair** ppSrc, int* pSrcLen, Ipp8u** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u* pLZ77State);
```

## Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>ppDst</i>	Double pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>flush</i>	Specifies the encoding mode for data blocks.
<i>pLZ77State</i>	Pointer to the LZ77 decoding structure.

## Description

The function `ippsDecodeLZ77` is declared in the `ippdc.h` file. This function performs LZ77 decoding of a sequence of pairs of the `IppLZ77Pair` type. The sliding window technique implemented in this function ensures compatibility with the [RFC1950](#), [RFC1951](#), and [RFC1952](#) specifications.

The function `ippsDecodeLZ77` can be used for decoding data not only in the DEFLATE, ZLIB, GZIP format, but in any other format based on the LZ77 algorithm.

[Code example 13-3](#) shows how to use the `ippsDecodeLZ77` function.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>ppLZ77State</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the destination buffer is full.

## DecodeLZ77GetBlockType

*Determines the type of encoded data block.*

---

### Syntax

```
IppStatus ippsDecodeLZ77GetBlockType_8u(Ipp8u** ppSrc, int* pSrcLen,
IppLZ77HuffMode* pHuffMode, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
--------------	--------------------------------------

<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>pHuffMode</i>	Pointer to the value of the parameter that indicates the encoding mode. Possible values are: IppLZ77UseFixed Encoding with fixed Huffman codes. IppLZ77UseDynamic Encoding with dynamic Huffman codes. IppLZ77UseStored Stored block without Huffman coding.
<i>pLZ77State</i>	Pointer to the LZ77 decoding state structure.

## Description

The function `ippsDecodeLZ77GetBlockType` is declared in the `ippdc.h` file. This function analyzes the source data *ppSrc* and returns the parameter *pHuffMode* which value indicates what Huffman decoding mode must be used for decoding the source bit stream. In accordance with this value the one of the functions [ippsDecodeLZ77FixedHuff](#), [ippsDecodeLZ77DynamicHuff](#), or [ippsDecodeLZ77StoredBlock](#) must be used.



**NOTE.** If you use only one specified encoding mode, then type of the input data for decoding is known, and this function is not necessary.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>ppLZ77State</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>SrcLen</i> is less than or equal to 0.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning that the source buffer is insufficient (internal bit stream and source buffer do not contain enough bits to decode the type of block).

## DecodeLZ77FixedHuff

*Performs fixed Huffman decoding.*

---

### Syntax

```
IppStatus ippDecodeLZ77FixedHuff_8u(Ipp8u** ppSrc, int* pSrcLen,
IppLZ77Pair** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u*
pLZ77State);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source vector.
<i>ppDst</i>	Double pointer to the destination vector.
<i>pDstLen</i>	Pointer to the length of the destination vector.
<i>flush</i>	Specifies the decoding mode for data blocks.
<i>pLZ77State</i>	Pointer to the LZ77 decoding structure.

### Description

The function `ippDecodeLZ77FixedHuff` is declared in the `ippdc.h` file. This function performs decoding of the source data *ppSrc* with the fixed Huffman codes (see [RFC1951](#) specification). The source data must be compatible with the [RFC1950](#), [RFC1951](#), and [RFC1952](#) specifications.

Exit from the `ippDecodeLZ77FixedHuff` function occurs when:

- the input stream has ended, but the end-of-block marker is not decoded and the function returns the `ippStsSrcSizeLessExpected` status.
- the input stream has not ended, but the destination buffer is full and the function returns the `ippStsDstSizeLessExpected` status.
- the input stream has ended, the end-of-stream marker is decoded and the function returns the `ippStsNoErr` status.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.

`ippStsSrcSizeLessExpected` Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).

`ippStsDstSizeLessExpected` Indicates a warning that the destination buffer is full.

`ippStsStreamEnd` Indicates a warning that the stream ends. This warning returns only if the *flush* parameter is set to `IppLZ77FinishFlush`.

## DecodeLZ77DynamicHuff

*Performs dynamic Huffman decoding.*

### Syntax

```
IppStatus ippDecodeLZ77DynamicHuff_8u(Ipp8u** ppSrc, int* pSrcLen,
IppLZ77Pair** ppDst, int* pDstLen, IppLZ77Flush flush, IppLZ77State_8u*
pLZ77State);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source vector.
<i>ppDst</i>	Double pointer to the destination vector.
<i>pDstLen</i>	Pointer to the length of the destination vector.
<i>flush</i>	Specifies the decoding mode for data blocks.
<i>pLZ77State</i>	Pointer to the LZ77 decoding structure.

### Description

The function `ippDecodeLZ77DynamicHuff` is declared in the `ippdc.h` file. This function performs decoding of the source data *ppSrc* with the dynamic Huffman codes (see [RFC1951](#) specification).

Exit from the `ippDecodeLZ77FixedHuff` function occurs when:

- the input stream has ended, but the end-of-block marker is not decoded and the function returns the `ippStsSrcSizeLessExpected` status.
- the input stream has not ended, but the destination buffer is full and the function returns the `ippStsDstSizeLessExpected` status.
- the input stream has ended, the end-of-stream marker is decoded and the function returns the `ippStsNoErr` status.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the destination buffer is full.
<code>ippStsStreamEnd</code>	Indicates a warning that the stream ends. This warning returns only if the <i>flush</i> parameter is set to <i>IppLZ77FinishFlush</i> .

## DecodeLZ77StoredBlock

*Performs decoding of the block stored without compression.*

---

### Syntax

```
IppStatus ippDecodeLZ77StoredBlock_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u** ppDst, int* pDstLen, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source vector.
<i>ppDst</i>	Double pointer to the destination vector.
<i>pDstLen</i>	Pointer to the length of the destination vector.
<i>pLZ77State</i>	Pointer to the LZ77 decoding structure.

### Description

The function `ippDecodeLZ77StoredBlock` is declared in the `ippdc.h` file. This function performs decoding of the source data block *ppSrc* that is stored without Huffman coding.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>DstLen</i> is less than or equal to 0.

---

<code>ippStsWrongBlockType</code>	Indicates a warning that the type of the block is not the “stored without compression type”.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning that the source buffer is less than expected (the end-of-block marker is not encoded).
<code>ippStsDstSizeLessExpected</code>	Indicates a warning that the destination buffer is full.
<code>ippStsStreamEnd</code>	Indicates a warning that the stream ends. This warning returns only if the <i>flush</i> parameter is set to <code>IppLZ77FinishFlush</code> .

## DecodeLZ77GetPairs

*Retrieves pair data from the LZ77 decoding state structure.*

---

### Syntax

```
IppStatus ippDecodeLZ77GetPairs_8u(IppLZ77Pair** ppPairs, int* pPairsInd,
int* pPairsLen, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>ppPairs</i>	Double pointer to the pair buffer.
<i>pPairsInd</i>	Pointer to the current index in the pair buffer.
<i>pPairsLen</i>	Pointer to the length of the pair buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 decoding state structure.

### Description

The `ippDecodeLZ77GetPairs` function is declared in the `ippdc.h` file. This function returns the following pointers: *ppPairs* to the pair buffer, *pPairsLen* to the length of the pair buffer, and *pPairsInd* to the current index. These pointers are retrieved from the initialized LZ77 decoding state structure *pLZ77State*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>ppPairs</i> pointer is <code>NULL</code> .

## DecodeLZ77SetPairs

*Sets pair data in the LZ77 decoding state structure.*

---

### Syntax

```
IppStatus ippsDecodeLZ77SetPairs_8u(IppLZ77Pair* pPairs, int pairsInd, int pairsLen, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>pPairs</i>	Pointer to the pair buffer.
<i>pairsInd</i>	Current index in the pair buffer.
<i>pairsLen</i>	Length of the pair buffer.
<i>pLZ77State</i>	Pointer to the initialized LZ77 decoding state structure.

### Description

The `ippsDecodeLZ77SetPairs` function is declared in the `ippdc.h` file. This function sets the pointer to the pair buffer *pPairs*, the length of the pair buffer *pairsLen*, and the current index *pairsInd* in the initialized LZ77 decoding state structure *pLZ77State*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pLZ77State</i> or <i>pPairs</i> pointer is NULL.

## DecodeLZ77GetStatus

*Reads the inflate status value from the LZ77 decoding state structure.*

---

### Syntax

```
IppStatus ippsDecodeLZ77GetStatus_8u(IppLZ77InflateStatus* pInflateStatus, IppLZ77State_8u* pLZ77State);
```

### Parameters

<i>pInflateStatus</i>	Pointer to the inflate status parameter (see <a href="#">inflateStatus Parameter</a> ).
-----------------------	---



*pLZ77State* Pointer to the LZ77 decoding structure.

### Description

The `ippsDecodeLZ77GetStatus` function is declared in the `ippdc.h` file. This function reads the current decoding status value *pInflateStatus* from the initialized LZ77 state structure *pLZ77State*.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error if one of the specified pointers is `NULL`.

## DecodeLZ77SetStatus

*Sets the inflate status to the desired value in the LZ77 decoding state structure.*

---

### Syntax

```
IppStatus ippsDecodeLZ77SetStatus_8u(IppLZ77InflateStatus inflateStatus,  
IppLZ77State_8u* pLZ77State);
```

### Parameters

*pLZ77State* Pointer to the initialized LZ77 encoding state structure.  
*pInflateStatus* Pointer to the inflate status parameter (see [inflateStatus Parameter](#)).

### Description

The `ippsDecodeLZ77SetStatus` function is declared in the `ippdc.h` file. This function sets the parameter *inflateStatus* to the specified value in the initialized LZ77 decoding state structure *pLZ77State*.

### Return Values

`ippStsNoErr` Indicates no error.  
`ippStsNullPtrErr` Indicates an error if the *pLZ77State* pointer is `NULL`.  
`ippStsBadArgErr` Indicates an error if the *inflateStatus* parameter has an illegal value.

## DecodeLZ77Reset

*Resets the LZ77 decoding state structure.*

---

### Syntax

```
IppStatus ippDecodeLZ77Reset_8u(IppLZ77State_8u* pLZ77State);
```

### Parameters

*pLZ77State*                      Pointer to the LZ77 encoding state structure.

### Description

The function `ippDecodeLZ77Reset` is declared in the `ippdc.h` file. This function resets the LZ77 decoding state structure to the initial state.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error if the *pLZ77State* pointer is NULL.

## Adler32

*Computes the Adler32 checksum for the source data buffer.*

---

### Syntax

```
IppStatus ippAdler32_8u (const Ipp8u* pSrc, int srcLen, Ipp32u* pAdler32);
```

### Parameters

*pSrc*                                Pointer to the source data buffer.  
*srcLen*                              Number of elements in the source data buffer.  
*pAdler32*                            Pointer to the checksum value.

## Description

The function `ippsAdler32` is declared in the `ippdc.h` file. This function computes the checksum for `srcLen` elements of the source data buffer `pSrc` and stores it in the `pAdler32`. The checksum is computed using the Adler32 algorithm that is a modified version of the Fletcher algorithm [Flet82], [ITU224 ], [RFC1950].

You can use this function to compute the accumulated value of the checksum for multiple buffers in the data stream by specifying as an input parameter the checksum value obtained in the preceding function call.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pSrc</code> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>srcLen</code> is less than or equal to 0.

## CRC32, CRC32C

*Computes the CRC32 checksum for the source data buffer.*

---

## Syntax

```
IppStatus ippsCRC32_8u (const Ipp8u* pSrc, int srcLen, Ipp32u* pCRC32);
IppStatus ippsCRC32C_8u(const Ipp8u* pSrc, Ipp32u srcLen, Ipp32u* pCRC32C);
```

## Parameters

<code>pSrc</code>	Pointer to the source data buffer.
<code>srcLen</code>	Number of elements in the source data buffer.
<code>pCRC32, pCRC32C</code>	Pointer to the checksum value.

## Description

The functions `ippsCRC32` and `ippsCRC32C` are declared in the `ippdc.h` file. These functions compute the checksum for `srcLen` elements of the source data buffer `pSrc` using different algorithms and stores it in the `pCRC32` or `pCRC32C` respectively.

The function `ippsCRC32` uses algorithm described in [Grif87], [Nel92], the function `ippsCRC32C` uses algorithm described in [Cast93].

These functions can be used to compute the accumulated value of the checksum for multiple buffers in the data stream by specifying as an input parameter the checksum value obtained in the preceding function call.

Example 13-3a below shows how to use the function `ippSCRC32C_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if the length of the source vector is less than or equal to 0.

**Example 13-3a Using the `ippsCRC32C_8u` Function**

```

#include <iostream>
#include <iomanip>
#include "ipp.h"
using namespace std;

void crc32c_core( Ipp8u* src, Ipp32u src_len )
{
    Ipp32u crc32c = ~(Ipp32u)0;
    ippsCRC32C_8u( src, src_len, &crc32c );
    ippsSwapBytes_32u_I( &crc32c, 1 );
    cout << "0x" << setbase(16) << ~crc32c << endl;
}

int main()
{
    Ipp8u buff[48] = { 0x01, 0xc0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                      0x14, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00,
                      0x00, 0x00, 0x00, 0x14, 0x00, 0x00, 0x00, 0x18,
                      0x28, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                      0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

    cout << "An iSCSI - SCSI Read (10) Command PDU: ";
    crc32c_core( buff, 48 );

    cout << "32 bytes of zeroes: ";
    for( int i = 0; i < 32; i++ ) buff[i] = 0;
    crc32c_core( buff, 32 );
}

```

```

cout << "32 bytes of ones: ";

for( int i = 0; i < 32; i++ ) buff[i] = 0xff;

crc32c_core( buff, 32 );

cout << "32 bytes of incrementing 00..1f: ";

for( int i = 0; i < 32; i++ ) buff[i] = i;

crc32c_core( buff, 32 );

cout << "32 bytes of decrementing 1f..00: ";

for( int i = 0; i < 32; i++ ) buff[i] = 31 - i;

crc32c_core( buff, 32 );

return 0;

}

```

Output:

```

An iSCSI - SCSI Read (10) Command PDU: 0x563a96d9
32 bytes of zeroes: 0xaa36918a
32 bytes of ones: 0x43aba862
32 bytes of incrementing 00..1f: 0x4e79dd46
32 bytes of decrementing 1f..00: 0x5cdb3f11

```

## DeflateLZ77

*Performs LZ77 encoding according to the specified compression level.*

---

### Syntax

```

IppStatus ippsDeflateLZ77_8u(const Ipp8u** ppSrc, Ipp32u* pSrcLen, Ipp32u*
pSrcIdx, const Ipp8u* pWindow, Ipp32u winSize, Ipp32s* pHashHead, Ipp32s*
pHashPrev, Ipp32u hashSize, IppDeflateFreqTable pLitFreqTable[286],
IppDeflateFreqTable pDistFreqTable[30], Ipp8u* pLitDst, Ipp16u* pDistDst,
Ipp32u* pDstLen, int comprLevel, IppLZ77Flush flush);

```

### Parameters

*ppSrc* Double pointer to the source vector.

---

<i>pSrcLen</i>	Pointer to the length of the source vector.
<i>pSrcIdx</i>	Pointer to the index of the current position in the source vector.
<i>pWindow</i>	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
<i>winSize</i>	Size of the sliding window and the <i>pHashPrev</i> table.
<i>pHashHead</i>	Pointer to the table containing heads of the hash chains.
<i>pHashPrev</i>	Pointer to the table containing indexes to the previous strings with the same hash key.
<i>hashSize</i>	Size of the <i>pHashHead</i> table.
<i>pLitFreqTable</i>	Pointer to the literals/lengths frequency table.
<i>pDistFreqTable</i>	Pointer to the distances frequency table.
<i>pLitDst</i>	Pointer to the destination vector containing literals/lengths.
<i>pDistDst</i>	Pointer to the destination vector containing distances.
<i>pDstLen</i>	Pointer to the length of the destination vectors.
<i>comprLevel</i>	Compression level in range [0..9] in accordance with ZLIB.
<i>flush</i>	Specifies the encoding mode for data blocks (see <a href="#">flush Parameter</a> ).

## Description

The function `ippsDeflateLZ77` is declared in the `ippdc.h` file. This function performs LZ77 encoding of source data *ppSrc* according to the compression level *comprLevel*, which is similar to the ZLIB compression level.

The table *pHashHead* must be initialized with value *-winSize* for correct processing the first bytes of the source vector.

The function parameter *pSrcIdx* returns the index of the current position in the source vector and is used to correlate the current position in the source vector and indexes in the hash tables. After processing each 2GB of source data, this index and hash tables must be normalized (instead of 64K of source data in ZLIB).

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .

## DeflateDictionarySet

Presets the user's dictionary for LZ77 encoding.

### Syntax

```
IppStatus ippDeflateDictionarySet_8u(const Ipp8u* pDictSrc, Ipp32u dictLen,
Ipp32s* pHashHeadDst, Ipp32u hashSize, Ipp32s* pHashPrevDst, Ipp8u*
pWindowDst, Ipp32u winSize, int comprLevel);
```

### Parameters

<i>pDictSrc</i>	Pointer to the user's dictionary.
<i>dictLen</i>	Length of the user's dictionary.
<i>pHashHeadDst</i>	Pointer to the table containing heads of the hash chains.
<i>pHashPrevDst</i>	Pointer to the table containing indexes to the previous strings with the same hash key.
<i>hashSize</i>	Size of the <i>pHashHeadDst</i> table.
<i>pWindowDst</i>	Pointer to the sliding window that is used as the dictionary for LZ77 encoding.
<i>winSize</i>	Size of the sliding window and the elements of the <i>pHashPrevDst</i> table.
<i>comprLevel</i>	Compression level in range [0..9] in accordance with ZLIB.

### Description

The function `ippDeflateDictionarySet` is declared in the `ippdc.h` file. This function presets the user's dictionary for LZ77 encoding.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------



`ippStsNullPtrErr` Indicates an error if one of the specified pointers is `NULL`.

## DeflateHuff

*Performs Huffman encoding .*

---

### Syntax

```
IppStatus ippDeflateHuff_8u(const Ipp8u* pLitSrc, const Ipp16u* pDistSrc,
Ipp32u srcLen, Ipp8u* pCode, Ipp32u* pCodeLenBits, IppDeflateHuffCode*
pLitHuffCodes[256], IppDeflateHuffCode* pDistHuffCodes[30], Ipp8u* pDst,
Ipp32u* pDstIdx);
```

### Parameters

<i>pLitSrc</i>	Pointer to the literals/lengths source vector.
<i>pDistSrc</i>	Pointer to the distances source vector.
<i>srcLen</i>	Length of the source vectors.
<i>pCode</i>	Pointer to the bit buffer.
<i>pCodeLenBits</i>	Pointer to the number of valid bits in the bit buffer.
<i>pLitHuffCodes</i>	Pointer to the literals/lengths Huffman codes.
<i>pDistHuffCodes</i>	Pointer to the distances Huffman codes.
<i>pDst</i>	Pointer to the destination vector.
<i>pDstIdx</i>	Pointer to the index in the destination vector.

### Description

The function `ippDeflateHuff` is declared in the `ippdc.h` file. This function performs Huffman encoding of source data.

The function parameter *pDstIdx* returns the index of the current position in the destination vector: zlib uses the intermediate buffer for the Huffman encoding and we need to know the indexes of the first (input parameter) and the last (output parameter) symbols, which are written by the function.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .

## InflateBuildHuffTable

*Builds the Huffman code table for compressed block in the "deflate" format.*

### Syntax

```
IppStatus ippInflateBuildHuffTable(const Ipp16u* pCodeLens, unsigned int
nLitCodeLens, unsigned int nDistCodeLens, IppInflateState* pIppInflateState);
```

### Parameters

<i>pCodeLens</i>	Pointer to the common array with lengths of the Huffman codes for literals/lengths and distances.
<i>nLitCodeLens</i>	Number of lengths of the Huffman codes for literals/lengths.
<i>nDistCodeLens</i>	Number of lengths of the Huffman codes for distances.
<i>pIppInflateState</i>	Pointer to the structure with the parameters of decoding.

### Description

The function `ippInflateBuildHuffTable` is declared in the `ippdc.h` file. This function builds tables of Huffman codes for literals/lengths and distances to decode a block compressed with use of the dynamic Huffman codes in accordance with the "deflate" format [RFC1951].

The structure *IppInflateState* contains the following fields:

<i>pWindow</i>	Pointer to the sliding window (the dictionary for the LZ77 algorithm).
<i>winSize</i>	Size of the sliding window in the range [256, 32768].
<i>tableType</i>	Type of the Huffman code tables. For dynamic Huffman code it is greater than 0, for fixed Huffman codes is equal to 0.
<i>tableBufferSize</i>	Size of the buffer containing the tables. Its value is 8192 - <code>sizeof(IppInflateState)</code> . (8192 = ENOUGH* <code>sizeof(code)</code> ; ENOUGH is defined in ZLIB and is equal to 2048, <code>sizeof(code)=4</code> .)

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .

<code>ippStsSizeErr</code>	Indicates an error if <i>nLitCodeLens</i> is greater than 286 or <i>nDistCodeLens</i> is greater than 30.
<code>ippStsSrcDataErr</code>	Indicates an error if a not valid literal/length and distance set occurs in the common lengths array.

## Inflate

*Decodes data in the "deflate" format.*

---

### Syntax

```
IppStatus ippInflate_8u(Ipp8u** ppSrc, unsigned int* pSrcLen, Ipp32u* pCode,
unsigned int* pCodeLenBits, unsigned int winIdx, Ipp8u** ppDst, unsigned
int* pDstLen, unsigned int dstIdx, IppInflateMode* pMode, IppInflateState*
pIppInflateState);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source vector.
<i>pSrcLen</i>	Pointer to the length of source vector.
<i>pCode</i>	Pointer to the bit buffer.
<i>pCodeLenBits</i>	Number of valid bits in the bit buffer.
<i>winIdx</i>	Index of the start position of the sliding window.
<i>ppDst</i>	Double pointer to the destination vector.
<i>pDstLen</i>	Pointer to the length of destination vector.
<i>dstIdx</i>	Index of the current position in the destination vector.
<i>pMode</i>	Pointer to the current decode mode. Possible values are: <code>ippTYPE</code> - block decoding is completed; <code>ippLEN</code> - decoding from the beginning of the sequence; <code>ippLENEXT</code> - extra bits are required to decode the sequence.
<i>pIppInflateState</i>	Pointer to the structure that contains parameters of decoding.

## Description

The function `ippsInflate` is declared in the `ippdc.h` file. This function decodes the data encoded in the "deflate" format [RFC1951] in accordance with the parameters set in the structure `pIppInflateState`. If the data is compressed using dynamic Huffman codes, the Huffman code tables must be built by the function `ippsInflateBuildHuffTable` beforehand. If the data is compressed using the fixed Huffman codes, the field `tableType` in the `pIppInflateState` must be set to 0, and code tables are not required to be built at all.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>pCodeLenBits</code> is greater than 32, or if <code>winIdx</code> is greater than <code>pIppInflateState-&gt;winSize</code> , or if <code>dstIdx</code> is greater than <code>pDstLen</code> .
<code>ippStsSrcDataErr</code>	Indicates an error if a not valid literal/length and distance set occurs during decoding.

## Example of Using Intel IPP ZLIB Coding Functions

The following code example 13-3 demonstrates how the model of a compressor can be implemented using the Intel IPP ZLIB functions `ippsEncodeLZ77_8u` and `ippsDecodeLZ77_8u` and an arbitrary bit encoding/decoding methods for the pairs.

### Example 13-3 Using the `ippsEncodeLZ77` and `ippsDecodeLZ77` functions

```
#include <stdio.h>
#include <stdlib.h>
#include "ippdc.h"
#define BUFLen 1000

.....

int

main(int argc, char* argv[])
```

---

```

{
    int len;
    int lenr;
    Ipp8u* pSrc;
    int srcLen;
    IppLZ77Pair* pDst;
    int dstLen;

    IppLZ77Pair* pSrcr;
    int srcLenr;
    Ipp8u* pDstr;
    int dstLenr;

    IppStatus status;
    IppLZ77State_8u* pLZ77State;
    .....
    /* LZ77 encoding */
    status = ippsEncodeLZ77InitAlloc_8u(IppLZ77FastCompr, IppLZ77CRC32,
                                        &pLZ77State);

    /* Reading the srcLen bytes of input data from the input data file to pSrc */
    .....
    len = dstLen;

    status = ippsEncodeLZ77_8u(&pSrc, &srcLen, &pDst, &dstLen,
                              IppLZ77FinishFlush, pLZ77State);

    /* Here we have the array of Pairs in pDst of (len - dstLen) */

```

```
/* In this point the arbitrary bit coding method can be used, */
/* for example LZSS algorithm. */
.....
.....
/* Writing the compressed data to the output file */
ippsLZ77Free_8u(pLZ77State);
/* LZ77 decoding */
status = ippsDecodeLZ77InitAlloc_8u(IppLZ77CRC32, &pLZ77State);

/* Reading the compressed data from the file */
.....
/* On this point the bit decoding method, corresponding to the above */
/* selected bit encoding method (LZSS for example) must be used */
.....
.....
/* After bit decoding the pairs restored placed to the pSrcr of srcLenr */
/* for further performing of LZ77 decoding */
lenr = dstLenr;

status = ippsDecodeLZ77_8u(&pSrcr, &srcLenr, &pDstr, &dstLenr,
                          IppLZ77FinishFlush, pLZ77State);

/* Writing the (lenr - dstLenr) bytes of decompressed data to the output file */

ippsLZ77Free_8u(pLZ77State);
```

```
    return 0;
} /* main */
```

LZO Compression Functions

This section describes Intel IPP data compression functions, that implement the LZO (Lempel-Ziv-Oberhumer) compressed data format. This format and algorithm use 64KB compression dictionary and do not require additional memory for decompression. (See original code of the LZO library at <http://www.oberhumer.com>.)

Special Parameters

The LZO coding initialization functions have a special parameter *method*. This parameter specifies level of parallelization and generic LZO compatibility to be used in the LZO encoding. Table 13-10 below lists possible values of the *method* parameter and their meanings.

**Table 13-10. Parameter *method* for the LZO Compression Functions**

Value	Descriptions
IppLZO1XST	The compression and decompression are performed sequentially in a single-thread mode with full binary compatibility with generic LZO libraries and applications
IppLZO1XMT	The compression and decompression are performed in parallel (multi-threaded mode), it is more fast, but not compatible with the generic LZO.

EncodeLZOGetSize

*Calculates the size of LZO encoding structure.*

Syntax

```
ippStatus ippsEncodeLZOGetSize(IppLZOMethod method, Ipp32u maxInputLen,
Ipp32u* pSize);
```

Parameters

*method* Specifies required LZO compression method, possible values are listed in [Table "method Parameter"](#)).

<i>maxInputLen</i>	Specifies maximum length of input data buffer during compression operations. Not required for <code>IppLZO1XST</code> compression method.
<i>pSize</i>	Pointer to the variable, receiving the size of LZO encoding structure.

## Description

The function `ippsEncodeLZOGetSize` is declared in the `ippdc.h` file. This function calculates the size of the memory buffer that must be allocated for the LZO encoding structure.

For the single-thread compression (*method* = `IppLZO1XST`) the size of the structure is fixed, and the value of the *maxInputLen* parameter is ignored, for example, it can be set to 0.

For the multi-threaded compression (*method* = `IppLZO1XMT`) *maxInputLen* parameter is important and affects the size of the structure. If it is set to 0, then each compression operation starts with memory allocation for internal buffers and ends with memory freeing. This significantly decreases the performance of compression/decompression.

Code [example 13-4](#) shows how the Intel IPP functions for the LZO compression can be used.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pSize</i> is NULL.
<code>ippStsBadArgErr</code>	Indicates an error if the parameter <i>method</i> has an illegal value.

## EncodeLZOInit

*Initializes LZO encoding structure.*

---

### Syntax

```
IppStatus ippsEncodeLZOInit_8u(IppLZOMethod method, Ipp32u maxInputLen,
IppLZOState_8u* pLZOState);
```

### Parameters

<i>method</i>	Specifies required LZO compression method, possible values are listed in <a href="#">Table "method Parameter"</a> .
---------------	---



<i>maxInputLen</i>	Specifies maximum length of input data buffer during compression operations. Not required for <code>IppLZO1XST</code> compression method.
<i>pLZOState</i>	Pointer to the LZO encoding structure.

### Description

The function `ippsEncodeLZOInit` is declared in the `ippdc.h` file.

This function initializes the LZO encoding structure in the external buffer. Its size must be calculated by calling the function `ippsEncodeLZOGetSize` beforehand.

The parameter *method* must be the same for both functions.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pLZOState</i> is <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error if the parameter <i>method</i> has an illegal value.

## EncodeLZO

*Compresses input data, returns the length of the compressed data.*

---

### Syntax

```
IppStatus ippsEncodeLZO_8u (const Ipp8u* pSrc, Ipp32u srcLen, Ipp8u* pDst,
Ipp32u* pDstLen, IppLZOState_8u* pLZOState);
```

### Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Length of the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>pLZOState</i>	Pointer to the LZO state structure.

### Description

The function `ippsEncodeLZO` is declared in the `ippdc.h` file.

This function performs compression of the source data *pSrc* according to the method specified in the LZO state structure *pLZOState*. It must be previously initialized by the function [ippsEncodeLZOInit](#).

Compressed data are stored in the *pDst*, the pointer *pDstLen* points to the number of elements in this buffer.

Code [example 13-4](#) shows how the Intel IPP functions for the LZO compression can be used.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .

## DecodeLZO

*Decompresses input data, returns the length of the decompressed data.*

---

### Syntax

```
IppStatus ippsDecodeLZO_8u (Ipp8u* pSrc, Ipp32u srcLen, Ipp8u* pDst, Ipp32u* pDstLen);
```

### Parameters

<i>pSrc</i>	Pointer to the source data buffer.
<i>srcLen</i>	Number of elements in the source data buffer.
<i>pDst</i>	Pointer to the destination data buffer.
<i>pDstLen</i>	Pointer to the variable with the number of elements in the destination data buffer.

### Description

The function `ippsDecodeLZO` is declared in the `ippdc.h` file. The function decompresses the source (compressed) data according to the compressed data format. This function can decompress both single-thread and multi-threaded data. Note that the maximum performance can be obtained only in multi-threaded decompression of data compressed in the multi-threaded mode.

Code [example 13-4](#) shows how the Intel IPP functions for the LZO compression can be used.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsLzoBrokenStream</code>	Compressed data is not an LZO compressed data.
<code>ippStsDstSizeLessExpected</code>	Destination buffer is too small to store decompressed data.

## DecodeLZOSafe

*Decompresses input data with constantly checking integrity of output.*

---

### Syntax

```
IppStatus ippDecodeLZOSafe_8u(Ipp8u* pSrc, Ipp32u srcLen, Ipp8u* pDst,
Ipp32u* pDstLen);
```

### Parameters

<code>pSrc</code>	Pointer to the source data buffer.
<code>srcLen</code>	Number of elements in the source data buffer.
<code>pDst</code>	Pointer to the destination data buffer.
<code>pDstLen</code>	Pointer to the variable with the number of elements in the destination data buffer.

### Description

The function `ippDecodeLZOSafe` is declared in the `ippdc.h` file.

This function is a version of the function `ippDecodeLZO` - it additionally checks the integrity of the destination data buffer, that is checks the buffer boundary limits. This function works slower, it can be used in doubtful cases when the compressed data integrity is not guaranteed, for example, decoding data received via non-reliable communication lines.

Code example 13-4 below shows how the Intel IPP functions for the LZO compression can be used.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .

`ippStsLzoBrokenStreamErr` Indicates an error if compressed data is not valid - not an LZO compressed data.

`ippStsDstSizeLessExpected` Destination buffer is too small to store decompressed data.

**Example 13-4. Using the LZO Compression Functions**

```
/* Simple example of file compression using IPP LZO functions */

#include <stdio.h>
#include "ippdc.h"
#include "ipps.h"
#define BUFSIZE 1024

void CompressFile(const char* pInFileName, const char* pOutFileName)
{
    FILE *pIn, *pOut;
    IppLZOState_8u *pLZOState;
    Ipp8u src[BUFSIZE];

    /* For uncompressible data the size of output will be bigger */
    Ipp8u dst[BUFSIZE + BUFSIZE/10];
    Ipp32u srcLen, dstLen, lzoSize;

    pIn = fopen(pInFileName, "rb");
    pOut = fopen(pOutFileName, "wb");
    ippsEncodeLZOGetSize(IppLZO1XST, BUFSIZE, &lzoSize);
    pLZOState = (IppLZOState_8u*)ippsMalloc_8u(lzoSize);
    ippsEncodeLZOInit_8u(IppLZO1XST, BUFSIZE, pLZOState);
    while ((srcLen = (Ipp32u)fread(src, 1, BUFSIZE, pIn)) > 0) {
        ippsEncodeLZO_8u(src, srcLen, dst, &dstLen, pLZOState);

        fwrite(&srcLen, 1, sizeof(srcLen), pOut);
        fwrite(&dstLen, 1, sizeof(dstLen), pOut);

        fwrite(dst, 1, dstLen, pOut);
    }
    fclose(pIn);
    fclose(pOut);
}
```

```

/* Example of using of DecodeLZO function to decompress the file */
void DecompressFile(const char* pInFileName, const char* pOutFileName)
{
    FILE *pIn, *pOut;
    size_t allocSizeSrc = 0;
    size_t allocSizeDst = 0;
    Ipp32u srcLen, dstLen;
    Ipp8u *pSrc, *pDst;
    pIn = fopen(pInFileName, "rb");
    pOut = fopen(pOutFileName, "wb");
    while (1) {
        if (fread(&dstLen, 1, sizeof(dstLen), pIn) != sizeof(dstLen))
            break;

        fread(&srcLen, 1, sizeof(srcLen), pIn);
        if (srcLen > allocSizeSrc) {
            if (allocSizeSrc > 0)
                ippsFree(pSrc);
            pSrc = ippsMalloc_8u(allocSizeSrc = srcLen);
        }
        if (dstLen > allocSizeDst) {
            if (allocSizeDst > 0)

```

```
        ippsFree(pDst);

    pDst = ippsMalloc_8u(allocSizeDst = dstLen);
}

fread(pSrc, 1, srcLen, pIn);
ippsDecodeLZO_8u(pSrc, srcLen, pDst, &dstLen);
fwrite(pDst, 1, dstLen, pOut);
}

fclose(pIn);
fclose(pOut);
}
```

## BWT-Based Compression Functions

This section describes the Intel IPP functions that support composed algorithms based on the Burrows-Wheeler transform (BWT). The full list of these functions is given in [Table 13-3](#).

### Burrows-Wheeler Transform

Burrows-Wheeler Transform (BWT) does not compress data, but it simplifies the structure of input data and makes more effective further compression. One of the distinctive feature of this method is operation on the block of data (as a rule of size 512kB - 2 mB). The main idea of this method is block sorting which groups symbols with a similar context. Let us consider how BWT works on the input data block 'abracadabra'. The first step is to create a matrix containing all its possible cyclic permutations. The first row is input string, the second is created by shifting it to the left by one symbol and so on:

```
abracadabra
bracadabraa
racadabraab
acadabraabr
cadabraabra
adabraabrac
dabraabraca
abraabracad
braabracada
raabracadab
aabracadabr
```

Then all rows are sorted in accordance with the lexicographic order:

```

0 aabracadabr
1 abraabracad
2 abracadabra
3 acadabraabr
4 adabraabrac
5 braabracada
6 bracadabraa
7 cadabraabra
8 dabraabraca
9 raabracadab
10 racadabraab

```

The last step is to write out the last column and the index of the input string: `rdarcaaaabb`,  
2 - this is a result of the forward BWT transform.

Inverse BTW is performed as follows:

elements of the input string are numbered in ascending order

```

0 r
1 d
2 a
3 r
4 c
5 a
6 a
7 a
8 a
9 b
10 b

```

and sorted in accordance with the lexicographic order:

```

2 a
5 a
6 a
7 a
8 a
9 b
10 b
4 c
1 d
0 r
3 r

```



This index array is a vector of the inverse transform (*Inv*), the further reconstruction of the string is performed in the following manner:

```
src[] = "rdarcaaaabb";  
Inv[] = {2,5,6,7,8,9,10,4,1,0,3};  
  
index = 2; // index  
of the initial string is known from the forward BWT  
  
for( i = 0; i <  
len; i++ ) {  
    index = Inv[index];  
    dst[i] = src[index];  
}
```

## BWTFwdGetSize

*Computes the size of the external buffer for the forward BWT transform.*

---

### Syntax

```
IppStatus ippsBWTFwdGetSize_8u(int wndSize, int* pBWTFwdBuffSize);
```

### Parameters

<i>wndSize</i>	Window size for BWT transform.
<i>pBWTFwdStateSize</i>	Pointer to the computed size of the additional buffer.

### Description

The function `ippsBWTFwdGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required by the function `ippsBWTFwd` for the forward BWT transform.

Code [example 13-5](#) shows how to use the function `ippsBWTFwdGetSize_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pBWTFwdBuffSize</i> pointer is NULL.

## BWTFwd

*Performs the forward BWT transform.*

---

### Syntax

```
IppStatus ippSBWTFwd_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int* pIndex,
Ipp8u* pBWTFwdBuff);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the source and destination vectors.
<i>pIndex</i>	Pointer to the index of first position for the forward BWT transform.
<i>pBWTFwdBuff</i>	Pointer to the additional buffer.

### Description

The function `ippSBWTFwd` is declared in the `ippdc.h` file. This function performs the forward BWT transform of *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTFwdBuff*. The size of this buffer must be computed by calling the function `ippSBWTFwdGetSize` beforehand.

Code [example 13-5](#) shows how to use the function `ippSBWTFwd_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

## BWTInvGetSize

*Computes the size of the external buffer for the inverse BWT transform.*

---

### Syntax

```
IppStatus ippsBWTInvGetSize_8u(int wndSize, int* pBWTInvBuffSize);
```

### Parameters

<i>wndSize</i>	Window size for BWT transform.
<i>pBWTInvBuffSize</i>	Pointer to the computed size of the additional buffer.

### Description

The function `ippsBWTInvGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required by the function [ippsBWTInv](#) for the inverse BWT transform.

Code [example 13-5](#) shows how to use the function `ippsBWTInvGetSize_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pBWTInvBuffSize</i> pointer is NULL.

## BWTInv

*Performs the inverse BWT transform.*

---

### Syntax

```
IppStatus ippsBWTInv_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len, int index,  
Ipp8u* pBWTInvBuff);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the source and destination vectors.

<i>index</i>	Index of first position for the inverse BWT transform.
<i>pBWTInvBuff</i>	Pointer to the additional buffer.

## Description

The function `ippsBWTInv` is declared in the `ippdc.h` file. This function performs the inverse BWT transform of *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTInvBuff*. The size of this buffer must be computed by calling the function `ippsBWTInvGetSize` beforehand.

Example 13-5 below shows how to use the function `ippsBWTInv_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

## Example 13-5 Performing the Burrows-Wheeler Transform

```
void func_BWT()
{
    int wndSize = 8;
    int pBWTfwdBuffSize;
    int pBWTInvBuffSize;

    Ipp8u pSrc[] = "baadeffg";
    int len = 8;
    int pIndex;

    Ipp8u* pDst = ippsMalloc_8u(len);
    Ipp8u* pDstInv = ippsMalloc_8u(len);
    ippsBWTfwdGetSize_8u(wndSize, &pBWTfwdBuffSize);
    Ipp8u* pBWTfwdBuff = ippsMalloc_8u(pBWTfwdBuffSize);
    ippsBWTfwd_8u(pSrc, pDst, len, &pIndex, pBWTfwdBuff);
}
```

```

    ippsBWtInvGetSize_8u( wndSize, &pBWtInvBuffSize);

    Ipp8u* pBWtInvBuff = ippsMalloc_8u(pBWtInvBuffSize);

    ippsBWtInv_8u(pDst, pDstInv, len, pIndex, pBWtInvBuff);
}

Result:

pDst ->      "bagadef"
pDstInv ->    "baadefg"

```

## BWTGetSize\_SmalBlock

*Computes the size of the external buffer for the BWT transforms for small data block.*

---

### Syntax

```
IppStatus ippsBWTGetSize_SmallBlock_8u(int wndSize, int* pBuffSize);
```

### Parameters

<i>wndSize</i>	Window size for BWT transform.
<i>pBuffSize</i>	Pointer to the computed size of the additional buffer.

### Description

The function `ippsBWTGetSize_SmallBlock` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the external buffer that is required for the forward and inverse BWT transforms of the small data block by the functions [ippsBWTfwd\\_SmallBlock](#) and [ippsBWtInv\\_SmallBlock](#) respectively. The size of the data block must be in the range (0, 32768].

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pBuffSize</i> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>wndSize</i> is less than or equal to 0, or is greater than 32768.

## BWTFwd\_SmallBlock

*Performs the forward BWT transform for small data block.*

---

### Syntax

```
IppStatus ippsBWTFwd_SmallBlock_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
int* pIndex, Ipp8u* pBWTEBuff);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the source and destination vectors.
<i>pIndex</i>	Pointer to the index of first position for the forward BWT transform.
<i>pBWTEBuff</i>	Pointer to the additional buffer.

### Description

The function `ippsBWTFwd_SmallBlock` is declared in the `ippdc.h` file. This function performs the forward BWT transform of small block of data, that the number of elements must be in the range (0, 32768]. The function processes *len* elements starting from *pIndex* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTEBuff*. The size of this buffer must be computed by calling the function `ippsBWTGet-Size_SmalBlock` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0, or is greater than 32768.

## BWTInv\_SmallBlock

*Performs the inverse BWT transform for small data block.*

---

### Syntax

```
IppStatus ippSBWTInv_SmallBlock_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
int index, Ipp8u* pBWTBuff);
```

### Parameters

<i>pSrc</i>	Pointer to the source vector.
<i>pDst</i>	Pointer to the destination vector.
<i>len</i>	Number of elements in the source and destination vectors.
<i>index</i>	Index of first position for the inverse BWT transform.
<i>pBWTBuff</i>	Pointer to the additional buffer.

### Description

The function `ippSBWTInv_SmallBlock` is declared in the `ippdc.h` file. This function performs the inverse BWT transform of small block of data, that the number of elements must be in the range (0, 32768]. The function processes *len* elements starting from *index* element of the source vector *pSrc* and stores result in the vector *pDst*. The function uses the external buffer *pBWTBuff*. The size of this buffer must be computed by calling the function `ippSBWTGetSize_SmalBlock` beforehand.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0, or is greater than 32768; or if <i>index</i> is greater than or equal to <i>len</i> .

## Generalized Interval Transformation Coding

This section describes Intel IPP data compression functions that perform *Generalized Interval Transformation* (GIT) coding. The coding procedure is of a statistical entropy nature. The GIT algorithm is based on Interval Transformations also known as Distance coding or Inversion

frequencies [Arn97 ] and Rice-Golomb coding [Gal75]. This algorithm compresses blocks of data at ratios that are close to Huffman encoding compression ratios. However, with low entropy source data, the GIT algorithm typically provides coding at higher compression ratios as compared with the Huffman algorithm.

### Special Parameters

The GIT coding functions give you possibility to adjust the compression algorithm specifying the *strategyHint* parameter. This parameter suggests using different type of reordering the symbols in the alphabet in dependence on the input data. In some cases it can improve encoding. Table 13-11 below lists the possible values of the *strategyHint* argument and their meanings.

**Table 13-11. Parameter *strategyHint* for GIT Encoding Functions**

Value	Descriptions
<code>ippGITNoStrategy</code>	No strategy.
<code>ippGITLeftReorder</code>	Alphabet is reordered in such a way that more frequent symbols have less numbers (moved to the left).
<code>ippGITRightReorder</code>	Alphabet is reordered in such a way that more frequent symbols have greater numbers (move to right)
<code>ippGITFixedOrder</code>	Fixed alphabet order, no reordering

## EncodeGITInitAlloc

*Allocates memory and initializes the GIT encoding state structure.*

### Syntax

```
IppStatus ippEncodeGITInitAlloc_8u (int maxSrcLen, int maxDstLen,
IppGITState_8u** ppGITState);
```

### Parameters

<i>maxSrcLen</i>	Maximum length of the source buffer.
<i>maxDstLen</i>	Maximum length of the destination buffer.
<i>ppGITState</i>	Double pointer to the GIT encoding state structure.



## Description

The function `ippsEncodeGITInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the GIT encoding state structure `ppGITState`. This structure is used by the function `ippsEncodeGIT`.

The size of the GIT encoding state structure depends on the values of the `maxSrcLen` and `maxDstLen` parameters that set the upper bound on the sizes of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

The `maxSrcLen` parameter sets the size of a block for input data processing. Due to the adaptive coding used by the GIT algorithm, the compression ratio is very likely to improve as the block size increases.

## Return Values

<code>IppStsNoErr</code>	Indicates no errors.
<code>IppStsSizeErr</code>	Indicates an error if <code>maxSrcLen</code> or <code>maxDstLen</code> is less than or equal to 0.

## GITFree

*Frees memory allocated for the GIT coding structures.*

---

## Syntax

```
void ippsGITFree_8u (IppGITState_8u* pGITState);
```

## Parameters

<code>pGITState</code>	Pointer to the GIT encoding or decoding state structure.
------------------------	--

## Description

The function `ippsGITFree` is declared in the `ippdc.h` file. This function frees memory allocated for the GIT encoding or decoding state structure by the functions `ippsEncodeGITInitAlloc` or `ippsDecodeGITInitAlloc`, respectively.

## EncodeGITInit

*Initializes the GIT encoding state structure.*

---

### Syntax

```
IppStatus ippsEncodeGITInit_8u (int maxSrcLen, int maxDstLen, IppGITState_8u* pGITState);
```

### Parameters

<i>maxSrcLen</i>	Maximum length of the source buffer.
<i>maxDstLen</i>	Maximum length of the destination buffer.
<i>pGITState</i>	Pointer to the GIT encoding state structure.

### Description

The function `ippsEncodeGITInit` is declared in the `ippdc.h` file. This function initializes the GIT encoding state structure in the external buffer whose size must be computed previously by the calling the function `ippsEncodeGITGetSize`. Alternatively this structure can be initialized by the function `ippsEncodeGITInitAlloc`.

The size of the GIT encoding state structure depends on the values of the *maxSrcLen* and *maxDstLen* parameters, which set the upper bound on the sizes of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

The *maxSrcLen* parameter sets the size of a block for input data processing. Due to adaptive coding used by the GIT algorithm, the compression ratio is very likely to improve as the block size increases.




---

**NOTE.** The *maxSrcLen* and *maxDstLen* parameters of the `ippsEncodeGITInit` and `ippsEncodeGITGetSize` functions must have equal respective values to avoid errors in the `ippsEncodeGIT` function.

---

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pGITState</i> is NULL.

<code>ippStsSizeErr</code>	Indicates an error if <i>maxSrcLen</i> or <i>maxDstLen</i> is less than or equal to 0.
----------------------------	--

## EncodeGITGetSize

*Computes the size of the GIT encoding state structure.*

---

### Syntax

```
IppStatus ippEncodeGITGetSize_8u (int maxSrcLen, int maxDstLen, int*  
pGITStateSize);
```

### Parameters

<i>maxSrcLen</i>	Maximum length of the source buffer.
<i>maxDstLen</i>	Maximum length of the destination buffer.
<i>pGITStateSize</i>	Pointer to the size of the GIT encoding state structure.

### Description

The function `ippEncodeGITGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes of the GIT encoding state structure. Its size depends on the *maxSrcLen* and *maxDstLen* parameters, which are the maximum lengths of the input and output data buffers, respectively.

The function should be called prior to the function `ippEncodeGITInit`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pGITStateSize</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>maxSrcLen</i> or <i>maxDstLen</i> is less than or equal to 0.

## EncodeGIT

*Performs GIT encoding.*

---

### Syntax

```
IppStatus ippsEncodeGIT_8u (const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int* pDstLen, IppGITStrategyHint strategyHint, IppGITState_8u* pGITState);
```

### Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Length of the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>strategyHint</i>	Suggests using of the code implemented specific strategy for lexicographical reordering (see <a href="#">table "strategyHint Parameter"</a> ); default value is <code>ippGITNoStrategyHint</code> .
<i>pGITState</i>	Pointer to the GIT encoding state structure.

### Description

The function `ippsEncodeGIT` is declared in the `ippdc.h` file. This function performs GIT encoding. It processes the *srcLen* bytes of the input data *pSrc* and writes the results to the *pDst* buffer. The function uses the GIT encoding state structure *pGITState* that contains pointers to different data structures required for operation. The structure must be initialized by the functions `ippsEncodeGITInitAlloc` or `ippsEncodeGITInit` beforehand.

It is not recommended to use source data buffer less than 8Kb.




---

**NOTE.** Value of *srcLen* must not exceed that of *maxSrcLen* specified in the function `ippsEncodeGITInitAlloc` or `ippsEncodeGITInit`.

---

Nevertheless, the larger files can be encoded by subsequently calling the `ippsEncodeGIT` function several times (see the code example 13-6 below).

Encoding quality depends on the input data and in some cases can be improved by specifying the parameter *strategyHint*.

The default value for the *strategyHint* parameter is `ippGITNoStrategyHint`.

The code example 13-6 below shows how to use the `ippsEncodeGIT_8u` and supporting functions.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>srcLen</i> is greater than the value of the <i>maxSrcLen</i> parameter passed to <code>ippsEncodeGITGetSize</code> or <code>ippsEncodeGITInitAlloc</code> .

### Example 13-6 Using the Function `EncodeGIT` and Supporting Functions

```
#define BLOCKSIZE 65536

.....

FILE*          in, out;
Ipp8u*         data, code;
IppGITState_8u* pGITState;
int            GITStateSize, size, codeLen;
IppGITStrategyHint strategyHint;

.....

/*****
/* Opening the file containing input data and the file      */
/* for the compressed data:                                */
*****/

in = fopen("datafile.txt", "rb");
```

```

out = fopen("codefile", "wb");

/*****

/* Memory allocation for input data and output code buffers, */
/* and for the internal encoding state structure:           */
*****/

data = (Ipp8u*) malloc(BLOCKSIZE * sizeof(Ipp8u));
code = (Ipp8u*) malloc((BLOCKSIZE << 1) * sizeof(Ipp8u));
ippsEncodeGITGetSize_8u(BLOCKSIZE, (BLOCKSIZE << 1), &GITStateSize);
pGITState = (IppGITState_8u*) malloc(GITStateSize * sizeof(Ipp8u));
/*****

/* Initializing the memory, allocated for internal encoding */
/* state structure and setting the encoding strategy:        */
*****/

ippsEncodeGITInit_8u(BLOCKSIZE, (BLOCKSIZE << 1), pGITState);
strategyHint = ippsGITLeftReorder;
/*****

/* The main loop. In every iteration program reads the size */

/* bytes from the input data file to data array and          */

/* compresses by calling the ippsEncodeGIT_8u function.       */

/* ippsEncodeGIT_8u places the compressed data to the code   */
/* array and returns the length of compressed data in bytes  */

/* through the &codeLen argument. After that program writes  */
/* the value of codeLen and codeLen bytes of compressed data */

```

---

```

/* to the output data file. */
/*****
for( ; ; )
{
    size = fread(&data[0], sizeof(Ipp8u), BLOCKSIZE, in);
    if(size <= 0)
        break;
    if(ippEncodeGIT_8u(data, size, code, &codeLen, strategyHint, pGITState) != ippStsNoErr)
        return (-1);
    fwrite(&codeLen, sizeof(int), 1, out);
    fwrite(code, sizeof(Ipp8u), codeLen, out);
} /* for */
/*****
/* Free the memory allocated for input data and output */
/* code buffers and for the internal encoding state */
/* structure: */
/*****
free(pGITState);

free(data);
free(code);
fclose(in);
fclose(out);
.....

```

## DecodeGITInitAlloc

*Allocates memory and initializes the GIT decoding state structure.*

---

### Syntax

```
IppStatus ippDecodeGITInitAlloc_8u (int maxSrcLen, int maxDstLen,
IppGITState_8u** ppGITState);
```

### Parameters

<i>maxSrcLen</i>	Maximum length of the source buffer.
<i>maxDstLen</i>	Maximum length of the destination buffer.
<i>ppGITState</i>	Double pointer to the GIT decoding state structure.

### Description

The function `ippDecodeGITInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the GIT decoding state structure *ppGITState*. This structure is used by the function `ippDecodeGIT`.

The size of the GIT decoding state structure depends on the value of the *maxDstLen* parameter, which sets the upper bound on the lengths of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>ppGITState</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>maxSrcLen</i> or <i>maxDstLen</i> is less than or equal to 0.

## DecodeGITInit

*Initializes the GIT decoding state structure.*

---

### Syntax

```
IppStatus ippDecodeGITInit_8u (int maxDstLen, IppGITState_8u* pGITState);
```



## Parameters

<i>maxDstLen</i>	Maximum length of the destination buffer.
<i>pGITState</i>	Pointer to memory allocated for the GIT decoding state structure.

## Description

The function `ippsDecodeGITInit` is declared in the `ippdc.h` file. This function initializes the GIT decoding state structure *pGITState* in the external buffer. The size of this buffer must be computed by the function `ippsDecodeGITGetSize`. Alternatively this structure can be initialized by the function `ippsDecodeGITInitAlloc`.

The size of the GIT decoding state structure depends on the value of the parameter *maxDstLen*, which sets the upper bound on the lengths of the source and destination buffers. In subsequent functions, do not specify a greater size of the buffers, as the functions will return an error message.



**NOTE.** The parameter *maxDstLen* of the `ippsDecodeGITInit` and `ippsDecodeGITGetSize` functions must be the same to avoid an error in the function `ippsDecodeGIT`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pGITState</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>maxDstLen</i> is less than or equal to 0.

## DecodeGITGetSize

*Computes the size of the GIT decoding state structure.*

---

### Syntax

```
IppStatus ippsDecodeGITGetSize_8u (int maxSrcLen, int* pGITStateSize);
```

### Parameters

<i>maxSrcLen</i>	Maximum length of the source buffer.
------------------	--------------------------------------

*pGITStateSize*                      Pointer to the size of the GIT decoding state structure.

## Description

The function `ippDecodeGITGetSize` is declared in the `ippdc.h` file. This function computes the size in bytes *pGITStateSize* of the GIT decoding state structure. Its value depends on the *maxSrcLen* parameter, which is the maximum length of the input data buffer.

The function should be called prior to the function `ippDecodeGITInit`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <i>pGITStateSize</i> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>maxSrcLen</i> is less than or equal to 0.

## DecodeGIT

*Performs GIT decoding.*

---

### Syntax

```
IppStatus ippDecodeGIT_8u (const Ipp8u* pSrc, int srcLen, Ipp8u* pDst, int* pDstLen, IppGITStrategyHint strategyHint, IppGITState_8u* pGITState);
```

### Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>srcLen</i>	Length of the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>strategyHint</i>	Suggests using of the code implemented specific strategy for lexicographical reordering (see <a href="#">table "strategyHint Parameter"</a> ); default value is <code>ippGITNoStrategyHint</code> .
<i>pGITState</i>	Pointer to the GIT decoding state structure.

### Description

The function `ippDecodeGIT` is declared in the `ippdc.h` file. This function performs GIT decoding - it decodes data encoded by the `ippEncodeGIT` function.

The `ippsDecodeGIT` uses the GIT state structure `pGITState` that contains pointers to different data structures required for GIT decoding. This structure must be initialized by the functions `ippsDecodeGITInitAlloc` or `ippsDecodeGITInit` beforehand. The function `ippsDecodeGIT` requires enough memory for decoding. It obtains the source data from the `pSrc` source vector with the `srcLen` length and writes the decoded data to the `pDst` destination vector. The size of destination buffer must be equal to or greater than `maxDstLen` (see `ippsDecodeGITInitAlloc` and `ippsDecodeGITInit`). `pDstLen` is an output parameter, it returns the actual size of the decoded data.

The code Example 13-7 below shows how to use the `ippsDecodeGIT_8u` and supporting functions.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if there is not enough memory allocated for the destination buffer.

### Example 13-7 Using the Function `DecodeGIT` and Supporting Functions

```
#define BLOCKSIZE 65536

.....

FILE* in, out;

Ipp8u* rdata, code;

int GITStateSize, size,
codeLen, rdataLen;

IppGITState_8u pGITState;
```

```

IppGITStrategyHint strategyHint;

.....

/*****

/* Opening the file containing input compressed data and the */
/* file for the restored (uncompressed) data:                */
/*****

in = fopen("codefile", "rb");
out = fopen("datafile_restored.txt", "wb");

/*****

/* Memory allocation for input code and output restored data */
/* buffers, and for the internal decoding state structure:    */
/*****

rdata = (Ipp8u*) malloc((BLOCKSIZE << 1) * sizeof(Ipp8u));
code = (Ipp8u*) malloc(BLOCKSIZE * sizeof(Ipp8u));

ippsDecodeGITGetSize_8u((BLOCKSIZE << 1), &GITStateSize);
pGITState = (IppGITState_8u*) malloc(GITStateSize * sizeof(Ipp8u));

/*****

/* Initializing the memory, allocated for internal decoding */
/* state structure and setting the decoding strategy          */
/* (should be the same as the strategy for encoding) :        */
/*****

ippsDecodeGITInit_8u(BLOCKSIZE, pGITState);
strategyHint = ippGITLeftReorder;

/*****

/* The main loop. On every iteration program reads the size  */

```

---

```

/* of the compressed block (codeLen) from the input code */
/* file. After that program reads the codeLen bytes of code */
/* from the input code file to the code array. The call of */
/* ippsDecodeGIT_8u performs the decoding of codeLen bytes */
/* of code to the rdata error of rdataLen. At the end */
/* program writes the rdataLen bytes of restored data(rdata) */
/* to the output file. */
/*****/
for( ; ; )

{
    size = fread(&codeLen, sizeof(int), 1, in);
    if(size <= 0)
        break;

    fread(&code[0], sizeof(Ipp8u), codeLen, in);
    ippsDecodeGIT_8u(code, codeLen, rdata, &rdataLen, strategyHint, pGITState);

    fwrite(rdata, sizeof(Ipp8u), rdataLen, out);
} /* for */
/*****/
/* Free the memory allocated for input code and output */
/* restored data buffers and for the internal decoding state */
/* structure: */
/*****/
free(pGITState);

```

```
free(code);  
  
free(rdata);  
  
fclose(in);  
  
fclose(out);  
  
.....
```

### Move To Front Functions

This section describes the functions that performs Move To Front (MTF) data transform method. The basic idea is to represent the symbols of the source sequence as the current indexes of that symbols in the modified alphabet. This alphabet is a list where frequently used symbols are placed in the upper lines. When the given symbols occurs it is replaced by its index in the list, then this symbol is moved in the first position in the list, and all indexes are updated. For example, the sequence "baabbfffaczzdd" contains symbols that form the ordered 'alphabet'{'a', 'b', 'c', 'd', 'f', 'z'}. The function will operate in the following manner:

**Table 13-12 Move To Front Operation**

source	destination	alphabet
		0, 1, 2, 3, 4, 5, 6
		'a', 'b', 'c', 'd', 'f', 'z'
b	1	'b', 'a', 'c', 'd', 'f', 'z'
a	1	'a', 'b', 'c', 'd', 'f', 'z'
a	0	'a', 'b', 'c', 'd', 'f', 'z'
b	1	'b', 'a', 'c', 'd', 'f', 'z'
f	4	'f', 'b', 'a', 'c', 'd', 'z'
f	0	'f', 'b', 'a', 'c', 'd', 'z'
f	0	'f', 'b', 'a', 'c', 'd', 'z'
a	2	'a', 'f', 'b', 'c', 'd', 'z'
c	3	'c', 'a', 'f', 'b', 'd', 'z'
c	0	'c', 'a', 'f', 'b', 'd', 'z'
z	5	'z', 'c', 'a', 'f', 'b', 'd'
z	0	'z', 'c', 'a', 'f', 'b', 'd'
d	5	'd', 'z', 'c', 'a', 'f', 'b'
d	0	'd', 'z', 'c', 'a', 'f', 'b'

Finally, the function returns the destination sequence: 11014002305050.

These transformed data can be used for the following effective compression. This method is often used after Burrows-Wheeler transform.

## MTFInitAlloc

Allocates memory and initializes the MTF structure.

### Syntax

```
IppStatus ippMtfInitAlloc_8u(IppMTFState_8u** ppMTFState);
```

### Parameters

*ppMTFState*                      Double pointer to the MTF transform structure.

### Description

The function `ippMtfInitAlloc` is declared in the `ippdc.h` file. This function allocates memory and initializes the MTF structure that contains required parameters for the MTF transform. This structure is used by the functions `ippMtfFwd` and `ippMtfInv`.

Code [example 13-8](#) shows how to use the function `ippMtfInitAlloc_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>ppMTFState</i> pointer is NULL.
<code>ippMemAllocErr</code>	Indicates an error if no memory is allocated.

## MTFFree

Frees memory allocated for the MTF structure.

### Syntax

```
void ippMtfFree_8u(IppMTFState_8u* pMTFState);
```

### Parameters

*pMTFState*                      Pointer to the MTF structure.

### Description

The function `ippMtfFree` is declared in the `ippdc.h` file. This function frees memory allocated for the MTF structure by the function `ippMtfInitAlloc`.

## MTFInit

*Initializes the MTF structure.*

---

### Syntax

```
IppStatus ippmTFInit_8u(IppMTFState_8u* pMTFState);
```

### Parameters

*pMTFState*                      Pointer to the MTF structure.

### Description

The function `ippmTFInit` is declared in the `ippdc.h` file. This function initializes the MTF structure that contains parameters for the MTF transform in the external buffer. This structure is used by the functions `ippmTFFwd` and `ippmTFInv`. The size of this buffer must be computed previously by calling the function `ippmTFGetSize`.

### Return Values

`ippStsNoErr`                      Indicates no error.  
`ippStsNullPtrErr`                Indicates an error if *pMTFState* pointer is NULL.

## MTFGetSize

*Computes the size of the MTF structure.*

---

### Syntax

```
IppStatus ippmTFGetSize_8u(int* pMTFStateSize);
```

### Parameters

*pMTFStateSize*                      Pointer to the computed MTF structure size.

### Description

The function `ippmTFGetSize` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) that is required for the MTF structure. This function must be called prior to the function `ippmTFInit`.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pMTFStateSize</i> pointer is NULL.

## MTFFwd

*Performs the forward MTF transform.*

---

### Syntax

```
IppStatus ippMTFFwd_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,  
IppMTFState_8u* pMTFState);
```

### Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>srcLen</i>	Number of elements in the source and destination buffers.
<i>pMTFState</i>	Pointer to the MTF structure.

### Description

The function `ippMTFFwd` is declared in the `ippdc.h` file. This function performs the forward MTF transform of *len* elements of the data in the source buffer *pSrc* and stores result in the buffer *pDst*. The parameters of the MTF transform are specified in the MTF structure *pMTFState* that must be initialized by the functions `ippMTFInitAlloc` or `ippMTFInit` beforehand.

Example 13-8 below shows how to use the function `ippMTFFwd_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <i>len</i> is less than or equal to 0.

## Example 13-8. Using the Function `ippsMTFFwd`

```
void func_MTF()
{
    IppMTFState_8u* ppMTFState;
    Ipp8u pSrc[] = "adadasdasd";
    Ipp8u pDst[10];
    int len = 10;

    ippsMTFInitAlloc_8u(&ppMTFState);
    ippsMTFFwd_8u( pSrc, pDst, len, ppMTFState);
}
```

Result: pDst -> 97 100 1 1 1 115 2 2 2 2

## MTFInv

*Performs the inverse MTF transform.*

---

### Syntax

```
IppStatus ippsMTFInv_8u(const Ipp8u* pSrc, Ipp8u* pDst, int len,
IppMTFState_8u* pMTFState);
```

### Parameters

<i>pSrc</i>	Pointer to the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>len</i>	Number of elements in the source and destination buffers.
<i>pMTFState</i>	Pointer to the MTF structure.

### Description

The function `ippsMTFInv` is declared in the `ippdc.h` file. This function performs the inverse MTF transform of *len* elements of data in the source buffer *pSrc* and stores result in the buffer *pDst*. The parameters of the MTF transform are specified in the MTF structure *pMTFState* that must be initialized by the functions `ippsMTFInitAlloc` or `ippsMTFInit` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <code>len</code> is less than or equal to 0.

## Run Length Encoding Functions

This section describes functions that performs data compression using Run Length Encoding (RLE) method. The main idea of this method is to replace the sequence of the same symbol by a number of its occurrences. For example, the string "aaabbbbbaaaaabbabbbbb" will be transformed to "aa1bb2aa3bb0abb4". The important parameters of the RLE method is a threshold - that is a minimum number of the repeated symbols exceeding of which starts up the encoding procedure. In the Intel IPP data compression functions this threshold is set to 2. Upper bound of the number of repetitions is 255 - to use one byte for repetition counter.

## EncodeRLE

*Performs RLE encoding.*

---

### Syntax

```
IppStatus ippEncodeRLE_8u(const Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst,  
int* pDstLen);
```

### Parameters

<code>ppSrc</code>	Double pointer to the source data buffer.
<code>pSrcLen</code>	Pointer to the number of elements in the source buffer after encoding point to the number of remaining elements.
<code>pDst</code>	Pointer to the destination data buffer.
<code>pDstLen</code>	Pointer to the number of elements in the destination buffer, after encoding to the actual number of elements in the destination buffer.

### Description

The function `ippsEncoderLE` is declared in the `ippdc.h` file. This function performs RLE encoding *pSrcLen* elements of the source data buffer *ppSrc* and stores the result in the buffer *pDst*. After encoding the function updates the *pSrcLen* thus it points to the number of remaining elements of the source buffer, and *pDstLen* points to the actual number of elements stored in the destination buffer.

Sometimes the number of output elements can exceed the specified size of destination buffer. In this case the function returns the warning message and pointer to the number of not-encoded source elements. The user can call the function `ippsEncoderLE` again to complete encoding.

Code [example 13-9](#) shows how to use the function `ippsEncoderLE_8u`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specicified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>pSrcLen</i> or <i>pDstLen</i> is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

## DecodeRLE

*Performs RLE decoding.*

---

### Syntax

```
IppStatus ippsDecodeRLE_8u(const Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst,
int* pDstLen);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source data buffer.
<i>pSrcLen</i>	Pointer to the number of elements in the source buffer, after decoding pointer to the number of remaining elements.
<i>pDst</i>	Pointer to the destination data buffer.

*pDstLen* Pointer to the number of elements in the destination buffer, after encoding to the actual number of elements in the destination buffer.

## Description

The function `ippSDecodeRLE` is declared in the `ippdc.h` file. This function performs RLE decoding of *srcLen* elements of the source data buffer *ppSrc* and stores the result in the destination buffer *pDst*. After decoding the function updates the *pSrcLen* thus it points to the number of remaining elements of the source buffer, and *pDstLen* points to the actual number of elements stored in the destination buffer.

Sometimes the number of output elements can exceed the specified size of destination buffer. In this case the function returns the warning message and pointer to the number of not-decoded source elements. The user can call the function `ippSDecodeRLE` again to complete decoding.

Example 13-9 below shows how to use the function `ippSDecodeRLE_8u`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the specified pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if <i>pSrcLen</i> or <i>pDstLen</i> is less than or equal to 0.
<code>ippStsSrcDataErr</code>	Indicates an error if <i>ppSrc</i> contains unsupported data.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

## Example 13-9 Using RLE Encoding and Decoding Functions

```
void func_RLE()
{
    Ipp8u* pSrc = (Ipp8u*)"aaaddddssadsaadd";
    int pSrcLen = 18;
    int pDstLen = 18;
    int pDstDLen = 18;
```

```

Ipp8u* pDst = ippsMalloc_8u(pDstLen);
Ipp8u* pDstD = ippsMalloc_8u(pDstDLen);
ippsEncodeRLE_8u(&pSrc, &pSrcLen, pDst, &pDstLen);

ippsDecodeRLE_8u(&pDst, &pDstLen, pDstD, &pDstDLen);
}
Result:  pDst:  aa1dd3ss0adsaa0ddl
         pDstD: aaaddddssadsaaddd

```

## bzip2 Coding Functions

This section describes different Intel IPP functions to perform bzip2 encoding and decoding.

### EncodeRLEInitAlloc\_BZ2

*Allocates memory and initializes the bzip2-specific RLE structure.*

---

#### Syntax

```
IppStatus ippsEncodeRLEInitAlloc_BZ2_8u(IppRLEState_BZ2** ppRLEState);
```

#### Parameters

*ppRLEState*                      Double pointer to the bzip2-specific RLE structure.

#### Description

The function `ippsEncodeRLEInitAlloc_BZ2` is declared in the `ippdc.h` file. This function allocates memory and initializes the bzip2-specific RLE structure that contains required parameters for the RLE. This structure is used by the function [ippsEncodeRLE\\_BZ2](#).

#### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>ppRLEState</i> pointer is NULL.

## RLEFree\_BZ2

*Frees memory allocated for the bzip2-specific RLE structure.*

---

### Syntax

```
void ippsRLEFree_BZ2_8u(IppRLEState_BZ2* pRLEState);
```

### Parameters

*pRLEState*                      Pointer to the bzip2-specific RLE structure.

### Description

The function *ippsRLEFree\_BZ2* is declared in the *ippdc.h* file. This function frees memory allocated for bzip2-specific RLE structure by the function *ippsEncodeRLEInitAlloc\_BZ2*.

## EncodeRLEInit\_BZ2

*Initializes the bzip2-specific RLE structure.*

---

### Syntax

```
IppStatus ippsEncodeRLEInit_BZ2_8u(IppRLEState_BZ2* pRLEState);
```

### Parameters

*pRLEState*                      Pointer to the bzip2-specific RLE structure.

### Description

The function *ippsEncodeRLEInit\_BZ2* is declared in the *ippdc.h* file. This function initializes the bzip2-specific RLE structure that contains parameters for the RLE in the external buffer. This structure is used by the function *ippsEncodeRLE\_BZ2*. The size of this buffer must be computed previously by calling the function *ippsRLEGetSize\_BZ2*.

### Return Values

*ippStsNoErr*                      Indicates no error.  
*ippStsNullPtrErr*                Indicates an error if *pRLEState* pointer is NULL.

## RLEGetSize\_BZ2

Compute the size of the state structure for the bzip2-specific RLE.

---

### Syntax

```
IppStatus ippsRLEGetSize_BZ2_8u(int* pRLEStateSize);
```

### Parameters

<i>pRLEStateSize</i>	Pointer to the size of the state structure for bzip2-specific RLE.
----------------------	--

### Description

The function `ippsRLEGetSize_BZ2` is declared in the `ippdc.h` file. This function computes the size of memory (in bytes) of the internal state structure for the bzip2-specific RLE.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if <i>pRLEStateSize</i> pointer is NULL.

## EncodeRLE\_BZ2

Performs the bzip2-specific RLE.

---

### Syntax

```
IppStatus ippsEncoderRLE_BZ2_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen, IppRLEState_BZ2* pRLEState);
```

### Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer.
<i>pRLEState</i>	Pointer to the bzip2-specific RLE state structure.



## Description

The function `ippsEncoderLE_BZ2` is declared in the `ippdc.h` file. This function performs RLE encoding with thresholding equal to 4. It processes the input data `ppSrc` and writes the results to the `pDst` buffer. The function uses the bzip2-specific RLE state structure `pRLEState`. This structure must be initialized by the functions `ippsEncoderLEInitAlloc_BZ2` or `ippsEncoderLEInit_BZ2` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the source or destination buffer is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

## EncodeRLEFlush\_BZ2

*Flushes the remaining data after RLE.*

### Syntax

```
IppStatus ippsEncoderLEFlush_BZ2_8u(Ipp8u* pDst, int* pDstLen,
IppRLEState_BZ2* pRLEState);
```

### Parameters

<code>pDst</code>	Pointer to the destination buffer.
<code>pDstLen</code>	Pointer to the length of the destination buffer.
<code>pRLEState</code>	Pointer to the bzip2-specific RLE state structure.

### Description

The function `ippsEncoderLEFlush_BZ2` is declared in the `ippdc.h` file. This function flushes the remaining data after RLE encoding with thresholding equal to 4. The function uses the initialized bzip2-specific RLE state structure `pRLEState`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
--------------------------	---------------------

<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the destination buffer is less than or equal to 0.

## RLEGetInUseTable

*Gets the pointer to the `inUse` vector from the RLE state structure.*

---

### Syntax

```
IppStatus ippSRLEGetInUseTable_8u(Ipp8u inUse[256], IppRLEState_BZ2* pRLEState);
```

### Parameters

<code>inUse</code>	Pointer to the <code>inUse</code> vector.
<code>pRLEState</code>	Pointer to the bzip2-specific RLE state structure.

### Description

The function `ippSRLEGetInUseTable` is declared in the `ippdc.h` file. This function gets the pointer to the `inUse` vector (table) from the initialized bzip2-specific RLE state structure `pRLEState`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .

## DecodeRLE\_BZ2

*Performs the bzip2-specific RLE.*

---

### Syntax

```
IppStatus ippSDecodeRLE_BZ2_8u(Ipp8u** ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen);
```

## Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer, after decoding - pointer to the size of the remaining data.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer, after decoding - pointer to the resulting size of the destination buffer.

## Description

The function `ippsDecodeRLE_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific RLE decoding with thresholding equal to 4.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the source or destination buffer is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

## EncodeZ1Z2\_BZ2

Performs the bzip2-specific Z1Z2 encoding.

### Syntax

```
IpplStatus ippsEncodeZ1Z2_BZ2_8u16u(Ipp8u** ppSrc, int* pSrcLen, Ippl6u* pDst,
int* pDstLen, int freqTable[258]);
```

## Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer, after decoding - pointer to the size of the remaining data.
<i>pDst</i>	Pointer to the destination buffer.

<i>pDstLen</i>	Pointer to the length of the destination buffer, after decoding - pointer to the resulting size of the destination buffer.
<i>freqTable</i>	Table of frequencies collected for the alphabet symbols.

## Description

The function `ippsEncodeZ1Z2_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific Z1Z2 encoding.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the source or destination buffer is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

## DecodeZ1Z2\_BZ2

*Performs the bzip2-specific Z1Z2 decoding.*

---

## Syntax

```
IppStatus ippsDecodeZ1Z2_BZ2_16u8u(Ipp16u** ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen);
```

## Parameters

<i>ppSrc</i>	Double pointer to the source buffer.
<i>pSrcLen</i>	Pointer to the length of the source buffer, after decoding - pointer to the size of the remaining data.
<i>pDst</i>	Pointer to the destination buffer.
<i>pDstLen</i>	Pointer to the length of the destination buffer, after decoding - pointer to the resulting size of the destination buffer.

## Description

The function `ippsDecodeZ1Z2_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific Z1Z2 decoding.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the source or destination buffer is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

## ReduceDictionary

*Performs the dictionary reducing.*

---

### Syntax

```
IppStatus ippsReduceDictionary_8u_I(const Ipp8u inUse[256], Ipp8u* pSrcDst,
int srcDstLen, int* pSizeDictionary);
```

### Parameters

<i>inUse</i>	Table of 256 values of the <code>Ipp8u</code> type.
<i>pSrcDst</i>	Pointer to the source and destination buffer.
<i>srcDstLen</i>	Length of the source and destination buffer.
<i>pSizeDictionary</i>	Pointer to the size of the dictionary on entry, and to the size of reduced dictionary after operation.

### Description

The function `ippsReduceDictionary` is declared in the `ippdc.h` file. This function performs the dictionary reducing.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .

<code>ippStsSizeErr</code>	Indicates an error if length of the source and destination buffer is less than or equal to 0.
----------------------------	---

## ExpandDictionary

*Performs the dictionary expanding.*

---

### Syntax

```
IppStatus ippseExpandDictionary_8u_I(const Ipp8u inUse[256], Ipp8u* pSrcDst,
int srcDstLen, int sizeDictionary);
```

### Parameters

<i>inUse</i>	Table of 256 values of the Ipp8u type.
<i>pSrcDst</i>	Pointer to the source and destination buffer.
<i>srcDstLen</i>	Length of the source and destination buffer.
<i>sizeDictionary</i>	Size of the dictionary on entry, and to the size of expanded dictionary after operation.

### Description

The function `ippseExpandDictionary` is declared in the `ippdc.h` file. This function performs the dictionary expanding.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the source and destination buffer is less than or equal to 0.

## CRC32\_BZ2

*Computes the CRC32 checksum for the source data buffer.*

---

### Syntax

```
IppStatus ippseCRC32_BZ2_8u(const Ipp8u* pSrc, int srcLen, Ipp32u* pCRC32);
```

## Parameters

<i>pSrc</i>	Pointer to the source data buffer.
<i>srcLen</i>	Number of elements in the source data buffer.
<i>pCRC32</i>	Pointer to the accumulated checksum value.

## Description

The function `ippsCRC32_BZ2` is declared in the `ippdc.h` file. This function computes the checksum for *srcLen* elements of the source data buffer *pSrc* and stores it in the *pCRC32*. The checksum is computed using the CRC32 direct algorithm that is specific for the bzip2 coding.

You can use this function to compute the accumulated value of the checksum for multiple buffers by specifying as an input parameter the checksum value obtained in the preceding function call.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <i>pSrc</i> pointer is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if the length of the source vector is less than or equal to 0.

## EncodeHuffGetSize\_BZ2

*Computes the size of the internal state for bzip2-specific Huffman encoding.*

---

### Syntax

```
IppStatus ippsEncodeHuffGetSize_BZ2_16u8u(int wndSize, int*  
pEncodeHuffStateSize);
```

### Parameters

<i>wndSize</i>	Size of the block to be processed.
<i>pEncodeHuffStateSize</i>	Pointer to the size of the internal state for bzip2-specific Huffman coding.

## Description

The function `ippsEncodeHuffGetSize_BZ2` is declared in the `ippdc.h` file. This function computes the size of the internal state structure for bzip2-specific Huffman encoding in dependence of the size of the block to be encoded.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pEncodeHuffStateSize</code> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>wndSize</code> is less than or equal to 0.

## EncodeHuffInit\_BZ2

*Initializes the elements of the bzip2-specific internal state for Huffman encoding.*

---

### Syntax

```
ippStatus ippsEncodeHuffInit_BZ2_16u8u(int sizeDictionary, const int
freqTable[258], const Ipp16u* pSrc, int srcLen, IppEncodeHuffState_BZ2*
pEncodeHuffState);
```

### Parameters

<code>sizeDictionary</code>	Size of the dictionary.
<code>freqTable</code>	Table of frequencies of symbols.
<code>pSrc</code>	Pointer to the source vector.
<code>srcLen</code>	Length of the source vector.
<code>pEncodeHuffState</code>	Pointer to internal state structure for bzip2 specific Huffman coding.

## Description

The function `ippsEncodeHuffInit_BZ2` is declared in the `ippdc.h` file. This function initializes the elements of the bzip2-specific internal state for Huffman encoding. This structure is used by the function `ippsEncodeHuff_BZ2`. The size of this buffer must be computed previously by calling the function `ippsEncodeHuffGetSize_BZ2`.



## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the source buffer is less than or equal to 0.

## EncodeHuffInitAlloc\_BZ2

*Allocated memory and initializes the elements of the bzip2-specific internal state for Huffman encoding.*

---

### Syntax

```
IppStatus ippsEncodeHuffInitAlloc_BZ2_16u8u(int wndSize, int sizeDictionary,  
const int freqTable[258], const Ipp16u* pSrc, int srcLen,  
IppEncodeHuffState_BZ2** ppEncodeHuffState);
```

### Parameters

<code>wndSize</code>	Size in bytes of the block to be processed.
<code>sizeDictionary</code>	Size of the dictionary.
<code>freqTable</code>	Table of frequencies of symbols.
<code>pSrc</code>	Pointer to the source vector.
<code>srcLen</code>	Length of the source vector.
<code>ppEncodeHuffState</code>	Double pointer to the state structure for bzip2 specific Huffman coding.

### Description

The function `ippsEncodeHuffInit_BZ2` is declared in the `ippdc.h` file. This function allocates memory and initializes the elements of the bzip2-specific internal state `ppEncodeHuffState` for Huffman encoding. This structure is used by the function [ippsEncodeHuff\\_BZ2](#).

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .

<code>ippStsSizeErr</code>	Indicates an error if length of the source buffer is less than or equal to 0.
----------------------------	---

## EncodeHuffFree\_BZ2

*Frees memory allocated for the bzip2-specific Huffman encoding structure.*

---

### Syntax

```
void ippEncodeHuffFree_BZ2_16u8u(IppEncodeHuffState_BZ2* pEncodeHuffState);
```

### Parameters

<code>pEncodeHuffState</code>	Pointer to the bzip2-specific Huffman coding structure.
-------------------------------	---

### Description

The function `ippEncodeHuffFree_BZ2` is declared in the `ippdc.h` file. This function frees memory allocated for bzip2-specific Huffman encoding structure by the function `ippEncodeHuffInitAlloc_BZ2`.

## PackHuffContext\_BZ2

*Performs the bzip2-specific encoding of Huffman context.*

---

### Syntax

```
IppStatus ippPackHuffContext_BZ2_16u8u(Ipp32u* pCode, int* pCodeLenBits, Ipp8u* pDst, int* pDstLen, IppEncodeHuffState_BZ2* pEncodeHuffState);
```

### Parameters

<code>pCode</code>	Pointer to the bit buffer.
<code>pCodeLenBits</code>	Number of valid bits in the bit buffer.
<code>pDst</code>	Pointer to the destination vector.
<code>pDstLen</code>	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.

*pEncodeHuffState*      Pointer to internal state structure for bzip2 specific Huffman encoding.

## Description

The function `ippsPackHuffContext_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific encoding of the *Huffman context*. The function uses the bzip2-specific Huffman encoding state structure *pEncodeHuffState*. This structure must be initialized by the functions `ippsEncodeHuffInitAlloc_BZ2` or `ippsEncodeHuffInit_BZ2` beforehand.

## Return Values

`ippStsNoErr`            Indicates no error.

`ippStsNullPtrErr`      Indicates an error if one of the pointers is `NULL`.

`ippStsSizeErr`        Indicates an error if length of the destination buffer is less than or equal to 0.

`ippStsDstSizeLessExpected` Indicates a warning if size of the destination buffer is insufficient to store all output elements.

## EncodeHuff\_BZ2

Performs the bzip2-specific Huffman encoding.

### Syntax

```
IppStatus ippsEncodeHuff_BZ2_16u8u(Ipp32u* pCode, int* pCodeLenBits, Ipp16u** ppSrc, int* pSrcLen, Ipp8u* pDst, int* pDstLen, IppEncodeHuffState_BZ2* pEncodeHuffState);
```

### Parameters

*pCode*                    Pointer to the bit buffer.

*pCodeLenBits*           Number of valid bits in the bit buffer.

*ppSrc*                   Double pointer to the source vector.

*pSrcLen*                Pointer to the length of source vector.

*pDst*                    Pointer to the destination vector.

*pDstLen*                Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.

*pEncodeHuffState*      Pointer to internal state structure for bzip2 specific Huffman encoding.

## Description

The function `ippsEncodeHuff_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific Huffman encoding. The function uses the bzip2-specific Huffman encoding state structure *pEncodeHuffState*. This structure must be initialized by the functions `ippsEncodeHuffInitAlloc_BZ2` or `ippsEncodeHuffInit_BZ2` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the source or destination buffer is less than or equal to 0.
<code>ippStsDstSizeLessExpected</code>	Indicates a warning if size of the destination buffer is insufficient to store all output elements.

## DecodeHuffGetSize\_BZ2

*Computes the size of the internal state for bzip2-specific Huffman decoding.*

---

### Syntax

```
IppStatus ippsDecodeHuffGetSize_BZ2_8u16u(int wndSize, int*
pDecodeHuffStateSize);
```

### Parameters

<i>wndSize</i>	Size of the block to be processed.
<i>pDecodeHuffStateSize</i>	Pointer to the size of the internal state for bzip2-specific Huffman coding.

### Description

The function `ippsDecodeHuffGetSize_BZ2` is declared in the `ippdc.h` file. This function computes the size of the internal state structure for bzip2-specific Huffman decoding.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the pointer <code>pDecodeHuffStateSize</code> is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>wndSize</code> is less than or equal to 0.

## DecodeHuffInit\_BZ2

*Initializes the elements of the bzip2-specific internal state for Huffman decoding.*

---

### Syntax

```
IppStatus ippDecodeHuffInit_BZ2_8u16u(int sizeDictionary,  
IppDecodeHuffState_BZ2* pDecodeHuffState);
```

### Parameters

<code>sizeDictionary</code>	Size of the dictionary.
<code>pDecodeHuffState</code>	Pointer to internal state structure for bzip2 specific Huffman coding.

### Description

The function `ippDecodeHuffInit_BZ2` is declared in the `ippdc.h` file. This function initializes the elements of the bzip2-specific internal state for Huffman decoding. This structure is used by the function `ippDecodeHuff_BZ2`. The size of this buffer must be computed previously by calling the function `ippDecodeHuffGetSize_BZ2`.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pDecodeHuffState</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>sizeDictionary</code> is less than or equal to 0.

## DecodeHuffInitAlloc\_BZ2

*Allocated memory and initializes the elements of the bzip2-specific internal state for Huffman decoding.*

---

### Syntax

```
IppStatus ippsDecodeHuffInitAlloc_BZ2_8u16u(int wndSize, int sizeDictionary,
IppDecodeHuffState_BZ2** ppDecodeHuffState);
```

### Parameters

<i>wndSize</i>	Size in bytes of the block to be processed.
<i>sizeDictionary</i>	Size of the dictionary.
<i>pDecodeHuffState</i>	Pointer to the state structure for bzip2 specific Huffman decoding.

### Description

The function `ippsDecodeHuffInitAlloc_BZ2` is declared in the `ippdc.h` file. This function allocates memory and initializes the elements of the bzip2-specific internal state `pDecodeHuffState` for Huffman decoding. This structure is used by the function `ippsDecodeHuff_BZ2`.

### Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if the <code>pDecodeHuffState</code> pointer is NULL.
<code>ippStsSizeErr</code>	Indicates an error if <code>sizeDictionary</code> or <code>wndSize</code> is less than or equal to 0.

## DecodeHuffFree\_BZ2

*Frees memory allocated for the bzip2-specific Huffman decoding structure.*

---

### Syntax

```
void ippsDecodeHuffFree_BZ2_8u16u(IppDecodeHuffState_BZ2* pDecodeHuffState);
```

## Parameters

*pDecodeHuffState*      Pointer to the bzip2-specific Huffman decoding structure.

## Description

The function `ippsDecodeHuffFree_BZ2` is declared in the `ippdc.h` file. This function frees memory allocated for bzip2-specific Huffman decoding structure by the function `ippsDecodeHuffInitAlloc_BZ2`.

## UnpackHuffContext\_BZ2

*Performs the bzip2-specific decoding of Huffman context.*

---

## Syntax

```
IppStatus IppStatus ippsUnpackHuffContext_BZ2_8u16u(Ipp32u* pCode, int*
pCodeLenBits, Ipp8u** ppSrc, int* pSrcLen, IppDecodeHuffState_BZ2*
pDecodeHuffState);
```

## Parameters

*pCode*      Pointer to the bit buffer.

*pCodeLenBits*      Number of valid bits in the bit buffer.

*ppSrc*      Double pointer to the source vector.

*pSrcLen*      Pointer to the size of source buffer on input, pointer to the resulting length of the source vector on output.

*pDecodeHuffState*      Pointer to internal state structure for bzip2 specific Huffman decoding.

## Description

The function `ippsUnpackHuffContext_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific decoding of the *Huffman context*. The function uses the bzip2-specific Huffman decoding state structure *pDecodeHuffState*. This structure must be initialized by the functions `ippsDecodeHuffInitAlloc_BZ2` or `ippsDecodeHuffInit_BZ2` beforehand.

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the destination buffer is less than or equal to 0.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning if size of the source buffer is insufficient to store all output elements.

## DecodeHuff\_BZ2

*Performs the bzip2-specific Huffman decoding.*

---

### Syntax

```
IppStatus IppStatus ippsDecodeHuff_BZ2_8u16u(Ipp32u* pCode, int* pCodeLenBits,
Ipp8u** ppSrc, int* pSrcLen, Ipp16u* pDst, int* pDstLen,
IppDecodeHuffState_BZ2* pDecodeHuffState);
```

### Parameters

<code>pCode</code>	Pointer to the bit buffer.
<code>pCodeLenBits</code>	Number of valid bits in the bit buffer.
<code>ppSrc</code>	Double pointer to the source vector.
<code>pSrcLen</code>	Pointer to the size of source buffer.
<code>pDst</code>	Pointer to the destination vector.
<code>pDstLen</code>	Pointer to the size of destination buffer on input, pointer to the resulting length of the destination vector on output.
<code>pDecodeHuffState</code>	Pointer to internal state structure for bzip2 specific Huffman decoding.

### Description

The function `ippsDecodeHuff_BZ2` is declared in the `ippdc.h` file. This function performs the bzip2-specific Huffman decoding. The function uses the bzip2-specific Huffman decoding state structure `pDecodeHuffState`. This structure must be initialized by the functions [ippsDecodeHuffInitAlloc\\_BZ2](#) or [ippsDecodeHuffInit\\_BZ2](#) beforehand.



---

## Return Values

<code>ippStsNoErr</code>	Indicates no error.
<code>ippStsNullPtrErr</code>	Indicates an error if one of the pointers is <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error if length of the destination buffer is less than or equal to 0.
<code>ippStsSrcSizeLessExpected</code>	Indicates a warning if size of the source buffer is insufficient to store all output elements.

---

---

# Data Integrity Functions

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for data integrity. These functions implement error-correcting coding, which is featured in [Berl68], [Mor02], and [Bla84].

The data integrity functions are based on operations over finite fields. A field may be defined as a set of elements where

- The results of the addition (+) and multiplication (\*) operations on two arbitrary elements are defined uniquely.
- The associative and commutative laws are applicable to both addition and multiplication, and multiplication is distributive with respect to addition:  $u*(v+w) = u*v + u*w$ .
- An additive identity element 0 and multiplicative identity  $1 \neq 0$  exist.
- Each field element  $u$  has a unique additive inverse element  $-u$ , such that  $u+(-u) = 0$ .
- Each non-zero field element  $u$  has a unique multiplicative inverse element  $1/u$ , such that  $u*(1/u) = 1$ .

For each field element  $u$ , the following equations are true:

$$0+u = u$$

$$1*u = u$$

$$0*u = 0.$$

A field that contains only finitely many elements is called a *finite field* (or Galois field). The order of a finite field is the number of elements  $q$  it contains. It is traditional to denote the finite field of order  $q$  by  $GF(q)$ .

The error-correcting codes implemented in Intel IPP deal with binary finite fields  $GF(2^m)$ , consisting of  $2^m$  possible bit strings of length  $m$ , and essentially use polynomials over  $GF(2^m)$ .

Accordingly, the Intel IPP data integrity functions are divided into groups described in the following sections:

[GF\(2<sup>m</sup>\) Arithmetic Functions](#)

[Arithmetic Functions for Polynomials over GF\(2<sup>m</sup>\)](#)

[Reed-Solomon Code Functions.](#)

## GF(2<sup>m</sup>) Arithmetic Functions

Operations over  $GF(2^m)$  provide a basis for the functionality of Intel IPP for data integrity. The target data type for the data integrity domain is byte. Therefore, the implementation of the  $GF(2^m)$  arithmetic is limited to  $1 < m \leq 8$ .

Table 14-1 below lists the Intel IPP  $GF(2^m)$  arithmetic functions.

**Table 14-1. Intel IPP  $GF(2^m)$  Arithmetic Functions**

Function Base Name	Operation
<a href="#">GFGetSize</a>	Gets the size of the <code>IppsGFSpec_8u</code> context.
<a href="#">GFInit</a>	Initializes user-supplied memory as <code>IppsGFSpec_8u</code> context for future use.
<a href="#">GFAdd</a>	Adds two elements of a finite field.
<a href="#">GFSub</a>	Subtracts one element of a finite field from another.
<a href="#">GFMul</a>	Multiplies two elements of a finite field.
<a href="#">GFDiv</a>	Divides one element of a finite field by another.
<a href="#">GFPow</a>	Raises an element of a finite field to a power.
<a href="#">GFInv</a>	Computes a multiplicative inverse for an element of a finite field.
<a href="#">GFNeg</a>	Computes an additive inverse for an element of a finite field.
<a href="#">GFLogAlpha</a>	Computes the logarithm to the primitive-element base for an element of a finite field.
<a href="#">GFExpAlpha</a>	Raises the primitive element of a finite field to a power.

The functions described in this section use the context of the `IppsGFSpec_8u` type, which carries the definition of the finite field as well as a working buffer sufficient for arithmetic operations over the field.

## GFGetSize

*Gets the size of the `IppsGFSpec_8u` context in bytes.*

### Syntax

```
IppStatus ippsGFGetSize_8u(int feBitSize, int* pSize);
```

### Parameters

<i>feBitSize</i>	Size of the field element (in bits).
<i>pSize</i>	Pointer to the size of the context (in bytes).

## Description

This function is declared in the `ippdi.h` file. The function computes the size of the buffer to be allocated by the application and used in future as the context of the finite field  $\text{GF}(2^{\text{feBitSize}})$ . This context has type `IppsGFSpec_8u` and is necessary for each operation over the finite field.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error condition if the value of the parameter <i>feBitSize</i> is less than 1 or greater than 8.

## GFInit

*Initializes user-supplied memory as  
IppsGFSpec\_8u context for future use.*

---

### Syntax

```
IppStatus ippsgfInit_8u(int feBitSize, const Ipp8u* pPolynomial,  
IppsGFSpec_8u* pGF);
```

### Parameters

<i>feBitSize</i>	Size of the field element (in bits).
<i>pPolynomial</i>	Pointer to the polynomial generating the finite field.
<i>pGF</i>	Pointer to the finite field context to be initialized.

### Description

This function is declared in the `ippdi.h` file. The function initializes the user-supplied buffer as the context of the finite field  $\text{GF}(2^{\text{feBitSize}})$ . The context is necessary for each operation over the field.

The buffer for the context must have size that the function `GFGetSize` returns.

The polynomial is represented as an array of elements having type `Ipp8u` and length  $(\text{feBitSize}+1)$ . In Intel IPP, the lower index of an array corresponds to the lower power of the independent variable. For example, to define the polynomial  $f(x) = x^3 + x + 1$ , use the array `{1,1,0,1}`.



**NOTE.** The value of *feBitSize* for *GFInit* must not exceed the value of the same parameter in the preceding call to *GFGetSize*.

---

### Return Values

<i>ippStsNoErr</i>	Indicates no error. Any other value indicates an error or warning.
<i>ippStsNullPtrErr</i>	Indicates an error condition if any of the specified pointers is NULL.
<i>ippStsRangeErr</i>	Indicates an error condition if the value of the parameter <i>feBitSize</i> is less than 1 or greater than 8.
<i>ippStsBadArgErr</i>	Indicates an error condition if the value of the parameter pointed by <i>pPolynomial</i> is not irreducible.

## GFAdd

*Adds two elements of a finite field.*

---

### Syntax

```
IppStatus ippsgfAdd_8u(Ipp8u gfeA, Ipp8u gfeB, Ipp8u* pR, const IppsGFSpec_8u* pGF);
```

### Parameters

<i>gfeA</i>	The first summand element of the finite field.
<i>gfeB</i>	The second summand element of the finite field.
<i>pR</i>	Pointer to the result of the addition.
<i>pGF</i>	Pointer to the context of the finite field.

### Description

This function is declared in the *ippdi.h* file. The function performs addition over the finite field. The following pseudocode represents this function, provided *gfeR* is the result of the operation:

```
gfeR = gfeA + gfeB.
```

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pGF</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <i>gfeA</i> and/or <i>gfeB</i> is (are) not valid element(s) of the finite field.

## GFSub

*Subtracts one element of a finite field from another.*

---

### Syntax

```
IppStatus ippGFSub_8u(Ipp8u gfeA, Ipp8u gfeB, Ipp8u* pR, const IppsGFSpec_8u* pGF);
```

### Parameters

<i>gfeA</i>	The first (minuend) element of the finite field.
<i>gfeB</i>	The second (subtrahend) element of the finite field.
<i>pR</i>	Pointer to the result of subtraction.
<i>pGF</i>	Pointer to the context of the finite field.

### Description

This function is declared in the `ippdi.h` file. The function performs subtraction over the finite field. The following pseudocode represents this function, provided *gfeR* is the result of the operation:

```
gfeR = gfeA - gfeB.
```

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.

<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <code>pGF</code> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <code>gfeA</code> and/or <code>gfeB</code> is (are) not valid element(s) of the finite field.

## GFMul

*Multiplies two elements of a finite field.*

---

### Syntax

```
IppStatus ippsgfMul_8u(Ipp8u gfeA, Ipp8u gfeB, Ipp8u* pR, const IppsGFSpec_8u* pGF);
```

### Parameters

<code>gfeA</code>	The first (multiplicand) element of the finite field.
<code>gfeB</code>	The second (multiplier) element of the finite field.
<code>pR</code>	Pointer to the result of multiplication.
<code>pGF</code>	Pointer to the context of the finite field.

### Description

This function is declared in the `ippdi.h` file. The function performs multiplication over the finite field. The following pseudocode represents this function, provided `gfeR` is the result of the operation:

```
gfeR = gfeA * gfeB.
```

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <code>pGF</code> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <code>gfeA</code> and/or <code>gfeB</code> is (are) not valid element(s) of the finite field.



## GFDiv

*Divides one element of a finite field by another.*

---

### Syntax

```
IppStatus ippsGFDiv_8u(Ipp8u gfeA, Ipp8u gfeB, Ipp8u* pR, const IppsGFSpec_8u* pGF);
```

### Parameters

<i>gfeA</i>	The first (dividend) element of the finite field.
<i>gfeB</i>	The second (divisor) element of the finite field.
<i>pR</i>	Pointer to the result of division.
<i>pGF</i>	Pointer to context of the finite field.

### Description

This function is declared in the `ippdi.h` file. The function performs division over the finite field. The following pseudocode represents this function, provided *gfeR* is the result of the operation:

$gfeR = gfeA / gfeB$ , if  $gfeB \neq 0$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pGF</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <i>gfeA</i> and/or <i>gfeB</i> is (are) not valid element(s) of the finite field.
<code>ippStsDivByZeroErr</code>	Indicates an error condition if the divisor is zero.

## GFPow

*Raises an element of a finite field to a power.*

---

### Syntax

```
IppStatus ippsGFPow_8u(Ipp8u gfeA, int pow, Ipp8u* pR, const IppsGFSpec_8u* pGF);
```

### Parameters

<i>gfeA</i>	The element of the finite field to be exponentiated.
<i>pow</i>	The exponent.
<i>pR</i>	Pointer to the result of exponentiation.
<i>pGF</i>	Pointer to the context of the finite field.

### Description

This function is declared in the `ippdi.h` file. The function performs exponentiation over the finite field. The following pseudocode represents this function, provided *gfeR* is the result of the operation:

$$gfeR = gfeA^{pow}.$$

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pGF</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <i>gfeA</i> is not a valid element of the finite field.

## GFInv

*Computes the multiplicative inverse for an element of a finite field.*

---

### Syntax

```
IppStatus ippSGFInv_8u(Ipp8u gfeA, Ipp8u* pR, const IppsGFSpec_8u* pGF);
```

### Parameters

<i>gfeA</i>	The element of the finite field to be inverted.
<i>pR</i>	Pointer to the result of the multiplicative inversion.
<i>pGF</i>	Pointer to the context of the finite field.

### Description

This function is declared in the `ippdi.h` file. The function performs the multiplicative inversion over the finite field. The following pseudocode represents this function, provided *gfeR* is the result of the operation:

$gfeR = 1/gfeA$ , so that  $1 = gfeA * gfeR$ , if  $gfeA \neq 0$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pGF</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <i>gfeA</i> is not a valid element of the finite field.
<code>ippStsDivByZeroErr</code>	Indicates an error condition if <i>gfeA</i> is zero.

## GFNeg

*Computes the additive inverse for an element of a finite field.*

---

### Syntax

```
IppStatus ippGFNeg_8u(Ipp8u gfeA, Ipp8u* pR, const IppsGFSpec_8u* pGF);
```

### Parameters

<i>gfeA</i>	The element of the finite field to be inverted.
<i>pR</i>	Pointer to the result of the additive inversion.
<i>pGF</i>	Pointer to the context of the finite field.

### Description

This function is declared in the `ippdi.h` file. The function performs additive inversion over the finite field. The following pseudocode represents this function, provided *gfeR* is the result of the operation:

$gfeR = -gfeA$ , so that  $0 = gfeA + gfeR$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pGF</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <i>gfeA</i> is not a valid element of the finite field.

## GFLogAlpha

*Computes the logarithm to the primitive-element base for an element of a finite field.*

---

### Syntax

```
IppStatus ippSGFLogAlpha_8u(Ipp8u gfeA, Ipp8u* pLog, const IppsGFSpec_8u* pGF);
```

### Parameters

<i>gfeA</i>	The element of the finite field whose logarithm is to be computed.
<i>pLog</i>	Pointer to the resulting logarithm value.
<i>pGF</i>	Pointer to the context of the finite field.

### Description

This function is declared in the `ippdi.h` file. For an element of the finite field, the function computes the logarithm to the base being the primitive element of the field. The following pseudocode represents this function, provided  $i$  is the result of the operation:

$i = \log(gfeA)$ , so that  $gfeA = gf\_primitive\_element^i$ , if  $gfeA \neq 0$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pGF</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <i>gfeA</i> is not a valid element of the finite field.
<code>ippStsBadArgErr</code>	Indicates an error condition if the <i>gfeA</i> is zero.

## GFExpAlpha

*Raises the primitive element of a finite field to a power.*

---

### Syntax

```
IppStatus ippSGFExpAlpha_8u(Ipp8u pow, Ipp8u* pR, const IppsGFSpec_8u* pGF);
```

### Parameters

<i>pow</i>	The exponent.
<i>pR</i>	Pointer to the result of exponentiation.
<i>pGF</i>	Pointer to the context of the finite field.

### Description

This function is declared in the `ippdi.h` file. The function raises the primitive element of the finite field to the power *pow*. The following pseudocode represents this function, provided *gfeR* is the result of the operation:

```
gfeR = (gf_primitive_element)pow.
```

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pGF</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if <i>pow</i> is greater than or equal to the field order.

## Arithmetic Functions for Polynomials over GF(2<sup>m</sup>)

This section considers polynomials over GF(2<sup>m</sup>), which means that coefficients of polynomials belong to GF(2<sup>m</sup>). The arithmetic functions for the polynomials have the same limitation as the GF(2<sup>m</sup>) arithmetic functions: 1<*m*<=8. The operations that these functions implement form the next functionality level of Intel IPP for data integrity.

Table 14-2 below lists the Intel IPP arithmetic functions for polynomials over GF(2<sup>m</sup>).

**Table 14-2. Intel IPP Arithmetic Functions for Polynomials over GF(2<sup>m</sup>)**

Function Base Name	Operation
PolyGFGetSize	Gets the size of the <code>ippsPolyGF_8u</code> context in bytes.
PolyGFInit	Initializes user-supplied memory as the <code>IppsPolyGF_8u</code> context for future use.
PolyGFSetCoeffs	Defines the polynomial.
PolyGFSetDegree	Sets up the polynomial degree.
PolyGFCopy	Copies one polynomial to another.
PolyGFGetRef	Gets references to the polynomial characteristics: degree, coefficients, and the base finite field.
PolyGFAdd	Adds polynomials.
PolyGFSub	Subtracts one polynomial from another.
PolyGFMul	Multiplies polynomials.
PolyGFMod	Reduces a polynomial modulo another polynomial.
PolyGFDiv	Divides polynomials.
PolyGFShlC	Shifts a polynomial left.
PolyGFShrC	Shifts a polynomial right.
PolyGFIrreducible	Tests a given polynomial for being irreducible.
PolyGFPrimitive	Tests a given polynomial for being primitive.
PolyGFValue	Computes the value of a polynomial for a given argument.
PolyGFDerive	Computes the derivate of a given polynomial.
PolyGFRoots	Computes roots of a given polynomial.
PolyGFGCD	Computes the greatest common divisor of two polynomials.

A context of the `IppsPolyGF_8u` type represents each polynomial that the functions described in this section operate on. The context carries not only characteristics of the polynomial but also the definition of the base field GF(2<sup>m</sup>), the maximum possible degree of a polynomial, and a working buffer sufficient for the polynomial arithmetic operations.

PolyGFGetSize

*Gets the size of the `ippsPolyGF_8u` context in bytes.*

Syntax

```
IppStatus ippsPolyGFGetSize_8u(int maxDegree, int* pSize);
```

## Parameters

<i>maxDegree</i>	The highest possible value of the polynomial degree.
<i>pSize</i>	Pointer to the size of the context (in bytes).

## Description

This function is declared in the `ippdi.h` file. The function specifies the size (in bytes) that is required to define a structuralized working buffer of the `IppsPoly_GF8u` context to be used for operations on polynomials.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsbadArgErr</code>	Indicates an error condition if the value of the parameter <i>maxDegree</i> is less than 0.

## PolyGFInit

*Initializes user-supplied memory as  
IppsPolyGF\_8u context for future use.*

---

## Syntax

```
IppStatus ippPolyGFInit_8u(const IppsGFSpec_8u* pGF, int maxDegree,
IppsPoly_GF8u* pPoly);
```

## Parameters

<i>pGF</i>	Pointer to the context of the <i>finite field</i> to be used to define a polynomial.
<i>maxDegree</i>	The highest possible value of the polynomial degree.
<i>pPoly</i>	Pointer to the user-supplied buffer to be initialized as the <code>ippPoly_GF8u</code> context.



Description

This function is declared in the `ippdi.h` file. The function initializes the user-supplied buffer as the `ippsPoly_GF8u` context of the polynomial over the finite field  $GF(2^m)$  specified by the `pGF` parameter. The context is necessary for all operations on a polynomial.

The buffer pointed by `pPoly` must have size that the function `ippsPolyGFGetSize` returns.



**NOTE.** The value of `maxDegree` for `PolyGFInit` must not exceed the value of the same parameter in the preceding call to `PolyGFGetSize`.

Return Values

- `ippStsNoErr` Indicates no error. Any other value indicates an error or warning.
- `ippStsNullPtrErr` Indicates an error condition if any of the specified pointers is `NULL`.
- `ippStsbadArgErr` Indicates an error condition if the value of the parameter `maxDegree` is less than 0.
- `ippStsContextMatchErr` Indicates an error condition if the context pointed by `pGF` is not valid.

PolyGFSetCoeffs

*Defines the polynomial.*

Syntax

```
IppStatus ippsPolyGFSetCoeffs_8u(const Ipp8u* pCoeff, int degree,
IppsPoly_GF8u* pPoly);
```

Parameters

- `pCoeffs` Pointer to the array of values to be used as coefficients of the polynomial.
- `degree` The degree of the polynomial. The number of elements of the array `pCoeffs` to be used as polynomial coefficients is  $(degree+1)$ .

*pPoly* Pointer to the `ippPoly_GF8u` context to be updated with the specified value of *degree* and the set of coefficients.

## Description

This function is declared in the `ippdi.h` file. The function copies (*degree*+1) values from the array pointed by *pCoeffs* to the coefficients of the polynomial *pPoly* and assigns the *degree* value to the degree of the polynomial.

In Intel IPP, the lower index of an array corresponds to the lower power of the independent variable. For example, to define the polynomial  $p(x) = 2x^5 + 17x^2 + 5x + 3$ , use the array {3, 5, 17, 0, 0, 2}.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pPoly</i> is not valid.
<code>ippStsBadArgErr</code>	Indicates an error condition if the value of the parameter <i>degree</i> is less than 0 or greater than the highest possible value specified in the context.

## PolyGFSetDegree

*Sets up the polynomial degree.*

---

### Syntax

```
IppStatus ippPolyGFSetDegree_8u(int degree, IppsPoly_GF8u* pPoly);
```

### Parameters

<i>degree</i>	The value of the polynomial degree to set.
<i>pPoly</i>	Pointer to the <code>ippPoly_GF8u</code> context to be updated with the specified degree.

### Description

This function is declared in the `ippdi.h` file. The function assigns the value of *degree* to the polynomial degree in the context pointed by *pPoly*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pPoly</i> is not valid.
<code>ippStsBadArgErr</code>	Indicates an error condition if the value of the parameter <i>degree</i> is less than 0 or greater than the highest possible value specified in the context.

## PolyGFCopy

*Copies one polynomial to another.*

---

### Syntax

```
IppStatus ippPolyGFCopy_8u(const IppsPoly_GF8u* pSrcPoly, IppsPoly_GF8u* pDstPoly);
```

### Parameters

<i>pSrcPoly</i>	Pointer to the source polynomial.
<i>pDstPoly</i>	Pointer to the destination polynomial.

### Description

This function is declared in the `ippdi.h` file. The function copies the source polynomial to the destination.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.

`ippStsContextMatchErr` Indicates an error condition if any of the specified contexts is not valid.

## PolyGFGetRef

*Gets references to the polynomial parameters.*

---

### Syntax

```
IppStatus ippPolyGFGetRef_8u(Ipp8u** const ppCoeff, int* pDegree,
IppsGFSpec_8u** const ppGF, const IppsPoly_GF8u* pPoly);
```

### Parameters

<i>ppCoeffs</i>	Pointer to the <code>Ipps8u*</code> variable containing the address of the array of polynomial coefficients.
<i>pDegree</i>	Pointer to the actual degree of the polynomial.
<i>ppGF</i>	Pointer to the <code>IppsFDSpec_8u*</code> variable containing the address of the finite field context used for the polynomial definition.
<i>pPoly</i>	Pointer to the polynomial parameters are retrieved from.

### Description

This function is declared in the `ippdi.h` file. The function extracts pointers to the polynomial characteristics: degree, coefficients, and context of the finite field. If any of the target pointers is `NULL`, the appropriate reference is not extracted.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if the pointer to the polynomial context is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pPoly</i> is not valid.

## PolyGFAdd

*Adds polynomials.*

---

### Syntax

```
IppStatus ippsPolyGFAdd_8u(const IppsPoly_GF8u* pSrcA, const IppsPoly_GF8u*
pSrcB, IppsPoly_GF8u* pDstR);
```

### Parameters

<i>pSrcA</i>	Pointer to the first summand polynomial.
<i>pSrcB</i>	Pointer to the second summand polynomial.
<i>pDstR</i>	Pointer to the resulting polynomial.

### Description

This function is declared in the `ippdi.h` file. The function performs polynomial addition.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the specified contexts is not valid.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if $\text{maxDegree}(pDstR) < \text{max}(\text{degree}(pSrcA), \text{degree}(pSrcB))$ , where <i>degree</i> is the actual degree and <i>maxDegree</i> is the highest possible degree specified in the respective polynomial context.

## PolyGFSub

*Subtracts one polynomial from another.*

---

### Syntax

```
IppStatus ippsPolyGFSub_8u(const IppsPoly_GF8u* pSrcA, const IppsPoly_GF8u*
pSrcB, IppsPoly_GF8u* pDstR);
```

## Parameters

<i>pSrcA</i>	Pointer to the first (minued) polynomial.
<i>pSrcB</i>	Pointer to the second (subtrahend) polynomial.
<i>pDstR</i>	Pointer to the resulting polynomial.

## Description

This function is declared in the `ippdi.h` file. The function performs polynomial subtraction.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the specified contexts is not valid.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if $\text{maxDegree}(pDstR) < \text{max}(\text{degree}(pSrcA), \text{degree}(pSrcB))$ , where <i>degree</i> is the actual degree and <i>maxDegree</i> is the highest possible degree specified in the respective polynomial context.

## PolyGFMod

Reduces a polynomial modulo another polynomial.

### Syntax

```
ippStatus ippPolyGFMod_8u(const IppsPoly_GF8u* pPolyA, const IppsPoly_GF8u*
pPolyModulo, IppsPoly_GF8u* pDstR);
```

### Parameters

<i>pPolyA</i>	Pointer to the polynomial to be reduced.
<i>pPolyModulo</i>	Pointer to the modulo polynomial.
<i>pDstR</i>	Pointer to the resulting polynomial.

### Description

This function is declared in the `ippdi.h` file. The function performs reduction of the polynomial  $pPolyA$  modulo  $pPolyModulo$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the specified contexts is not valid.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if $maxDegree(pDstR) < degree(pPolyModulo)$ , where $degree$ is the actual degree and $maxDegree$ is the highest possible degree specified in the respective polynomial context.

## PolyGFMul

*Multiplies polynomials.*

---

### Syntax

```
IppStatus ippPolyGFMul_8u(const IppsPoly_GF8u* pSrcA, const IppsPoly_GF8u*
pSrcB, IppsPoly_GF8u* pDstR);
```

### Parameters

<code>pSrcA</code>	Pointer to the first (multiplicand) polynomial.
<code>pSrcB</code>	Pointer to the second (multiplier) polynomial.
<code>pDstR</code>	Pointer to the resultant polynomial.

### Description

This function is declared in the `ippdi.h` file. The function performs polynomial multiplication.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.

`ippStsContextMatchErr` Indicates an error condition if any of the specified contexts is not valid.

`ippStsOutOfRangeErr` Indicates an error condition if  $\text{maxDegree}(pDstR) < \text{degree}(pSrcA) + \text{degree}(pSrcB)$ , where  $\text{maxDegree}$  is the highest possible degree specified in the respective polynomial context.

## PolyGFDiv

*Divides polynomials.*

---

### Syntax

```
IppStatus ippPolyGFDiv_8u(const IppsPoly_GF8u* pDividend, const
IppsPoly_GF8u* pDivisor, IppsPoly_GF8u* pQuotient, IppsPoly_GF8u* pRemainder);
```

### Parameters

<i>pDividend</i>	Pointer to the dividend polynomial.
<i>pDivisor</i>	Pointer to the divisor polynomial.
<i>pQuotient</i>	Pointer to the quotient polynomial.
<i>pRemainder</i>	Pointer to the remainder polynomial.

### Description

This function is declared in the `ippdi.h` file. The function performs polynomial division.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the specified contexts is not valid.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if $\text{maxDegree}(pQuotient) < \text{degree}(pDividend) - \text{degree}(pDivisor)$ and/or $\text{maxDegree}(pRemainder) < \text{degree}(pDivisor)$ , where $\text{degree}$ is the actual degree and $\text{maxDegree}$ is the highest possible degree specified in the respective polynomial context.



## PolyGFShlC

*Shifts a polynomial left.*

---

### Syntax

```
IppStatus ippsPolyGFShlC_8u(const IppsPoly_GF8u* pSrc, int nShift,
IppsPoly_GF8u* pDst);
```

### Parameters

<i>pSrc</i>	Pointer to the polynomial to be shifted.
<i>nShift</i>	The shift value.
<i>pDstR</i>	Pointer to the resulting polynomial.

### Description

This function is declared in the `ippdi.h` file. The function performs the polynomial operation of left shift or, in other words, multiplication by  $x^{nShift}$ .

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the specified contexts is not valid.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if $maxDegree(pDst) < degree(pSrc) + nShift$ , where <i>degree</i> is the actual degree, and <i>maxDegree</i> is the highest possible degree specified in the respective polynomial context.

## PolyGFShrC

*Shifts a polynomial right.*

---

### Syntax

```
IppStatus ippsPolyGFShrC_8u(const IppsPoly_GF8u* pSrc, int nShift,
IppsPoly_GF8u* pDst);
```

## Parameters

<i>pSrc</i>	Pointer to the polynomial to be shifted.
<i>nShift</i>	The shift value.
<i>pDstR</i>	Pointer to the resulting polynomial.

## Description

This function is declared in the `ippdi.h` file. The function performs the polynomial operation of right shift or, in other words, division by  $x^{nShift}$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the specified contexts is not valid.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if $maxDegree(pDst) < degree(pSrc) - nShift$ , where <i>degree</i> is the actual degree and <i>maxDegree</i> is the highest possible degree specified in the respective polynomial context.

## PolyGFIrreducible

Tests a given polynomial for being irreducible.

### Syntax

```
IppStatus ippPolyGFIrreducible_8u(const IppsPoly_GF8u* pPoly, IppBool* pIsIrreducible);
```

### Parameters

<i>pPoly</i>	Pointer to the polynomial to be tested.
<i>pIsIrreducible</i>	Pointer to the result of the irreducibility test.

### Description

This function is declared in the `ippdi.h` file. The function checks irreducibility of the given polynomial.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the specified context <i>pPoly</i> is not valid.

## PolyGFPrimitive

*Tests a given polynomial for being primitive.*

---

### Syntax

```
ippStatus ippPolyGFPrimitive_8u(const IppsPoly_GF8u* pPoly, IppBool
isIrreducible, IppBool* pIsPrimitive);
```

### Parameters

<i>pPoly</i>	Pointer to the polynomial to be tested.
<i>isIrreducible</i>	The apriory-knowledge flag. If <i>isIrreducible</i> == <code>ippTrue</code> then <i>pPoly</i> is definitely irreducible.
<i>pIsPrimitive</i>	Pointer to the result of the primitivity test

### Description

This function is declared in the `ippdi.h` file. The function checks primitivity of the given polynomial. If *isIrreducible*==`ippTrue`, the function checks irreducibility of the polynomial first.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the specified context <i>pPoly</i> is not valid.

## PolyGFValue

*Computes the value of a polynomial for a given argument.*

---

### Syntax

```
IppStatus ippPolyGFValue_8u(const IppsPoly_GF8u* pPoly, Ipp8u argValue,
Ipp8u* pDstValue);
```

### Parameters

<i>pPoly</i>	Pointer to the polynomial.
<i>argValue</i>	Value of the argument.
<i>pDstValue</i>	Pointer to the computed value of the polynomial.

### Description

This function is declared in the `ippdi.h` file. The function computes the value of the given polynomial for the given value of the argument *argValue*.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the specified context <i>pPoly</i> is not valid.

## PolyGFDerive

*Computes the derivate of a given polynomial.*

---

### Syntax

```
IppStatus ippPolyGFDerive_8u(const IppsPoly_GF8u* pPoly, IppsPoly_GF8u*
pDerivative);
```

### Parameters

<i>pPoly</i>	Pointer to the polynomial.
--------------	----------------------------

*pDerivative* Pointer to the polynomial derivative.

### Description

This function is declared in the `ippdi.h` file. The function computes the formal derivate of the given polynomial.

### Return Values

`ippStsNoErr` Indicates no error. Any other value indicates an error or warning.

`ippStsNullPtrErr` Indicates an error condition if any of the specified pointers is NULL.

`ippStsContextMatchErr` Indicates an error condition if any of the specified contexts is not valid.

`ippStsOutOfRangeErr` Indicates an error condition if  $\text{maxDegree}(pDst) < \text{degree}(pPoly)-1$ , where *degree* is the actual degree and *maxDegree* is the highest possible degree specified in the respective polynomial context.

## PolyGFRoots

Computes roots of a given polynomial.

### Syntax

```
IppStatus ippPolyGFRoots_8u(const IppsPoly_GF8u* pPoly, Ipp8u* pRoot, int* pRootNum);
```

### Parameters

*pPoly* Pointer to the polynomial.

*pRoots* Array of polynomial roots found.

*pRootNum* Pointer to the number of roots found.

### Description

This function is declared in the `ippdi.h` file. The function computes roots of the given polynomial among elements of  $\text{GF}(2^m)$  associated with the polynomial.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the specified context <i>pPoly</i> is not valid.

## PolyGFGCD

*Computes the greatest common divisor of two polynomials.*

---

### Syntax

```
IppStatus ippPolyGFGCD_8u(const IppsPoly_GF8u* pSrcA, const IppsPoly_GF8u* pSrcB, IppsPoly_GF8u* pGCD);
```

### Parameters

<i>pSrcA</i>	Pointer to the first polynomial.
<i>pSrcB</i>	Pointer to the second polynomial.
<i>pDCG</i>	Pointer to the resulting polynomial.

### Description

This function is declared in the `ippdi.h` file. The function computes the largest-degree polynomial that both given polynomials are divisible by.

### Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the specified contexts is not valid.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if $\maxDegree(pDst) < \min(\text{degree}(pSrcA), \text{degree}(pSrcB))$ , where <i>degree</i> is the actual degree and <i>maxDegree</i> is the highest possible degree specified in the respective polynomial context.

## Reed-Solomon Code Functions

This section discusses functions implementing the Reed-Solomon (RS) codes. The section adopts a traditional notation  $RS(n,k)$  for  $n$ -symbol encoded block of  $k$ -symbol user's data. The IPP implementation of the RS functionality is limited to considering only  $m$ -bit symbols ( $m \leq 8$ ). Thus the implementation refers to [Arithmetic Functions for Polynomials over  \$GF\(2^m\)\$](#)  and [Arithmetic Functions](#), described earlier in this chapter.

Intel IPP implements only systematic RS codes, which means that the user's data to be encoded appears verbatim in the encoded codeword.

### RS Encoder Functions

Table 14-3 below lists the Intel IPP RS encoder functions.

**Table 14-3 Intel IPP RS Encoder Functions**

Function Base Name	Operation
<a href="#">RSEncodeGetSize</a>	Gets the size of the <code>ippsRSEncodeSpec_8u</code> context.
<a href="#">RSEncodeInit</a>	Initializes user-supplied memory as the <code>ippsRSEncodeSpec_8u</code> context for future use.
<a href="#">RSEncodeGetBufferSize</a>	Gets the size of a work buffer for the encoding operation.
<a href="#">RSEncode</a>	Performs the RS encoding operation.

The functions described in this section use the  $RS(n,k)$  encoder context having the `ippsRSEncodeSpec_8u` type, which, in addition to the lengths  $n$  and  $k$ , carries the definition of the finite field to be used for encoding and the RS-generating polynomial.

### RSEncodeGetSize

*Gets the size of the `ippsRSEncodeSpec_8u` context in bytes.*

#### Syntax

```
IppStatus ippsRSEncodeGetSize_8u(int codeLength, int dataLength, int* pSize);
```

#### Parameters

<i>codeLength</i>	The desired codeword length.
<i>dataLength</i>	The desired data length.

*pSize* Pointer to the size of the context (in bytes).

## Description

This function is declared in the `ippdi.h` file. The function computes the size of a buffer to be allocated by the application and used in future as the RS (*codeLength*, *dataLength*) encoder context having the `IppsRSEncodeSpec_8u` type.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error condition if values of the parameters <i>codeLength</i> and/or <i>dataLength</i> are out of bounds determined by the inequalities: $2 \leq \text{codeLength} < 256$ ; $0 < \text{dataLength} < \text{codeLength}$ .

## RSEncodeInit

*Initializes user-supplied memory as the IppsRSEncodeSpec\_8u context for future use.*

---

## Syntax

```
IppStatus ippSRSEncodeInit_8u(int codeLength, int dataLength, const
IppsGFSpec_8u* pGF, Ipp8u root, IppsRSEncodeSpec_8u* pRS);
```

## Parameters

<i>codeLength</i>	The desired codeword length.
<i>dataLength</i>	The desired data length.
<i>pGF</i>	Pointer to the context of the finite field (GF) to be used for the encoding operation.
<i>root</i>	The root of the (first) minimal polynomial over GF.
<i>pRS</i>	Pointer to the user-supplied buffer to be initialized as the <code>IppsRSEncodeSpec_8u</code> context.



## Description

This function is declared in the `ippdi.h` file. The function initializes the user-supplied buffer as the `RS(codeLength, dataLength)` encoder context.

The buffer for the context must have size that the function `ippsRSEncodeGetSize` returns.

The `codeLength` and `dataLength` parameters for `ippsRSEncodeInit` must have the same values as in the preceding call to `ippsRSEncodeGetSize`. The `root` value defines the generating polynomial  $g(x)$  of `RS(codeLength, dataLength)`:

$$g(x) = (x - \text{root}) * (x - \text{root} * a^1) * (x - \text{root} * a^2) * \dots * (x - \text{root} * a^{(2^t-1)}),$$

where  $t$  is the error-correcting ability, such that  $2^t = \text{codeLength} - \text{dataLength}$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error condition if values of the parameters <code>codeLength</code> and/or <code>dataLength</code> are out of bounds determined by the inequalities: $2 \leq \text{codeLength} < \text{orderof}(\text{GF})$ ; $0 < \text{dataLength} < \text{codeLength}$ .
<code>ippStsBadArgErr</code>	Indicates an error condition if the value of the parameter <code>root</code> is not a valid element of the finite field or is equal to 0.

## RSEncodeGetBufferSize

*Gets the size of a work buffer for the encoding operation.*

---

### Syntax

```
IppStatus ippsRSEncodeGetBufferSize_8u(const IppsRSEncodeSpec_8u* pRS, int* pSize);
```

### Parameters

<code>pRS</code>	Pointer to the RS encoder context.
<code>pSize</code>	Pointer to the size of the work buffer (in bytes).

## Description

This function is declared in the `ippdi.h` file. The function computes the size of a buffer to be allocated by the application and used in future as the work buffer for the RS encoder that the context pointed by `pRS` defines.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <code>pRS</code> is not valid.

## RSEncode

*Performs the RS encoding operation.*

---

## Syntax

```
IppStatus ippSRSEncode_8u(const Ipp8u* pMsg, Ipp8u* pCodeWord, const
IppsRSEncodeSpec_8u* pRS, Ipp8u* pBuffer);
```

## Parameters

<code>pMsg</code>	Pointer to the message to be encoded.
<code>pCodeWord</code>	Pointer to the output codeword.
<code>pRS</code>	Pointer to the RS encoder context.
<code>pBuffer</code>	Pointer to the work buffer.

## Description

This function is declared in the `ippdi.h` file. The function performs RS encoding that the context pointed by `pRS` defines for an input message pointed by `pMsg` and stores the systematic code at the address of `pCodeWord`.

The work buffer pointed by `pBuffer` must have size not less than the function [ippSRSEncodeGet-BufferSize](#) returns.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
--------------------------	--

<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pRS</i> is not valid.

RS Decoder Functions

Table 14-4 lists the Intel IPP RS decoder functions.

Table 14-4. Intel IPP RS Decoder Functions

Function Base Name	Operation
<code>RSDecodeGetSize</code>	Gets the size of the <code>ippRSDecodeSpec_8u</code> context.
<code>RSDecodeInit</code>	Initializes user-supplied memory as the <code>IppsRSDecodeSpec_8u</code> context for future use.
<code>RSDecodeBMGetBufferSize</code>	Gets the size of a work buffer for the decoding operation that uses the Berlekamp-Massey (BM) decoding algorithm.
<code>RSDecodeEEGetBufferSize</code>	Gets the size of a work buffer for the decoding operation that uses the Extended Euclidean (EE) decoding algorithm.
<code>RSDecodeBM</code>	Performs the RS decoding operation that uses the BM decoding algorithm.
<code>RSDecodeEE</code>	Performs the RS decoding operation that uses the EE decoding algorithm.

The functions described in this section use the RS(*n*,*k*) decoder context having the `IppsRSDecodeSpec_8u` type, which, in addition to the lengths *n* and *k*, carries the definition of the finite field to be used for decoding and roots of the RS-generating polynomial.

RSDecodeGetSize

Gets the size of the `ippRSDecodeSpec_8u` context in bytes.

Syntax

```
IppStatus ippRSDecodeGetSize_8u(int codeLength, int dataLength, int* pSize);
```

Parameters

*codeLength*                      The desired codeword length.

<i>dataLength</i>	The desired data length.
<i>pSize</i>	Pointer to the size of the context (in bytes).

## Description

This function is declared in the `ippdi.h` file. The function computes the size of a buffer to be allocated by the application and used in future as the `RS(codeLength, dataLength)` decoder context having the `IppsRSDecodeSpec_8u` type.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error condition if values of the parameters <i>codeLength</i> and/or <i>dataLength</i> are out of bounds determined by the inequalities: $2 \leq \textit{codeLength} < 256$ ; $0 < \textit{dataLength} < \textit{codeLength}$ .

## RSDecodeInit

*Initializes user-supplied memory as the IppsRSDecodeSpec\_8u context for future use.*

---

## Syntax

```
IppStatus ippRSDecodeInit_8u(int codeLength, int dataLength, const
IppsGFSpec_8u* pGF, Ipp8u root, IppsRSDecodeSpec_8u* pRS);
```

## Parameters

<i>codeLength</i>	The desired codeword length.
<i>dataLength</i>	The desired data length.
<i>pGF</i>	Pointer to the finite field (GF) context to be used for the decoding operation.
<i>root</i>	The root of the (first) minimal polynomial over GF.
<i>pRS</i>	Pointer to the user-supplied buffer to be initialized as the <code>IppsRSDecodeSpec_8u</code> context.

## Description

This function is declared in the `ippdi.h` file. The function initializes the user-supplied buffer as the `RS(codeLength, dataLength)` decoder context.

The buffer for the context must have size that the function `ippsRSDecodeGetSize` returns.

The `codeLength` and `dataLength` parameters for `ippsRSDecodeInit` must have the same values as in the preceding call to `ippsRSDecodeGetSize`. The `root` value defines the generating polynomial  $g(x)$  of `RS(codeLength, dataLength)`:

$$g(x) = (x - \text{root}) * (x - \text{root} * a^1) * (x - \text{root} * a^2) * \dots * (x - \text{root} * a^{(2^t-1)}),$$

where  $t$  is the error-correcting ability, such that  $2^t = \text{codeLength} - \text{dataLength}$ .

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsRangeErr</code>	Indicates an error condition if values of the parameters <code>codeLength</code> and/or <code>dataLength</code> are out of bounds determined by the inequalities: $2 \leq \text{codeLength} < \text{orderof}(\text{GF})$ ; $0 < \text{dataLength} < \text{codeLength}$ .
<code>ippStsBadArgErr</code>	Indicates an error condition if the value of the parameter <code>root</code> is not a valid element of the finite field or is equal to 0.

## RSDecodeBMGetBufferSize, RSDecodeEEGetBufferSize

*Gets size of work buffer for the decoding operation.*

### Syntax

```
ippStatus ippsRSDecodeBMGetBufferSize_8u(const IppsRSDecodeSpec_8u* pRS,
int* pSize);
```

```
ippStatus ippsRSDecodeEEGetBufferSize_8u(const IppsRSDecodeSpec_8u* pRS,
int* pSize);
```

### Parameters

`pRS`                      Pointer to the RS decoder context.

*pSize* Pointer to the size of the work buffer (in bytes).

## Description

This function is declared in the `ippdi.h` file. The function computes the size of a buffer to be allocated by the application and used in future as the work buffer for the RS decoder that the context pointed by *pRS* defines. The functions `ippsRSDecodeBMGetBufferSize` and `ippsRSDecodeEEGetBufferSize` compute the size of the work buffer for the Berlekamp-Massey (BM) and Extended Euclidean (EE) decoding algorithms, respectively.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pRS</i> is not valid.

## RSDecodeBM, RSDecodeEE

*Performs the RS decoding operation.*

---

## Syntax

```
IppStatus ippsRSDecodeBM_8u(const int* pErasureList, int erasureListLength,
Ipp8u* pCodeWord, const IppsRSDecodeSpec_8u* pRS, Ipp8u* pBuffer);

IppStatus ippsRSDecodeEE_8u(const int* pErasureList, int erasureListLength,
Ipp8u* pCodeWord, const IppsRSDecodeSpec_8u* pRS, Ipp8u* pBuffer);
```

## Parameters

<i>pErasureList</i>	Pointer to the array with the list of erasure locations.
<i>erasureListLength</i>	Length of the list of erasure locations.
<i>pCodeWord</i>	Pointer to the codeword to be decoded.
<i>pRS</i>	Pointer to the RS decoder context.
<i>pBuffer</i>	Pointer to the work buffer.

## Description

This function is declared in the `ippdi.h` file. The function performs RS decoding defined by the context *pRS* for the systematic code in the given codeword and stores the result back at the address of *pCodeWord*. The RS decoders `ippsRSDecodeBM` and `ippsRSDecodeEE` implement the Berlekamp-Massey (BM) and Extended Euclidean (EE) decoding algorithms, respectively.

The work buffer pointed by *pBuffer* must have size not less than the respective function `ippsRSDecodeBMGetBufferSize` or `ippsRSDecodeEEGetBufferSize` returns.

## Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is NULL.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context pointed by <i>pRS</i> is not valid.
<code>ippStsBadArgErr</code>	Indicates an error condition if any element of the array pointed by <i>pErasureList</i> is not valid, that is, negative or exceeding the <i>codeLength</i> value specified in the context.

---

---



# Handling of Special Cases

Some mathematical functions implemented in Intel IPP are not defined for all possible argument values. This appendix describes how the corresponding Intel IPP functions used in signal processing domains handle situations when their input arguments fall outside the range of function definition or may lead to ambiguously determined output results.

Table A-1 below summarizes these special cases for general vector functions described in chapter 5 [Essential Functions](#) and lists result values together with status codes returned by these functions. The status codes ending with `Err` (except for the `ippStsNoErr` status) indicate an error. When an error occurs, the function execution is interrupted. All other status codes indicate that the input argument is outside the range, but the function execution is continued with the corresponding result value.

**Table A-1. Special Cases for Intel IPP Signal Processing Functions**

Function Base Name	Data Type	Case Description	Result Value	Status Code
<a href="#">Sqrt</a>	16s	Sqrt ( $x < 0$ )	0	<code>ippStsSqrtNegArg</code>
	32f	Sqrt ( $x < 0$ )	<code>NAN_32F</code>	<code>ippStsSqrtNegArg</code>
	64s	Sqrt ( $x < 0$ )	0	<code>ippStsSqrtNegArg</code>
	64f	Sqrt ( $x < 0$ )	<code>NAN_64F</code>	<code>ippStsSqrtNegArg</code>
<a href="#">Div, Div_Round</a>	8u	Div (0/0)	0	<code>ippStsDivByZero</code>
		Div ( $x/0$ )	<code>IPP_MAX_8U</code>	<code>ippStsDivByZero</code>
	16s	Div (0/0)	0	<code>ippStsDivByZero</code>
		Div ( $x/0$ ), $x > 0$	<code>IPP_MAX_16S</code>	<code>ippStsDivByZero</code>
		Div ( $x/0$ ), $x < 0$	<code>IPP_MIN_16S</code>	<code>ippStsDivByZero</code>
	16sc	Div (0/0)	0	<code>ippStsDivByZero</code>
		Div ( $x/0$ )	0	<code>ippStsDivByZero</code>
	32f	Div (0/0)	<code>NAN_32F</code>	<code>ippStsDivByZero</code>
		Div ( $x/0$ ), $x > 0$	<code>INF_32F</code>	<code>ippStsDivByZero</code>

Function Base Name	Data Type	Case Description	Result Value	Status Code
DivC	32fc	Div (x/0), x<0	INF_NEG_32F	ippStsDivByZero
		Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0)	NAN_32F	ippStsDivByZero
	64f	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0), x>0	INF_32F	ippStsDivByZero
		Div (x/0), x<0	INF_NEG_32F	ippStsDivByZero
	64fc	Div (0/0)	NAN_32F	ippStsDivByZero
		Div (x/0)	NAN_32F	ippStsDivByZero
	all	Div(x/0)	-	ippStsDivByZeroErr
Ln	16s	Ln (0)	IPP_MIN_16S	ippStsLnZeroArg
		Ln (x<0)	IPP_MIN_16S	ippStsLnZeroArg
	32s	Ln (0)	IPP_MIN_32S	ippStsLnZeroArg
		Ln (x<0)	IPP_MIN_32S	ippStsLnNegArg
	32f	Ln (x<0)	NAN_32F	ippStsLnNegArg
		Ln(x<IPP_MINABS_32F)	INF_NEG_32F	ippStsLnZeroArg
	64f	Ln (x<0)	NAN_64F	ippStsLnNegArg
		Ln(x<IPP_MINABS_64F)	INF_NEG_64F	ippStsLnZeroArg
Exp	16s	overflow	IPP_MAX_16S	ippStsNoErr
	32s	overflow	IPP_MAX_32S	ippStsNoErr

Function Base Name	Data Type	Case Description	Result Value	Status Code
Threshold, Threshold_LT, Threshold_GT, Threshold_LTVal, Threshold_GTVal	64s	overflow	IPP_MAX_64S	ippStsNoErr
	32f	overflow	INF_32F	ippStsNoErr
	64f	overflow	INF_64F	ippStsNoErr
	16sc	level < 0	-	ippStsThreshNegLevelErr
	32fc	level < 0	-	ippStsThreshNegLevelErr
	64fc	level < 0	-	ippStsThreshNegLevelErr
	Threshold_LTInv	32f	level < 0	ippStsThreshNegLevelErr
		level = 0, x=0	INF_32F	ippStsInvZero
	32fc	level < 0	-	ippStsThreshNegLevelErr
		level = 0, x=0	INF_32F	ippStsInvZero
10Log10	64f	level < 0	-	ippStsThreshNegLevelErr
		level = 0, x=0	INF_64F	ippStsInvZero
	64fc	level < 0	-	ippStsThreshNegLevelErr
		level = 0, x=0	INF_64F	ippStsInvZero
10Log10	32s	10Log10 (0)	IPP_MIN_32S	ippStsLnZeroArg
		10Log10 (x<0)	IPP_MIN_32S	ippStsLnNegArg

Here x denotes an input value. For the definition of the constants used, see [Data Ranges](#) in chapter 2.

Note that flavors of the same math function operating on different data types may produce different results for equal argument values. However, for a given function and a fixed data type, handling of special cases is the same for all function flavors that have different [descriptors](#) in their names. For example, the logarithm function `ippiLn` operating on 16s data treats zero argument values in the same way for all its flavors `ippsLn_16s_Sfs` and `ippiLn_16s_ISfs`.

Table A-2 below summarizes special cases for fixed-accuracy arithmetic functions.

**Table A-2. Special Cases for Intel IPP Fixed-Accuracy Arithmetic Functions**

Function Base Name	Data Type	Case Description	Result Value	Status Code
Inv	32f	Inv (x=+0)	INF_32F	ippStsSingularity
		Inv (x=-0)	-INF_32F	ippStsSingularity
	64f	Inv (x=+0)	INF_64F	ippStsSingularity
		Inv (x=-0)	-INF_64F	ippStsSingularity
Div	32f	Div (x>0, y=+0)	INF_32F	ippStsSingularity
		Div (x>0, y=-0)	-INF_32F	ippStsSingularity
		Div (x<0, y=+0)	INF_32F	ippStsSingularity
		Div (x<0, y=-0)	-INF_32F	ippStsSingularity
		Div (x=0, y=0)	NAN_32F	ippStsSingularity
	64f	Div (x>0, y=+0)	INF_64F	ippStsSingularity
		Div (x>0, y=-0)	-INF_64F	ippStsSingularity
		Div (x<0, y=+0)	INF_64F	ippStsSingularity
		Div (x<0, y=-0)	-INF_64F	ippStsSingularity
		Div (x=0, y=0)	NAN_64F	ippStsSingularity
Sqrt	32f	Sqrt(x<0)	NAN_32F	ippStsDomain
		Sqrt(x=-INF)	NAN_32F	ippStsDomain
	64f	Sqrt(x<0)	NAN_64F	ippStsDomain
		Sqrt(x=-INF)	NAN_64F	ippStsDomain
InvSqrt	32f	InvSqrt (x<0)	NAN_32F	ippStsDomain
		InvSqrt (x=+0)	INF_32F	ippStsSingularity
		InvSqrt (x=-0)	-INF_32F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_32F	ippStsDomain
	64f	InvSqrt (x<0)	NAN_64F	ippStsDomain

Function Base Name	Data Type	Case Description	Result Value	Status Code
		InvSqrt (x=+0)	INF_64F	ippStsSingularity
		InvSqrt (x=-0)	-INF_64F	ippStsSingularity
		InvSqrt (x=-INF)	NAN_64F	ippStsDomain
InvCbrt	32f	InvCbrt (x=+0)	INF_32F	ippStsSingularity
		InvCbrt (x=-0)	-INF_32F	ippStsSingularity
	64f	InvCbrt (x=+0)	INF_64F	ippStsSingularity
		InvCbrt (x=-0)	-INF_64F	ippStsSingularity
Pow3o2	32f	Pow3o2 (x<0)	NAN_32F	ippStsDomain
		Pow3o2 (x=-INF)	NAN_32F	ippStsDomain
	64f	Pow3o2 (x<0)	NAN_64F	ippStsDomain
		Pow3o2 (x=-INF)	NAN_64F	ippStsDomain
Pow, Powx	32f	Pow (x=+0, y=-ODD_INT)	INF_32F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_32F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_32F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_32F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_32F	ippStsDomain
		Pow (x=0, y=-INF)	INF_32F	ippStsSingularity
	64f	Pow (x=+0, y=-ODD_INT)	INF_64F	ippStsSingularity
		Pow (x=-0, y=-ODD_INT)	-INF_64F	ippStsSingularity
		Pow (x=0, y=-EVEN_INT)	INF_64F	ippStsSingularity
		Pow (x=0, y=NON_INT_NEG)	INF_64F	ippStsSingularity
		Pow (x<0, y=NON_INT_POS)	NAN_64F	ippStsDomain
		Pow (x=0, y=-INF)	INF_64F	ippStsSingularity
Exp	32f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_32F	ippStsOverflow

Function Base Name	Data Type	Case Description	Result Value	Status Code
Expml	64f	Exp (x), x<underflow	0	ippStsUnderflow
		Exp (x), x>overflow	INF_64F	ippStsOverflow
	32f	Expml(x), x>overflow	INF_32F	ippStsOverflow
	64f	Expml(x), x>overflow	INF_64F	ippStsOverflow
Ln, Log10	32f	Ln(x<0)	NAN_32F	ippStsDomain
		Ln(x=-INF)	NAN_32F	ippStsDomain
		Ln(x=0)	-INF_32F	ippStsSingularity
	64f	Ln(x<0)	NAN_64F	ippStsDomain
		Ln(x=-INF)	NAN_64F	ippStsDomain
		Ln(x=0)	-INF_64F	ippStsSingularity
Loglp	32f	Ln(x<-1)	NAN_32F	ippStsDomain
		Ln(x=-INF)	NAN_32F	ippStsDomain
		Ln(x=-1)	-INF_32F	ippStsSingularity
	64f	Ln(x<-1)	NAN_64F	ippStsDomain
		Ln(x=-INF)	NAN_64F	ippStsDomain
		Ln(x=-1)	-INF_64F	ippStsSingularity
Cos	32f	Cos(INF)	NAN_32F	ippStsDomain
	64f	Cos(INF)	NAN_64F	ippStsDomain
Sin	32f	Sin(INF)	NAN_32F	ippStsDomain
	64f	Sin(INF)	NAN_64F	ippStsDomain
SinCos	32f	SinCos(INF)	NAN_32F, NAN_32F	ippStsDomain
	64f	SinCos(INF)	NAN_64F, NAN_64F	ippStsDomain
CIS	32fc	CIS(INF)	NAN_32F, NAN_32F	ippStsDomain
	64fc	CIS(INF)	NAN_64F, NAN_64F	ippStsDomain

Function Base Name	Data Type	Case Description	Result Value	Status Code
Tan	32f	Tan(INF)	NAN_32F	ippStsDomain
	64f	Tan(INF)	NAN_64F	ippStsDomain
Acos	32f	Acos(x),  x >1	NAN_32F	ippStsDomain
		Acos(INF)	NAN_32F	ippStsDomain
	64f	Acos(x),  x >1	NAN_64F	ippStsDomain
		Acos(INF)	NAN_64F	ippStsDomain
Asin	32f	Asin(x),  x >1	NAN_32F	ippStsDomain
		Asin(INF)	NAN_32F	ippStsDomain
	64f	Asin(x),  x >1	NAN_64F	ippStsDomain
		Asin(INF)	NAN_64F	ippStsDomain
Cosh	32f	Cosh(x),  x >overflow	INF_32F	ippStsOverflow
	64f	Cosh(x),  x >overflow	INF_64F	ippStsOverflow
Sinh	32f	Sinh(x),  x >overflow	INF_32F	ippStsOverflow
	64f	Sinh(x),  x >overflow	INF_64F	ippStsOverflow
Acosh	32f	Acosh(x<1)	NAN_32F	ippStsDomain
		Acosh(x=-INF)	NAN_32F	ippStsDomain
	64f	Acosh(x<1)	NAN_64F	ippStsDomain
		Acosh(x=-INF)	NAN_64F	ippStsDomain
Atanh	32f	Atanh(x=1)	INF_32F	ippStsSingularity
		Atanh(x=-1)	-INF_32F	ippStsSingularity
		Atanh(x),  x >1	NAN_32F	ippStsDomain
		Atanh(INF)	NAN_32F	ippStsDomain
	64f	Atanh(x=1)	INF_64F	ippStsSingularity
		Atanh(x=-1)	-INF_64F	ippStsSingularity

Function Base Name	Data Type	Case Description	Result Value	Status Code
		$\text{Atanh}(x),  x  > 1$	NAN_64F	ippStsDomain
		$\text{Atanh}(\text{INF})$	NAN_64F	ippStsDomain
Erfc	32f	$\text{Erfc}(x),  x  > \text{underflow}$	0	ippStsUnderflow
	64f	$\text{Erfc}(x),  x  > \text{underflow}$	0	ippStsUnderflow
ErfInv	32f	$\text{ErfInv}(x=1)$	INF_32F	ippStsSingularity
		$\text{ErfInv}(x=-1)$	-INF_32F	ippStsSingularity
		$\text{ErfInv}(x),  x  > 1$	NAN_32F	ippStsDomain
		$\text{ErfInv}(\text{INF})$	NAN_32F	ippStsDomain
	64f	$\text{ErfInv}(x=1)$	INF_64F	ippStsSingularity
		$\text{ErfInv}(x=-1)$	-INF_64F	ippStsSingularity
		$\text{ErfInv}(x),  x  > 1$	NAN_64F	ippStsDomain
		$\text{ErfInv}(\text{INF})$	NAN_64F	ippStsDomain
ErfcInv	32f	$\text{ErfcInv}(x=2)$	-INF_32F	ippStsSingularity
		$\text{ErfcInv}(x=0)$	INF_32F	ippStsSingularity
		$\text{ErfcInv}(x),  x  < 0$	NAN_32F	ippStsDomain
		$\text{ErfcInv}(x),  x  > 2$	NAN_32F	ippStsDomain
		$\text{ErfcInv}(\text{INF})$	NAN_32F	ippStsDomain
	64f	$\text{ErfcInv}(x=2)$	-INF_64F	ippStsSingularity
		$\text{ErfcInv}(x=0)$	INF_64F	ippStsSingularity
		$\text{ErfcInv}(x),  x  < 0$	NAN_64F	ippStsDomain
		$\text{ErfcInv}(x),  x  > 2$	NAN_64F	ippStsDomain
		$\text{ErfcInv}(\text{INF})$	NAN_64F	ippStsDomain
CdfNorm	32f	$\text{CdfNorm}(x),  x  < \text{underflow}$	0	ippStsUnderflow
	64f	$\text{CdfNorm}(x),  x  < \text{underflow}$	0	ippStsUnderflow
CdfNormInv	32f	$\text{CdfNormInv}(x=1)$	INF_32F	ippStsSingularity

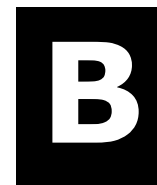


Function Base Name	Data Type	Case Description	Result Value	Status Code
64f		CdfNormInv(x=0)	-INF_32F	ippStsSingularity
		CdfNormInv(x), x<0	NAN_32F	ippStsDomain
		CdfNormInv(x), x>1	NAN_32F	ippStsDomain
		CdfNormInv(INF)	NAN_32F	ippStsDomain
		CdfNormInv(x=1)	INF_64F	ippStsSingularity
		CdfNormInv(x=0)	-INF_64F	ippStsSingularity
		CdfNormInv(x), x<0	NAN_64F	ippStsDomain
		CdfNormInv(x), x>1	NAN_64F	ippStsDomain
		CdfNormInv(INF)	NAN_64F	ippStsDomain

---

---

# Removed Functions



This appendix contains Table B-1 that lists the functions that have been removed from the Intel IPP version 5.0. If an application created with the previous versions calls a function listed here, then the source code must be modified. The table specifies the corresponding Intel IPP 5.0 functions to replace the removed functions.

**Table B-1 Removed Functions**

Removed from 5.0	Substitution or Workaround	Comments
<b>Audio Coding Fucntions</b>		
ippsApplySF_32f_I	ippsScale_32f_I	
ippsBuildHDT_32s	ippsVLCDecodeInit_32s	Data Compression domain
ippsBuildHET_16s	ippsVLCEncodeInit_32s	Data Compression domain
ippsBuildHET_VLC_32s	ippsVLCEncodeInit_32s	Data Compression domain
ippsCdbkFree_VQ_32f	ippsVQCodeBookFree_32f	
ippsCdbkInitAlloc_VQ_32f	ippsVQCodeBookInitAlloc_32f	
ippsCountBits_1tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsCountBits_2tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsCountBits_4tuple_VLC_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsDecodeVLC_32s	ippsVLCDecodeOne_1u16s	Data Compression domain
ippsDecodeVLC_AACESCBLOCK_32s	ippsVLCDe- codeEscBlock_AAC_1u16s	

Removed from 5.0	Substitution or Workaround	Comments
ippsDecodeVLC_Block_32s	ippsVLCDecodeBlock_1u16s	Data Compression domain
ippsDecodeVLC_MP3ESCBBlock_32s	ippsVLCDe- codeEscBlock_MP3_1u16s	
ippsEncodeBlock_1tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeBlock_2tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeBlock_4tuple_VLC_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeVLC_Block_16s	ippsVLCEncodeBlock_16s1u	Data Compression domain
ippsEncodeVLC_MP3ESCBBlock_16s	ippsVLCEn- codeEscBlock_MP3_16s1u	
ippsFDPInv_32f	ippsFDPInv_32f_I	
ippsGetSizeHDT_32s	ippsVLCDecodeGetSize_32s	Data Compression domain
ippsGetSizeHET_16s	ippsVLCEncodeGetSize_32s	Data Compression domain
ippsGetSizeHET_VLC_32s	ippsVLCEncodeGetSize_32s	Data Compression domain
ippsHuffmanCountBits_16s	ippsVLCCountBits_16s32s	Data Compression domain
ippsIndexSelect_VQ_32f	ippsVQIndexSelect_32f	Data Compression domain
ippsMainSelect_VQ_32f	ippsVQMainSelect_32f	Data Compression domain
ippsPreSelect_VQ_32f	ippsVQPreliminarySelect_32f	
ippsResetFDP_32f	ippsFDPReset_32f	
ippsResetFDP_SFB_32f	ippsFDPResetSfb_32f	
ippsResetFDPGroup_32f	ippsFDPResetGroup_32f	
ippsVectorReconstruction_VQ_32f	ippsVQReconstruction_32f	

Removed from 5.0	Substitution or Workaround	Comments
Commom Functions		
ippCoreGetCpuClocks	ippGetCpuClocks	
ippCoreGetCpuType	ippGetCpuType	
ippCoreGetStatusString	ippGetStatusString	
ippCoreSetDenormAreZeros	ippSetDenormAreZeros	
ippCoreSetFlushToZero	ippSetFlushToZero	
ippStaticFree		useless, no memory allocation
ippStaticInitBest	ippStaticInit	
Speech Recognition Functions		
ippsAddMulCol_64f_D2L	ippsAddMulColumn_64f_D2L	
ippsAddNRow_32f_D2	ippsAddNRows_32f_D2	
ippsCoefUpdateAECNLMS_32fc_I		
ippsCoefUpdateAECNLMS_32sc_I		
ippsControllerGetSizeAEC_32f		
ippsControllerGetSizeAEC_32s		
ippsControllerInitAEC_32f		
ippsControllerInitAEC_32s		
ippsControllerUpdateAEC_32f		
ippsControllerUpdateAEC_32s		
ippsCopyCol_32f_D2	ippsCopyColumn_32f_D2	
ippsDeltaDeltaW1_32f_D2	ippsDeltaDelta_Win1_32f_D2	
ippsDeltaDeltaW2_32f_D2	ippsDeltaDelta_Win2_32f_D2	
ippsDeltaW1_32f_D2	ippsDelta_Win1_32f_D2	
ippsDeltaW2_32f_D2	ippsDelta_Win2_32f_D2	
ippsDotProdCol_64f_D2L	ippsDotProdColumn_64f_D2L	

Removed from 5.0	Substitution or Workaround	Comments
ippsFilterAECNLMS_32fc		
ippsFilterAECNLMS_32sc_Sfs		
ippsLMThreshold_32f	ippsScaleLM_32f	
ippsLogGauss_16s32f_D2	ippsLog-	
ippsLogGauss_16s32f_D2L	Gauss_Scaled_16s32f_D2	
ippsLogGauss_IdVar_16s32f_D2	ippsLog-	
ippsLogGauss_IdVar_16s32f_D2L	Gauss_Scaled_16s32f_D2L	
ippsLogGauss_IdVar-	ippsLogGauss_Id-	
Low_16s32f_D2	VarScaled_16s32f_D2	
ippsLogGauss_IdVar-	ippsLogGauss_Id-	
Low_16s32f_D2L	VarScaled_16s32f_D2L	
ippsLogGauss_Low_16s32f_D2	ippsLogGauss_IdVarLowS-	
ippsLogGauss_Low_16s32f_D2L	caled_16s32f_D2	
ippsLogGauss1_32f_D2	ippsLogGauss_IdVarLowS-	
ippsLogGauss1_32f_D2L	caled_16s32f_D2L	
ippsLogGauss1_64f_D2	ippsLogGauss_LowS-	
ippsLogGauss1_64f_D2L	caled_16s32f_D2	
ippsLogGauss2_32f_D2	ippsLogGauss_LowS-	
ippsLogGauss2_32f_D2L	caled_16s32f_D2L	
ippsLogGauss2_64f_D2	ippsLogGauss_32f_D2	
ippsLogGauss2_64f_D2L	ippsLogGauss_32f_D2L	
ippsLogGaussAdd_16s32f_D2	ippsLogGauss_64f_D2	
	ippsLogGauss_64f_D2L	
	ippsLogGaussAddMultiM-	
	ix_32f_D2	
	ippsLogGaussAddMultiM-	
	ix_32f_D2L	
	ippsLogGaussAddMultiM-	
	ix_64f_D2	
	ippsLogGaussAddMultiM-	
	ix_64f_D2L	

Removed from 5.0	Substitution or Workaround	Comments
	ippsLogGaussAdd_Scaled_16s32f_D2	
ippsLogGaussAdd_16s32f_D2L	ippsLogGaussAdd_Scaled_16s32f_D2L	
ippsLogGaussAdd_Id-Var_16s32f_D2	ippsLogGaussAdd_Id-VarScaled_16s32f_D2	
ippsLogGaussAdd_Id-Var_16s32f_D2L	ippsLogGaussAdd_Id-VarScaled_16s32f_D2L	
ippsLogGaussAdd1_32f_D2	ippsLogGaussAdd_32f_D2	
ippsLogGaussAdd2_32f_D2	ippsLogGaussAddMultiM-ix_32f_D2	
ippsLogGaussAddMultiM-ix_16s32f_D2	ippsLogGaussAddMultiM-ix_Scaled_16s32f_D2	
ippsLogGaussAddMultiM-ix_16s32f_D2L	ippsLogGaussAddMultiM-ix_Scaled_16s32f_D2L	
ippsLogGaussMax_16s32f_D2		
ippsLogGaussMax_16s32f_D2L	ippsLogGaussMax_Scaled_16s32f_D2	
ippsLogGaussMax_Id-Var_16s32f_D2	ippsLogGaussMax_Scaled_16s32f_D2L	
ippsLogGaussMax_Id-Var_16s32f_D2L	ippsLogGaussMax_Id-VarScaled_16s32f_D2	
ippsLogGaussMax1_32f_D2	ippsLogGaussMax_Id-VarScaled_16s32f_D2L	
ippsLogGaussMaxMultiM-ix_16s32f_D2	ippsLogGaussMax_32f_D2	
ippsLogGaussMaxMultiM-ix_16s32f_D2L	ippsLogGaussMaxMultiM-ix_Scaled_16s32f_D2	
ippsLogGaussMultiM-ix_16s32f_D2	ippsLogGaussMaxMultiM-ix_Scaled_16s32f_D2L	
ippsLogGaussMultiM-ix_16s32f_D2L	ippsLogGaussMultiM-ix_Scaled_16s32f_D2	
ippsLogGaussMultiM-ix_Low_16s32f_D2	ippsLogGaussMultiM-ix_Scaled_16s32f_D2L	

Removed from 5.0	Substitution or Workaround	Comments
ippsLogGaussMultiMix_LowScaled_16s32f_D2L	ippsLogGaussMultiMix_LowScaled_16s32f_D2	
ippsLogGaussSingle_16s32f	ippsLogGaussMultiMix_LowScaled_16s32f_D2L	
ippsLogGaussSingle_BlockDVar_16s32f	ippsLogGaussSingle_Scaled_16s32f	
ippsLogGaussSingle_DirectVar_16s32	ippsLogGaussSingle_BlockDVarScaled_16s32f	
	ippsLogGaussSingle_DirectVarScaled_16s32f	
ippsLogGaussSingle_IdVar_16s32f	ippsLogGaussSingle_IdVarScaled_16s32f	
ippsLogGaussSingle_IdVarLow_16s32f	ippsLogGaussSingle_IdVarLowScaled_16s32f	
ippsLogGaussSingle_Low_16s32f	ippsLogGaussSingle_LowScaled_16s32f	
ippsLPToLSP_16s_D2Sfs	ippsLPToLSP_16s_Sfs	
ippsLPToLSP_32f_D2	ippsLPToLSP_32f	
ippsLSPToLP_16s_D2Sfs	ippsLSPToLP_16s_Sfs	
ippsLSPToLP_32f_D2	ippsLSPToLP_32f	
ippsMahDist1_32f_D2	ippsMahDist_32f_D2	
ippsMahDist1_32f_D2L	ippsMahDist_32f_D2L	
ippsMahDist1_64f_D2	ippsMahDist_64f_D2	
ippsMahDist1_64f_D2L	ippsMahDist_64f_D2L	
ippsMahDist2_32f_D2	ippsMahDistMultiMix_32f_D2	
ippsMahDist2_32f_D2L	ippsMahDistMultiMix_32f_D2L	
ippsMahDist2_64f_D2	ippsMahDistMultiMix_64f_D2	
ippsMahDist2_64f_D2L	ippsMahDistMultiMix_64f_D2L	
ippsMeanCol_32f_D2	ippsMeanColumn_32f_D2	



Removed from 5.0	Substitution or Workaround	Comments
ippsMeanCol_32f_D2L	ippsMeanColumn_32f_D2L	
ippsMeanVarCol_32f_D2	ippsVarColumn_32f_D2	
ippsMeanVarCol_32f_D2L	ippsVarColumn_32f_D2L	
ippsMulCol_64f_D2L	ippsMulColumn_64f_D2L	
ippsNormalizeCol_32f_D2	ippsNormalizeColumn_32f_D2	
ippsQRTransCol_64f_D2L	ippsQRTransColumn_64f_D2L	
ippsRecSqrt_32f_Th	ippsRecSqrt_32f	
ippsStepSizeUpdateAECNLMS_32f		
ippsStepSizeUpdateAECNLMS_32s		
ippsSumAllRow_32f_D2	ippsSumRow_32f_D2	
ippsSumAllRow_32f_D2L	ippsSumRow_32f_D2L	
ippsSumCol_32f_D2	ippsSumColumn_32f_D2	
ippsSumCol_32f_D2L	ippsSumColumn_32f_D2L	
ippsSumCol_64f_D2	ippsSumColumn_64f_D2	
ippsSumCol_64f_D2L	ippsSumColumn_64f_D2L	
ippsSumColAbs_64f_D2L	ippsSumColumnAbs_64f_D2L	
ippsSumColSqr_64f_D2L	ippsSumColumnSqr_64f_D2L	
ippsVarCol_32f_D2	ippsVarColumn_32f_D2	
ippsVarCol_32f_D2L	ippsVarColumn_32f_D2L	

---

---

# Bibliography

---

This bibliography provides a list of reference books and other sources of additional information that might be useful to the application programmer. This list is neither complete nor exhaustive, but serves as a starting point. Of all the references listed, [Mit93] will be the most useful to those readers who already have a basic understanding of signal processing. This reference collects the work of 27 experts in the field and has both great breadth and depth.

The books [Opp75], [Opp89], [Jack89], and [Zie83] are undergraduate signal processing texts. [Opp89] is a much revised edition of the classic [Opp75]; [Jack89] is more concise than the others; and [Zie83] also covers continuous-time systems.

- [AMRWB+]            3GPP Technical Specification 26.290: *Extended Adaptive Multi-Rate - Wideband (AMR-WB+) codec; Transcoding functions*, v.6.3.0, rel.6, June-2005.
- [Arn97]            Z. Arnavut, S. Magliveras. *Block-Sorting and Compression*. IEEE Data Compression Conference, Snowbird, Utah, pp.181-190, March 1997.
- [Ash94]            M.R. Asharif and F. Amano. *Acoustic Echo Canceled Using the FBAF Algorithm*. IEEE Trans. Comm., Vol. 42, No. 12, pp. 3090-3094, Dec. 1994.
- [Berl68]            Elwin R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill Book Company, 1968
- [Bla84]            Richard E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley Publishing Company, 1984
- [Bris94]            C. Brislawn. *Classification of Nonexpansive Symmetric Extension Transforms for Multirate Filter Banks*. Los Alamos Report LA-UR-94-1747, 1994.
- [Cap78]            V. Cappellini, A. G. Constantinides, and P. Emilani. *Digital Filters and Their Applications*. Academic Press, London, 1978.
- [Cap94]            Cappé O. *Elimination of the musical noise phenomenon with the Ephraim and Malah noise suppressor*. IEEE Trans. Speech and Audio Processing, vol. 2(2), (1994).
- [Cast93]            G.Castagnoli, S.Brauer, M.Herrmann. *Optimization of cyclic redundancy-check codes with 24 and 32 parity bits*. IEEE Transactions on Communications, Vol.41, No.6, June, 1993, pp.883-892.
- [CCITT]            CCITT, Recommendation G.711. *Pulse Code Modulation of Frequencies*, 1984.
- [Coh02]            I. Cohen and B. Berdugo. *Noise Estimation by Minima Controlled Recursive Averaging for Robust Speech Enhancement*. IEEE Signal Proc. Letters, Vol. 9, No. 1, Jan. 2002, pp. 12-15.
- [Cro83]            R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1983.
- [Dau92]            I. Daubechies. *Ten Lectures on Wavelets*. Springer Verlag, Pennsylvania, 1992.

- [Eph84] Y. Ephraim and D. Malah. Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator. *IEEE Trans. ASSP*, Vol. 32, No. 6, Dec. 1984, pp. 1109-1121.
- [ETSI02] ETSI TS 102 114:2002 (DTS Coherent Acoustics – Core and Extensions)(2002).  
[http://www.etsi.org/services\\_products/freestandard/home.htm](http://www.etsi.org/services_products/freestandard/home.htm)
- [EC126] ETSI TS 126 404 V6.0.0. UMTS. *General audio codec audio processing functions; Enhanced aacPlus general audio codec; Encoder specification; SBR PART - 3GPP TS 26.404 version 6.0.0 Release 6* (09/2004).
- [ES201] ETSI ES 201 108 V1.1.2. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Front-end feature extraction algorithm; Compression algorithms.*
- [ES202] ETSI ES 202 050 V1.1.1. ETSI Standard. *Speech processing, Transmission and Quality aspects (STQ); Distributed speech recognition; Advanced front-end feature extraction algorithm; Compression algorithms.*
- [Feig92] E. Feig and S. Winograd. *Fast algorithms for DCT*. *IEEE Transactions on Signal Processing*, vol.40, No.9, 1992.
- [Gal75] R. Gallager, D.Van Voorhis. *Optimal source codes for geometrically distributed integer alphabets*. *IEEE Transactions on Information Theory*, vol. IT-21, No. 3, pp.228-230, 1975.
- [Griff87] G.Griffiths, G.C.Stones. *The tea-leaf reader algorithm: an efficient implementation of CRC-16 and CRC-32*. *Communications of the ACM* , vol.30, No.7, 1987, pp.617-620.
- [Ham83] R.W. Hamming. *Digital Filters*, Prentice-Hall, New Jersey 1983.
- [Har78] F. Harris. On the Use of Windows. *Proceedings of the IEEE*, vol. 66, No.1, IEEE, 1978.
- [Hay91] S. Haykin. *Adaptive Filter Theory*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [ISO11172] ISO/IEC 11172-3 - Information technology. *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s*. Part 3: Audio (1993)
- [ISO13818 ] ISO/IEC 13818-3 - Information technology. *Generic coding of moving pictures and associated audio information*. Part 3: Audio (1998).
- [ISO14496] ISO/IEC 14496-3 - Information technology. *Coding of audio-visual objects*. Part 3: Audio (2001).
- [ISO14496A] ISO/IEC 14496-3/Amd.1:2003 - *Bandwidth Extension* (2003).
- [ISO14496B] ISO/IEC 14496-3/Amd.2:2004 - *Parametric coding for high-quality audio* (2004).

- [ITU169] ITU-T Recommendation G.169. *Automatic Level Control Devices*, (06/99).
- [ITU224] ITU-T Recommendation X.224 Annex D. *Checksum Algorithms*, (11/93), pp144,145.
- [ITU722] ITU-T Recommendation G.722.1 (09/99), *Coding at 24 and 32 8 kbit/s for hand-free operation in systems with low frame loss*.
- [ITU723] ITU-T Recommendation G.723.1 . *Dual Rate speech coder for Multimedia Communications transmitting at 5.3 and 6.3 Kbit/s*, (03/96).
- [ITU723A] ITU-T Recommendation G.723.1 Annex A. *Silence compression scheme*, (11/96).
- [ITU728] ITU-T Recommendation G.728. *Coding of Speech at 16 kbits/s Using Low-Delay Code Excited Linear Prediction*, 1992.
- [ITU729] ITU-T Recommendation G.729. *Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)*, (03/96).
- [ITU729A] ITU-T Recommendation G.729 Annex A. *Reduced Complexity 8 kbit/s CS-ACELP speech codec*, (11/96).
- [ITU729B] ITU-T Recommendation G.729 Annex B. *A silence compression scheme for G.729 optimized for terminals conforming to Recommendation V.70*, (10/96).
- [ITU729B1] ITU-T Recommendation G.729 Annex B Corrigendum 1, (02/98).
- [ITU7291] ITU-T Recommendation G.729.1. *G729 Based Embedded Variable Bit-Rate Coder: a 8-32 kbits/s Scalable Wideband Coder Bitstream Interoperable with G.729*, (06/06).
- [ITUV34] ITU-T Recommendation V.34. *A modem operating at data signalling rates of up to 33 600 bit/s for use on the general switched telephone network and on leased point-to-point 2-wire telephone-type circuit*. (02/98).
- [Jack89] Leland B. Jackson. *Digital Filters and Signal Processing*. Kluwer Academic Publishers, second edition, 1989.
- [Kab86] P. Kabal and P.Ramachandran. *The Computation of line Spectral Frequencies Using Chebyshev Polynomials*. *IEEE transaction on acoustic, speech and signal processing*, vol. ASSP-34, No.6, 1986.
- [Lyn89] Paul A. Lynn. *Introductory Digital Signal Processing with Computer Applications*. John Wiley&Sons, Inc., New York, 1993.
- [Mar01] R. Martin. *Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics*. *IEEE Trans. Speech and Audio*, Vol. 9, No. 5, July 2001, pp. 504-512.

- [Med91] Y. Medan, E. Yair, D. Chazan. *Super Resolution Pitch Determination of Speech Signals*. IEEE Transactions on Signal Processing, vol 39, No.1, 1991.
- [Mit93] Sanjit K. Mitra and James F. Kaiser editors. *Handbook for Digital Signal Processing*. John Wiley & Sons, Inc., New York, 1993
- [Mit98] Sanjit K. Mitra. *Digital Signal Processing*. McGraw Hill, 1998.
- [Mor02] Robert H. Morelos-Zaragoza. *The Art of Error Correcting Coding*. Wiley & Sons, Ltd., 2002.
- [Nel92] M.R. Nelson. *File verification using CRC 32-bit cyclical redundancy check*. Dr. Dobb's Journal, vol. 17, Issue 5, 1992, p.64.
- [NIC91] Nam Ik Cho and Sang Uk Lee. *Fast algorithm and implementation of 2D DCT*. IEEE Transactions on Circuits and Systems, vol. 31, No.3, 1991.
- [Opp75] A.V. Oppenheim and R.W. Schaffer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
- [Opp89] A.V. Oppenheim and R.W. Schaffer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [Rab78] L.R. Rabiner and R.W. Schaffer. *Digital Processing of Speech Signals*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.
- [Rao90] K.R. Rao and P. Yip. *Discrete Cosine Transform. Algorithms, Advantages and Applications*. Academic Press, San Diego, 1990.
- [RFC1950] P. Deutsch, J.-L. Gailly. *ZLIB Compressed Data Format Specification* version 3.3, May, 1996. <http://www.ietf.org/rfc/rfc1950.txt>
- [RFC1951] P. Deutsch. *DEFLATE Compressed Data Format Specification* version 1.3, May, 1996. <http://www.ietf.org/rfc/rfc1951.txt>
- [RFC1952] P. Deutsch. *GZIP file format specification* version 4.3, May, 1996. <http://www.ietf.org/rfc/rfc1952.txt>
- [Seg78] R. Sedgewick. Implementing quicksort programs. Communications of the ACM, Vol. 21, No. 10, pp. 847-857, Oct. 1978.
- [Storer82] J. Storer, T. Szymanski. Data Compression via Textual Substitution. Journal of the Association for Computing Machinery (ACM), vol.19, No.4, pp.928-951, Oct.1982.
- [Strang96] G. Strang and T. Nguyen. *Wavelet and Filter Banks*. Wellesley-Cambridge Press, 1996, pp. 153-157.
- [Tuc92] R. Tucker. Voice Activity Detection Using a Periodicity Measure. IEEE Proceedings-I, vol. 139, No. 4, August 1992, pp. 377-380.
- [Vai93] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, Englewood Cliffs, New Jersey.

- [Wid85] B. Widrow and S.D. Stearns. Adaptive Signal Processing. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
- [Zie83] Rodger E. Ziemer, William H. Tranter and D. Ronald Fannin. Signals and Systems: Continuous and Discrete. Macmillan Publishing Co., New York, 1983.
- [Ziv77] J.Ziv and A.Lempel. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, vol.23, No.3, pp.337-343, May 1977.

---

---



# Glossary

---

adaptive filter	Colors specified by each pixel's coordinates in a color space. Intel Integrated Performance Primitives for image processing use images with absolute colors. An adaptive filter varies its filter coefficients (taps) over time. Typically, the filter's coefficients are varied to make its output match a prototype "desired" signal as closely as possible. Non-adaptive filters do not vary their filter coefficients over time.
BQ	One of the modes, which indicates that the IIR initialization function initializes a cascade of biquads.
CCS	See <i>complex conjugate-symmetric</i>
companding functions	The functions that perform an operation of data compression by using a logarithmic encoder-decoder. Companding allows you to maintain the percentage error constant by logarithmically spacing the quantization levels.
complex conjugate-symmetric	A kind of symmetry that arises in the Fourier transform of real signals. A complex conjugate-symmetric signal has the property that $x(-n) = x(n)^*$ , where "*" denotes conjugation.
conjugate	The conjugate of a complex number $a + bj$ is $a - bj$ .
conjugate-symmetric	See <i>complex conjugate-symmetric</i> .
DCT	Acronym for the discrete cosine transform.
decimation	Filtering a signal followed by down-sampling. Filtering prevents aliasing distortion in the subsequent down-sampling. See <i>down-sampling</i> .
down-sampling	Down-sampling conceptually decreases a signal's sampling rate by removing samples from between neighboring samples of a signal. See <i>decimation</i> .
element-wise	An element-wise operation performs the same operation on each element of a vector, or uses the elements of the same position in multiple vectors as inputs to the operation. For example, the element-wise addition of the vectors $\{x0, x1, x2\}$ and $\{y0, y1, y2\}$ is performed as follows: $\{x0, x1, x2\} + \{y0, y1, y2\} = \{x0 + y0, x1 + y1, x2 + y2\}$ .
FIR	Abbreviation for finite impulse response filter. Finite impulse response filters do not vary their filter coefficients (taps) over time.
FIR LMS	Abbreviation for least mean squares finite impulse response filter.
fixed-point data format	A format that assigns one bit for a sign and all other bits for fractional part. This format is used for optimized conversion operations with signed, purely fractional vectors. For example, S.31 format assumes a sign bit and 31 fractional bits; S15.16 assumes a sign bit, 15 integer bits, and 16 fractional bits.
IIR	Abbreviation for infinite impulse response filters.

in-place operation	A function that performs its operation in-place, takes its input from an array and returns its output to the same array. See <i>not-in-place operation</i> .
interpolation	Up-sampling a signal followed by filtering. The filtering gives the inserted samples a value close to the samples of their neighboring samples in the original signal. See <i>up-sampling</i> .
LMS	Abbreviation for <b>least mean square</b> , an algorithm frequently used as a measure of the difference between two signals. Also used as shorthand for an adaptive FIR filter employing the LMS algorithm for adaptation.
LTI	Abbreviation for linear time-invariant systems. In LTI systems, if an input consists of the sum of a number of signals, then the output is the sum of the system's responses to each signal considered separately.
MMX™ technology	An enhancement to Intel architecture aimed at better performance in multimedia and communications applications. The technology uses four additional data types, eight 64-bit MMX registers, and 57 additional instructions implementing the SIMD (single instruction, multiple data) technique.
MR	One of the modes, indicating the multi-rate variety of the function.
multi-rate	An operation or signal processing system involving signals with multiple sample rates. Decimation and interpolation are examples of multi-rate operations.
not-in-place operation	A function that performs its operation not-in-place takes its input from a source array and puts its output in a second, destination array.
polyphase	A computationally efficient method for multi-rate filtering. For example, interpolation or decimation.
CCS	A representation of a complex conjugate-symmetric sequence which is easier to use than the <code>Pack</code> or <code>Perm</code> formats.
Pack	A compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real FFT algorithms ("natural" in the sense that bit-reversed order is natural for radix-2 complex FFTs).
Perm	A format for storing the values for the FFT algorithm. The <code>Perm</code> format stores the values in the order in which the FFT algorithm uses them. That is, the real and imaginary parts of a given sample need not be adjacent.
saturation	Using saturation arithmetic, when a number exceeds the data-range limit for its data type, it saturates to the upper data-range limit. For example, a signed word greater than 7FFFh saturates to 7FFFh. When

	a number is less than the lower data-range limit, it saturates to the lower data-range. For example, a signed word less than 8000h saturates to 8000h.
sinusoid	See <i>tone</i>
Streaming SIMD Extensions	The major enhancement to Intel architecture instruction set. Incorporates a group of general-purpose floating-point instructions operating on packed data, additional packed integer instructions, together with cacheability control and state management instructions. These instructions significantly improve performance of applications using compute-intensive processing of floating-point and integer data.
tone	A sinusoid of a given frequency, phase, and magnitude. Tones are used as test signals and as building blocks for more complex signals.
up-sampling	Up-sampling conceptually increases the signal sampling rate by inserting zero-valued samples between neighboring samples of a signal.
window	A mathematical function by which a signal is multiplied to improve the characteristics of some subsequent analysis. Windows are commonly used in FFT-based spectral analysis.

---

---

# *Index*

10Log10 216

## **A**

### AAC

- ADIF header structure 1335
- ADTS frame header structure 1335
- channel information structure 1335
- channel pair element structure 1335
- individual channel side information structure 1335
- LTP structure 1335
- scalable extension element structure 1335
- scalable main element structure 1335
- TNS structure for one layer 1335

about this manual 53

about this software 51

Abs 205, 1509

AccCovarianceMatrix 694

accuracy, of arithmetic functions 1493

ACELPFixedCodebookSearch\_G723 961

Acos 1561

Acosh 1578

adaptive codebook 905

adaptive filters 1242

AdaptiveCodebookContribution\_G729 904

AdaptiveCodebookDecode\_AMRWB 1054

AdaptiveCodebookDecode\_AMRWBE 1078

AdaptiveCodebookDecode\_GSMAMR 1005

AdaptiveCodebookDecodeGetSize\_AMRWB 1052

AdaptiveCodebookDecodeInit\_AMRWB 1052

AdaptiveCodebookDecodeUpdate\_AMRWB 1053

AdaptiveCodebookGain\_G729 905

AdaptiveCodebookGain\_GSMAMR 1007

AdaptiveCodebookGainCoeff\_AMRWB 1049

AdaptiveCodebookSearch\_AMRWB 1050

AdaptiveCodebookSearch\_AMRWBE 1077

AdaptiveCodebookSearch\_G723 963

AdaptiveCodebookSearch\_G729 891

AdaptiveCodebookSearch\_GSMAMR 1004

AdaptiveCodebookSearch\_RTA 1200

Add 179, 1496

AddAllRowSum 627

AddC 177

AddMulColumn 780

AddMulRow 781

AddNRows 712

AddProduct 183

AddProductC 182

Adler32 1694

ALawToLin 1188

ALawToMuLaw 1191

ALC\_G169 1177

ALCGetStateSize\_G169 1175

ALCInit\_G169 1175

ALCSetGain\_G169 1177

ALCSetLevel\_G169 1176

AlgebraicCodebookDecode\_AMRWB 1048

AlgebraicCodebookSearch\_AMRWB 1043

AlgebraicCodebookSearch\_GSMAMR 1009

AlgebraicCodebookSearchL1\_G7291 937

AlgebraicCodebookSearchL2\_G7291 937

AltInitMCRA 839

amplitude ratio 969

AMR WB speech codec

Deemphasize\_AMRWB 1038

AdaptiveCodebookDecode\_AMRWB 1054

AdaptiveCodebookDecodeGetSize\_AMRWB 1052

AdaptiveCodebookDecodeInit\_AMRWB 1052

AMR WB speech codec (*continued*)

AdaptiveCodebookDecodeUpdate\_AMRWB 1053  
 AdaptiveCodebookGainCoeff\_AMRWB 1049  
 AdaptiveCodebookSearch\_AMRWB 1050  
 AlgebraicCodebookDecode\_AMRWB 1048  
 AlgebraicCodebookSearch\_AMRWB 1043  
 DecDTXBuffer\_AMRWB 1063  
 DecodeGain\_AMRWB 1061  
 EncDTXBuffer\_AMRWB 1062  
 GainQuant\_AMRWB 1059  
 HighPassFilter\_AMRWB 1034  
 HighPassFilterGetDlyLine\_AMRWB 1035  
 HighPassFilterGetSize\_AMRWB 1032  
 HighPassFilterInit\_AMRWB 1033  
 HighPassFilterSetDlyLine\_AMRWB 1036  
 ISFQuant\_AMRWB 1055  
 ISFQuantDecode\_AMRWB 1057  
 ISFQuantDecodeDTX\_AMRWB 1059  
 ISFQuantDTX\_AMRWB 1058  
 ISFToISP\_AMRWB 1030  
 ISPToISF\_Norm\_AMRWB 1029  
 ISPToLPC\_AMRWB 1028  
 LPCToISP\_AMRWB 1026  
 OpenLoopPitchSearch\_AMRWB 1031  
 Preemphasize\_AMRWB 1037  
 ResidualFilter\_AMRWB 1025  
 SynthesisFilter\_AMRWB 1039  
 VAD\_AMRWB 1041  
 VADGetEnergyLevel\_AMRWB 1042  
 VADGetSize\_AMRWB 1040  
 VADInit\_AMRWB 1040

AMR WB+ speech codec

AdaptiveCodebookSearch\_AMRWBE 1077  
 BandJoin\_AMRWBE 1082  
 BandJoinUpsample\_AMRWBE 1084  
 BandSplit\_AMRWBE 1081  
 BandSplitDownsample\_AMRWBE 1083  
 DecodeDemux\_AMRWBE 1096  
 Deemphasize\_AMRWBE 1072  
 Downsample\_AMRWBE 1079  
 EncodeMux\_AMRWBE 1095  
 FFTInv\_PermToR\_AMRWBE 1076, 1078  
 FFTFwd\_RToPerm\_AMRWBE 1075  
 FIRGenMidBand\_AMRWBE 1072  
 GainDecodeTCX\_AMRWBE 1094  
 GainQuant\_AMRWBE 1090  
 GainQuantTCX\_AMRWBE 1092  
 ISFQuantDecode\_AMRWBE 1087  
 ISFQuantDecodeHighBand\_AMRWBE 1088

AMR WB+ speech codec (*continued*)

ISFQuantHighBand\_AMRWBE 1089  
 LPCToISP\_AMRWBE 1070  
 OpenLoopPitchSearch\_AMRWBE 1069  
 PostFilterLowBand\_AMRWBE 1073  
 QuantTCX\_AMRWBE 1091  
 ResamplePolyphase\_AMRWBE 1085  
 SNR\_AMRWBE 1067  
 SynthesisFilter\_AMRWBE 1071  
 Upsample\_AMRWBE 1080  
 AnalysisFilter\_PQMF\_MP3 1292  
 AnalysisFilter\_PS 1434  
 AnalysisFilter\_SBR 1415  
 AnalysisFilterEnc\_SBR 1395  
 AnalysisFilterEncFree\_SBR 1397  
 AnalysisFilterEncGetSize\_SBR 1394  
 AnalysisFilterEncInit\_SBR 1394  
 AnalysisFilterEncInitAlloc\_SBR 1396  
 AnalysisFilterFree\_PQMF\_MP3 1291  
 AnalysisFilterFree\_SBR 1403  
 AnalysisFilterGetSize\_PQMF\_MP3 1291  
 AnalysisFilterGetSize\_SBR 1406  
 AnalysisFilterInit\_PQMF\_MP3 1290  
 AnalysisFilterInit\_SBR 1410  
 AnalysisFilterInitAlloc\_PQMF\_MP3 1290  
 AnalysisFilterInitAlloc\_SBR 1400  
 AnalysisPQMF\_MP3 1288  
 And 167  
 AndC 166  
 Arctan 218  
 Arg 1511  
 Arithmetic functions  
     Add 179  
     AddC 177  
     10Log10 216  
     Abs 205  
     AddProduct 183  
     AddProductC 182  
     Arctan 218  
     Cauchy 221  
     Cubrt 211  
     Div 199  
     Div\_Round 203  
     DivC 196  
     DivCRev 198  
     Exp 212  
     Ln 214  
     Mul 186  
     MulC 184

Arithmetic functions (*continued*)

- Normalize 219
- Sqr 206
- Sqrt 208
- Sub 194
- SubC 189
- SubCRev 192
- SumLn 217

Asin 1564

Asinh 1581

Atan 1566

Atan2 1569

Atanh 1584

audience for this manual 60

audio coding 1207

Audio coding functions

- CalcSF 1226
- Deinterleave 1215
- FDPFree 1249
- FDPFwd 1253
- FDPGetSize 1249
- FDPInit 1248
- FDPInitAlloc 1247
- FDPInv 1253
- FDPReset 1250
- FDPResetGroup 1251
- FDPResetSfb 1250
- FIRBlockFree 1243
- FIRBlockInitAlloc 1242
- FIRBlockOne 1244
- Interleave 1213
- MakeFloat 1230
- MDCTFwd, MDCTInv 1238
- MDCTFwdFree, MDCTInvFree 1235
- MDCTFwdGetBufSize, MDCTInvGetBufSize 1237
- MDCTFwdGetSize, MDCTInvGetSize 1236
- MDCTFwdInit, MDCTInvInit 1234
- MDCTFwdInitAlloc, MDCTInvInitAlloc 1232
- Pow34 1217
- Pow43 1221
- Pow43Scale 1222
- PreSelect\_VQ 1272
- Scale 1228
- VLCCountEscBits\_AAC 1263
- VLCCountEscBits\_MP3 1262
- VLCDecodeEscBlock\_AAC 1258
- VLCDecodeEscBlock\_MP3 1255
- VLCDecodeUTupleEscBlock\_AAC 1261
- VLCDecodeUTupleEscBlock\_MP3 1260

Audio coding functions (*continued*)

- VLCDecodeEscBlock\_AAC 1266
- VLCDecodeEscBlock\_MP3 1264
- VQCodeBookFree 1271
- VQCodeBookGetSize 1271
- VQCodeBookInit 1270
- VQCodeBookInitAlloc 1269
- VQIndexSelect 1276
- VQMainSelect 1274
- VQReconstruction 1279

Aurora functions

- BlindEqualization\_Aurora 827
- EvalDelta\_Aurora 828
- HighBandCoding\_Aurora 826
- LowHighFilter\_Aurora 825
- MelFBankInitAlloc\_Aurora 822
- NoiseSpectrumUpdate\_Aurora 820
- ResidualFilter\_Aurora 824
- SmoothedPowerSpectrumAurora 819
- TabCalculation\_Aurora 823
- VADDecision\_Aurora 832
- VADFlush\_Aurora 833
- VADGetBufSize\_Aurora 831
- VADInit\_Aurora 831
- WaveProcessing\_Aurora 824
- WienerFilterDesign\_Aurora 821

AutoCorr 339, 861

AutoCorr\_G723 952

AutoCorr\_G729 873

AutoCorr\_GSMAMR 982

AutoCorr\_NormE 863

AutoCorr\_NormE\_G723 953

autocorrelation, in speech codecs 873

AutoCorrLagMax 862

automatic audio level control

- ALC\_G169 1177
- ALCGetStateSize\_G169 1175
- ALCInit\_G169 1175
- ALCSetGain\_G169 1177
- ALCSetLevel\_G169 1176

AutoScale 858

**B**

BandJoin\_AMRWBE 1082

BandJoinUpsample\_AMRWBE 1084

BandPassFilter\_RTA 1205

BandSplit\_AMRWBE 1081

BandSplitDownsample\_AMRWBE 1083  
 bandwidth expansion 956  
 Basic arithmetic functions for speech recognition  
     AddAllRowSum 627  
     BlockDMatrixFree 637  
     BlockDMatrixInitAlloc 636  
     CopyColumn\_Indirect 634  
     MatVecMul 640  
     NthMaxElement 637  
     SubRow 633  
     SumColumn 629  
     SumRow 631  
     VecMatMul 638  
 BhatDist 771  
 binomial window 952  
 bit stream 1226  
 BitReservoirInit\_MP3 1316  
 BlindEqualization\_Aurora 827  
 block filtering 1242  
 BlockDMatrixFree 637  
 BlockDMatrixInitAlloc 636  
 BuildSignTable 744  
 BuildSymbITableDV4D 286  
 Burrows-Wheeler transform  
     BUTFwd 1718  
     BUTFwd\_SmallBlock 1722  
     BUTFwdGetSize 1717  
     BWTGetSize\_SmallBlock 1721  
     BWTInv 1719  
     BWTInv\_SmallBlock 1723  
     BWTInvGetSize 1719  
 BUTFwd 1718  
 BUTFwd\_SmallBlock 1722  
 BUTFwdGetSize 1717  
 BWTGetSize\_SmallBlock 1721  
 BWTInv 1719  
 BWTInv\_SmallBlock 1723  
 BWTInvGetSize 1719  
 bzip2 Coding Functions 1746  
 bzip2 compatibility functions  
     CRC32\_BZ2 1754  
 bzip2 compatible functions  
     DecodeHuff\_BZ2 1764  
     DecodeHuffFree\_BZ2 1762  
     DecodeHuffGetSize\_BZ2 1760  
     DecodeHuffInit\_BZ2 1761  
     DecodeHuffInitAlloc\_BZ2 1762  
     DecodeRLE\_BZ2 1750  
     DecodeZ1Z2\_BZ2 1752

bzip2 compatible functions (*continued*)  
     EncodeHuff\_BZ2 1759  
     EncodeHuffFree\_BZ2 1758  
     EncodeHuffGetSize\_BZ2 1755  
     EncodeHuffInit\_BZ2 1756  
     EncodeHuffInitAlloc\_BZ2 1757  
     EncodeRLE\_BZ2 1748  
     EncodeRLEFlush\_BZ2 1749  
     EncodeRLEInit\_BZ2 1747  
     EncodeZ1Z2\_BZ2 1751  
     ExpandDictionary 1754  
     PackHuffContext\_BZ2 1758  
     ReduceDictionary 1753  
     RLEFree\_BZ2 1746, 1747  
     RLEGetInUseTable 1750  
     RLEGetSize\_BZ2 1748  
     UnpackHuffContext\_BZ2 1763

## C

CalcSF 1226  
 CalcStatesDV 285  
 CartToPolar, complex 276  
 Cauchy 221  
 CauchyD 221  
 CauchyDD2 221  
 Cbrt 1523  
 CdbkFree 798  
 CdbkGetSize 794  
 CdbkInit 795  
 CdbkInitAlloc 796  
 CdfNorm 1591  
 CdfNormInv 1599  
 Ceil 1604  
 CepstrumToLP 654  
 Chebyshev polynomials 877  
 CIS 1557  
 codebook 880, 884, 900, 904  
     fixed 900  
     indices 884  
     search 904  
 CodebookSearch\_G728 1133  
 CodebookSearchTCQ\_G728 1134  
 CombinedFilter\_G728 1122  
 CombinedFilterGetStateSize\_G728 1121  
 CombinedFilterInit\_G728 1121  
 common functions  
     ippAlignPtr 100



common functions (*continued*)

- ippFree 103
- ippGetCpuClocks 94
- ippGetCpuFeatures 95
- ippGetCpuFreqMhz 95
- ippGetCpuType 92
- ippGetMaxCacheSizeB 98
- ippGetNumCoresOnDie 98
- ippGetNumThreads 102
- ippGetStatusString 91
- ippMalloc 102
- ippSetDenormAreZeros 100
- ippSetFlushToZero 99
- ippSetNumThreads 101
- ippsGetLibVersion 87
- ippStaticInit 103

Companding functions

- ALawToLin 1188
- bALawToMuLaw 1191
- LinToALaw 1189
- LinToMuLaw 1187
- MuLawToALaw 1190
- MuLawToLin 1186

Compare 1451

CompareIgnoreCase 1452

CompareIgnoreCaseLatin 1452

CompensateOffset 643, 696

CompressEnvelopTime\_G7291 944

Concat 1463

ConcatC 1465

Conj 244, 1507

ConjCcs 500

ConjFlip 245

ConjPack 496

ConjPerm 498

Conv 336

ConvBiased 337

ConvCyclic 339

Conversion functions

- Imag 253
- CartToPolar, complex 276
- Conj 244
- ConjFlip 245
- Convert 233
- CplxToReal 256
- DemodulateFM 257
- FindNearest 283
- FindNearestOne 282
- Flip 281

Conversion functions (*continued*)

- Join 239
- JoinScaled 241
- Magnitude 246
- MagSquared 248
- MaxOrder 279
- Phase, complex 249
- PolarToCart, complex 278
- Preemphasize 280
- Real 252
- RealToCplx 254
- SortAscend 223
- SortDescend 223
- SortIndexAscend 224
- SortIndexDescend 224
- SortRadixAscend 226
- SortRadixDescend 226
- SortRadixIndexAscend 228
- SortRadixIndexDescend 228
- SplitScaled 243
- SwapBytes 231
- Threshold 258
- Threshold\_GT 262
- Threshold\_GTAbs 266
- Threshold\_GTVal 268
- Threshold\_LT 262
- Threshold\_LTAbs 266
- Threshold\_LTVal 268
- Threshold\_LTValGTVal 268

Convert 233

Convolution and correlation functions

- Conv 336
- ConvCyclic 339
- CrossCorr 342
- UpdateLinear 345
- UpdatePower 347

ConvPartial 852

Copy 113

CopyColumn 699

CopyColumn\_Indirect 634

CopyWithPadding 667

Cos 1550

Cosh 1571

CountInRange 329

CplxToReal 256

CRC32 1695

CRC32\_BZ2 1754

CRC32C 1695

Cross-Architecture Alignment 52

CrossCorr 342, 865  
 CrossCorrCoeff 709  
 CrossCorrCoeffDecim 707  
 CrossCorrCoeffInterpolation 710  
 CrossCorrLagMax 866  
 Cubrt 211

## D

### Data compression functions

Adler32 1694  
 BWTfwd 1718  
 BWTfwd\_SmallBlock 1722  
 BWTfwdGetSize 1717  
 BWTGetSize\_SmallBlock 1721  
 BWTInv 1719  
 BWTInv\_SmallBlock 1723  
 BWTInvGetSize 1719  
 CRC32 1695  
 CRC32\_BZ2 1754  
 CRC32C 1695  
 DecodeGIT 1734  
 DecodeGITGetSize 1733  
 DecodeGITInit 1732  
 DecodeGITInitAlloc 1732  
 DecodeHuff 1649  
 DecodeHuffFree\_BZ2 1762  
 DecodeHuffInit 1648  
 DecodeHuffInitAlloc 1647  
 DecodeHuffOne 1648  
 DecodeLZ77 1685  
 DecodeLZ77DynamicHuff 1689  
 DecodeLZ77FixedHuff 1688  
 DecodeLZ77GetPairs 1691  
 DecodeLZ77GetSize 1684  
 DecodeLZ77GetStatus 1692  
 DecodeLZ77Init 1683  
 DecodeLZ77InitAlloc 1685  
 DecodeLZ77InitGetBlockType 1686  
 DecodeLZ77Reset 1694  
 DecodeLZ77SelectHuffMode 1677  
 DecodeLZ77SetPairs 1692  
 DecodeLZ77SetStatus 1693  
 DecodeLZ77StoredBlock 1690  
 DecodeLZSS 1662  
 DecodeLZSSInit 1661  
 DecodeLZSSInitAlloc 1660  
 DecodeRLE 1744

### Data compression functions (*continued*)

DeflateDictionarySet 1700  
 DeflateHuff 1701  
 DeflateLZ77 1698  
 EncodeGIT 1728  
 EncodeGITGetSize 1727  
 EncodeGITInit 1726  
 EncodeGITInitAlloc 1724  
 EncodeHuff 1645  
 EncodeHuffFinal 1646  
 EncodeHuffFree\_BZ2 1758  
 EncodeHuffInit 1643  
 EncodeHuffInitAlloc 1641  
 EncodeHuffOne 1644  
 EncodeLZ77 1674  
 EncodeLZ77FixedHuff 1676  
 EncodeLZ77Flush 1679  
 EncodeLZ77GetPairs 1680  
 EncodeLZ77GetSize 1672  
 EncodeLZ77GetStatus 1681  
 EncodeLZ77Init 1671  
 EncodeLZ77InitAlloc 1672  
 EncodeLZ77Reset 1683  
 EncodeLZ77SelectHuffMode 1675  
 EncodeLZ77SetPairs 1681  
 EncodeLZ77SetStatus 1682  
 EncodeLZ77StoredBlock 1678  
 EncodeLZSS 1655  
 EncodeLZSSFlush 1656  
 EncodeLZSSInit 1654  
 EncodeLZSSInitAlloc 1653  
 EncodeRLE 1743  
 EncodeRLEInitAlloc 1746  
 GITFree 1725  
 HuffFree 1642  
 HuffGetDstBuffSize 1650  
 HuffGetLenCodeTable 1646  
 HuffGetSize 1643  
 HuffLenCodeTablePack 1651  
 HuffLenCodeTableUnpack 1652  
 Inflate 1703  
 InflateBuildHuffTable 1702  
 LZ77Free 1673  
 LZSSFree 1654  
 LZSSGetSize 1655  
 MTFFree 1739  
 MTFFwd 1741  
 MTFGetSize 1740  
 MTFInit 1740

Data compression functions (*continued*)

MTFInitAlloc 1739  
MTFInv 1742  
RLEFree\_BZ2 1747  
VLCCountBits 1627  
VLCDecodeBlock 1631  
VLCDecodeFree 1628  
VLCDecodeGetSize 1630  
VLCDecodeInit 1629  
VLCDecodeInitAlloc 1627  
VLCDecodeOne 1632  
VLCDecodeUTupleBlock 1638  
VLCDecodeUTupleFree 1635  
VLCDecodeUTupleGetSize 1637  
VLCDecodeUTupleInit 1636  
VLCDecodeUTupleInitAlloc 1634  
VLCDecodeUTupleOne 1639  
VLCEncodeBlock 1625  
VLCEncodeFree 1622  
VLCEncodeGetSize 1624  
VLCEncodeInit 1623  
VLCEncodeInitAlloc 1622  
VLCEncodeOne 1626

Data integrity functions 1767, 1768, 1769, 1770,  
1771, 1772, 1773, 1774, 1775, 1776, 1777,  
1778, 1779, 1780, 1781, 1782, 1783, 1784,  
1785, 1786, 1787, 1788, 1789, 1790, 1791,  
1792, 1793, 1794, 1795, 1796, 1797, 1798,  
1799, 1800, 1801, 1802

GFAAdd 1770  
GFDiv 1773  
GFExpAlpha 1778  
GFGetSize 1768  
GFInit 1769  
GFInv 1775  
GFLogAlpha 1777  
GFMul 1772  
GFNeg 1776  
GFPow 1774  
GFSub 1771  
PolyGFAdd 1785  
PolyGFCopy 1783  
PolyGFDiv 1788  
PolyGFGCD 1794  
PolyGFGetRef 1784  
PolyGFGetSize 1779  
PolyGFInit 1780  
PolyGFIrreducible 1790  
PolyGFMod 1786

Data integrity functions (*continued*)

PolyGFMul 1787  
PolyGFPrimitive 1791  
PolyGFRoots 1793  
PolyGFSetCoeffs 1781  
PolyGFSetDegree 1782  
PolyGFSetDerive 1792  
PolyGFShIC 1789  
PolyGFShrC 1789  
PolyGFSub\_8u 1785  
PolyGFValue 1792  
RSDecodeBM 1802  
RSDecodeBMGetBufferSize 1801  
RSDecodeEE 1802  
RSDecodeEEGetBufferSize 1801  
RSDecodeGetSize 1799  
RSDecodeInit 1800  
RSEncode 1798  
RSEncodeGetBufferSize 1797  
RSEncodeGetSize 1795  
RSEncodeInit 1796

DcsClustLAccumulate 778

DcsClustLCompute 779

## DCT functions

DCT4GetSize 584  
DCTFwd 576  
DCTFwdFree 569  
DCTFwdGetBufSize 570  
DCTFwdGetSize 574  
DCTFwdInitAlloc 567  
DCTIFwdInit 572  
DCTInv 578  
DCTInvFree 569  
DCTInvGetBufSize 571  
DCTInvGetSize 575  
DCTInvInit 573  
DCTInvInitAlloc 568

DCT4 585

## DCT4 functions

DCT4 585  
DCT4Free 583  
DCT4Init 583  
DCT4InitAlloc 582

DCT4Free 583

DCT4GetSize 584

DCT4Init 583

DCT4InitAlloc 582

DCTFwd 576

DCTFwd\_G722 1105

DCTFwd_G7221	1105	DecodeLZ77FixedHuff	1688
DCTFwdFree	569	DecodeLZ77GetBlockType	1686
DCTFwdGetBufSize	570	DecodeLZ77GetPairs	1691
DCTFwdGetSize	574	DecodeLZ77GetSize	1684
DCTFwdInit	572	DecodeLZ77GetStatus	1692
DCTFwdInitAlloc	567	DecodeLZ77Init	1683
DCTInv	578	DecodeLZ77InitAlloc	1685
DCTInv_G722	1106	DecodeLZ77Reset	1694
DCTInv_G7221	1106	DecodeLZ77SetStatus	1693
DCTInvFree	569	DecodeLZ77StoredBlock	1690
DCTInvGetBufSize	571	DecodeLZO	1710
DCTInvGetSize	575	DecodeLZOSafe	1711
DCTInvInit	573	DecodeLZSS	1662
DCTInvInitAlloc	568	DecodeLZSSInit	1661
DCTLifter	684	DecodeLZSSInitAlloc	1660
DCTLifterFree	684	DecodeMainHeader_AAC	1372
DCTLifterGetSize_MulC0	680	DecodeMsPNS_AAC	1375
DCTLifterInit_MulC0	681	DecodeMsStereo_AAC	1359
DCTLifterInitAlloc	682	DecodePNS_AAC	1374
DecDTXBuffer_AMRWB	1063	DecodePrgCfgElt_AAC	1350
DecDTXBuffer_GSMAMR	1021	DecodeRLE	1744
Decode_G726	1115	DecodeRLE_BZ2	1750
DecodeAdaptiveVector_G723	976	DecodeTNS_AAC	1364
DecodeAdaptiveVector_G729	892	DecomposeDCTToMLT_G722	1108
DecodeChanPairElt_AAC	1350	DecomposeDCTToMLT_G7221	1108
DecodeChanPairElt_MP4_AAC	1376	DecomposeMLTToDCT_G722	1107
DecodeDatStrElt_AAC	1355	DecomposeMLTToDCT_G7221	1107
DecodeDemux_AMRWBE	1096	Deemphasize_AMRWB	1038
DecodeExtensionHeader_AAC	1373	Deemphasize_AMRWBE	1072
DecodeFillElt_AAC	1356	Deemphasize_GSMFR	1098
DecodeGain_AMRWB	1061	DEFLATE	1666
DecodeGain_G729	900	DeflateDictionarySet	1700
DecodeGetStateSize_G726	1113	DeflateHuff	1701
DecodeGIT	1734	DeflateLZ77	1698
DecodeGITGetSize	1733	Deinterleave	1215
DecodeGITInit	1732	DeinterleaveSpectrum_AAC	1363
DecodeGITInitAlloc	1732	Delta	701
DecodeHuff_BZ2	1764	DeltaDelta	704
DecodeHuffFree_BZ2	1762	DemodulateFM	257
DecodeHuffGetSize_BZ2	1760	Derivative functions	
DecodeHuffInit	1648	CopyColumn	699
DecodeHuffInit_BZ2	1761	Delta	701
DecodeHuffInitAlloc	1647	DeltaDelta	704
DecodeHuffInitAlloc_BZ2	1762	EvalDelta	700
DecodeHuffOne	1648	DetectTransient_SBR	1390
DecodeInit_G726	1114	DFT for given frequency (Goertzel) functions	
DecodeIsStereo_AAC	1361	Goertz	557
DecodeLZ77	1685	GoertzTwo	560
DecodeLZ77DynamicHuff	1689		

## DFT functions

- DFTFree\_C 534
- DFTFree\_R 534
- DFTFwd\_CToC 536
- DFTFwd\_RToCCS 543
- DFTFwd\_RToPack 543
- DFTFwd\_RToPerm 543
- DFTGetBufSize\_C 535
- DFTGetBufSize\_R 535
- DFTInitAlloc\_C 532
- DFTInitAlloc\_R 532
- DFTInv\_CCSToR 548
- DFTInv\_CToC 540
- DFTInv\_PackToR 548
- DFTInv\_PermToR 548
- DFTOutOrdFree\_C 553
- DFTOutOrdFwd\_CToC 555
- DFTOutOrdGetBufSize\_C 554
- DFTOutOrdInitAlloc\_C 552
- DFTOutOrdInv\_CToC 556
- DFTFwd\_RToCCS 543
- DFTFwd\_RToPack 543
- DFTFwd\_RToPerm 543
- DFTInv\_CToC 540
- DFTOutOrdFree\_C 553
- DFTOutOrdFwd\_CToC 555
- DFTOutOrdGetBufSize\_C 554
- DFTOutOrdInitAlloc\_C 552
- DFTOutOrdInv\_CToC 556
- discrete Hartley transform
  - Hartley 562
- dispatcher functions
  - ippEnableCpu 106
  - ippInit 104
  - ippInitCpu 106
  - ippStaticInitCpu 105
- Distance coding 1723
- Div 196, 199, 1515
- Div\_Round 203
- DivCRev 198
- DotProd 323
- DotProd\_G729 871
- DotProdAutoScale 859
- DotProdColumn 783
- Downsample\_AMRWBE 1079
- DTW 750
- Durbin 649

**E**

## Echo Cancellation

- FIR\_EC 1161
- FIRSubband\_EC 1162
- FIRSubbandCoeffUpdate\_EC 1163
- FIRSubbandLow\_EC 1162
- FIRSubbandLowCoeffUpdate\_EC 1163
- FullbandController\_EC 1159
- FullbandControllerGetSize\_EC 1156
- FullbandControllerInit\_EC 1157
- FullbandControllerReset\_EC 1160
- FullbandControllerUpdate\_EC 1158
- NLMS\_EC 1165
- SubbandController\_EC 1146
- SubbandControllerDT\_EC 1150
- SubbandControllerDTGetSize\_EC 1148
- SubbandControllerDTReset\_EC 1150
- SubbandControllerDTUpdate\_EC 1152
- SubbandControllerGetSize\_EC 1143
- SubbandControllerInit\_EC 1144
- SubbandControllerReset\_EC 1147
- SubbandControllerUpdate\_EC 1145
- SubbandProcessGetSize 1138
- SubbandProcessInit 1139, 1140, 1149
- SubbandSynthesis 1141
- ToneDetect\_EC 1155
- ToneDetectGetStateSize\_EC 1154
- ToneDetectInit\_EC 1154

## EMNS functions

- AltInitMCRA 839
- FilterUpdateEMNS 835
- FilterUpdateWiener 836
- GetSizeMCRA 838
- InitMCRA 838
- UpdateNoisePSDMCRA 840
- EmptyFBankInitAlloc 674
- EncDTXBuffer\_AMRWB 1062
- EncDTXBuffer\_GSMAMR 1021
- EncDTXHandler\_GSMAMR 1019
- EncDTXSID\_GSMAMR 1018
- Encode\_G726 1112
- EncodeGetStateSize\_G726 1111
- EncodeGIT 1728
- EncodeGITGetSize 1727
- EncodeGITInit 1726
- EncodeGITInitAlloc 1724
- EncodeHuff 1645
- EncodeHuff\_BZ2 1759

EncodeHuffFinal 1646  
EncodeHuffFree\_BZ2 1758  
EncodeHuffGetSize\_BZ2 1755  
EncodeHuffInit 1643  
EncodeHuffInit\_BZ2 1756  
EncodeHuffInitAlloc 1641  
EncodeHuffInitAlloc\_BZ2 1757  
EncodeHuffOne 1644  
EncodeInit\_G726 1112  
EncodeLZ77 1674  
EncodeLZ77DynamicHuff 1677  
EncodeLZ77FixedHuff 1676  
EncodeLZ77Flush 1679  
EncodeLZ77GetPairs 1680, 1692  
EncodeLZ77GetSize 1672  
EncodeLZ77GetStatus 1681  
EncodeLZ77Init 1671  
EncodeLZ77InitAlloc 1672  
EncodeLZ77Reset 1683  
EncodeLZ77SelectHuffMode 1675  
EncodeLZ77SetPairs 1681  
EncodeLZ77SetStatus 1682  
EncodeLZ77StoredBlock 1678  
EncodeLZO 1709  
EncodeLZOGetSize 1707  
EncodeLZOInit 1708  
EncodeLZSS 1655  
EncodeLZSSFlush 1656  
EncodeLZSSInit 1654  
EncodeLZSSInitAlloc 1653  
EncodeMux\_AMRWB 1095  
EncodeRLE 1743  
EncodeRLE\_BZ2 1748  
EncodeRLEFlush\_BZ2 1749  
EncodeRLEInit\_BZ2 1747  
EncodeRLEInitAlloc 1746  
EncodeTNS\_AAC 1383  
EncodeZ1Z2\_BZ2 1751  
energy, average 903  
Entropy 768  
EnvelopTime\_G7291 939, 940, 942, 943  
Equal 1454  
Erf 1586  
Erfc 1588  
ErfcInv 1596  
ErfInv 1593  
EstimateTNR\_SBR 1392  
EvalDelta 700  
EvalDelta\_Aurora 828

EvalFBank 679  
excitation 891, 925  
    random 925  
Exp 212, 1539  
ExpandDictionary 1754  
Expm1 1542  
ExpNegSqr 770

## F

FBankFree 675  
FBankGetCenters 676  
FBankGetCoeffs 677  
FBankSetCenters 676  
FBankSetCoeffs 678  
FDPFree 1249  
FDPFwd 1253  
FDPGetSize 1249  
FDPInit 1248  
FDPInitAlloc 1247  
FDPInv 1253  
FDPReset 1250  
FDPResetGroup 1251  
FDPResetSfb 1250  
Feature processing functions  
    CepstrumToLP 654  
    CompensateOffset 643, 696  
    CopyWithPadding 667  
    DCTLifter 684  
    DCTLifterFree 684  
    DCTLifterGetSize\_MulC0 680  
    DCTLifterInit\_MulC0 681  
    DCTLifterInitAlloc 682  
    Durbin 649  
    EmptyFBankInitAlloc 674  
EvalFBank 679  
FBankFree 675  
FBankGetCenters 676  
FBankGetCoeffs 677  
FBankSetCenters 676  
FBankSetCoeffs 678  
LinearPrediction 646  
LinearToMel 666  
LPToCepstrum 653  
LPToLSP 663  
LPToReflection 655  
LPToSpectrum 652  
LSPToLP 664

Feature processing functions (*continued*)

MelfBankGetSize 668  
 MelfBankInit 669  
 MelfBankInitAlloc 670  
 MelLinFBankInitAlloc 672  
 MelToLinear 665  
 PitchmarkToF0Cand 660  
 ReflectionToAR 657  
 ReflectionToLP 656  
 ReflectionToTilt 659  
 Schur 650  
 SignChangeRate 645  
 UnitCurve 661  
 ZeroMean 642  
 FFT functions  
   FFTFree\_C 510  
   FFTFree\_R 510  
   FFTFwd\_CToC 517  
   FFTFwd\_RToCCS 524  
   FFTFwd\_RToPack 524  
   FFTFwd\_RToPerm 524  
   FFTGetBufSize\_C 516  
   FFTGetBufSize\_R 516  
   FFTGetSize\_C 514  
   FFTGetSize\_R 514  
   FFTInit\_C 512  
   FFTInit\_R 512  
   FFTInitAlloc\_C 508  
   FFTInitAlloc\_R 508  
   FFTInv\_CCSToR 528  
   FFTInv\_CToC 521  
   FFTInv\_PackToR 528  
   FFTInv\_PermToR 528  
 FFTFwd\_CToC 517  
 FFTFwd\_RToCCS 524  
 FFTFwd\_RToPack 524  
 FFTFwd\_RToPerm 524  
 FFTFwd\_RToPerm\_AMRWBE 1075  
 FFTFwd\_RToPerm\_GSMAMR 981  
 FFTGetBufSize\_C 516  
 FFTGetBufSize\_R 516  
 FFTGetSize\_C 514  
 FFTGetSize\_R 514  
 FFTInit\_C 512  
 FFTInit\_R 512  
 FFTInitAlloc\_C 508  
 FFTInitAlloc\_R 508  
 FFTInv\_CCSToR 528  
 FFTInv\_CToC 521

FFTInv\_PackToR 528  
 FFTInv\_PermToR 528  
 FFTInv\_PermToR\_AMRWBE 1076  
 FillShortlist\_Column 748  
 FillShortlist\_Row 745  
 filter  
   inverse 911  
   long-term 910  
   short-term 911  
 FilteredExcitation\_G729 926  
 FilterHighpass\_G7291 929  
 FilterHighpassGetStateSize\_G7291 928  
 FilterHighpassInit\_G7291 928  
 Filtering functions  
   SumWindow 351  
 FilterLowpass\_G7291 930  
 FilterMedian 487  
 FilterNoise 1172  
 FilterNoiseDetect 1169  
 FilterNoiseDetectModerate 1170  
 FilterNoiseGetStateSize 1166  
 FilterNoiseInit 1167  
 FilterNoiseLevel 1168  
 FilterNoiseSetMode 1171  
 FilterUpdateEMNS 835  
 FilterUpdateWiener 836  
 Find 1443  
 FindC 1445  
 FindCAny 1447  
 FindNearest 283  
 FindNearestOne 282  
 FindPeaks 842  
 FindRev 1443  
 FindRevC 1445  
 FindRevCAny 1447  
 finite field, concept 1767  
 FIR 385  
 FIR filter functions  
   FIR 385  
   FIR\_Direct 398  
   FIRFree 362  
   FIRGenBandpass 416  
   FIRGenBandstop 419  
   FIRGenHighpass 414  
   FIRGenLowpass 411  
   FIRGetDlyLine 380  
   FIRGetStateSize 372  
   FIRGetTaps 376  
   FIRInit 363

FIR filter functions (*continued*)

FIRInitAlloc 353  
 FIRMR\_Direct 402  
 FIRMRGetStateSize 372  
 FIRMRInit 367  
 FIRMRInitAlloc 357  
 FIRMRStreamGetStateSize 375  
 FIRMRStreamInit 370  
 FIRMRStreamInitAlloc 360  
 FIROne 383  
 FIROne\_Direct 394  
 FIRSetDlyLine 382  
 FIRSetTaps 378  
 FIRSparse 409  
 FIRSparseGetStateSize 408  
 FIRSparseInit 406  
 FIRStreamGetStateSize 375  
 FIRStreamInit 366  
 FIRStreamInitAlloc 355  
 FIR\_Direct 398  
 FIR\_EC 1161  
 FIRBlockFree 1243  
 FIRBlockInitAlloc 1242  
 FIRBlockOne 1244  
 FIRFree 362  
 FIRGenBandpass 416  
 FIRGenBandstop 419  
 FIRGenHighpass 414  
 FIRGenLowpass 411  
 FIRGenMidBand\_AMRWB 1072  
 FIRGetDlyLine 380  
 FIRGetStateSize 372  
 FIRGetTaps 376  
 FIRInit 363  
 FIRInitAlloc 353  
 FIRLMS 426  
 FIRLMSFree 422  
 FIRLMSGetDlyLine 424  
 FIRLMSGetTaps 423  
 FIRLMSInitAlloc 421  
 FIRLMSMRFree 433  
 FIRLMSMRGetDlyLine 438  
 FIRLMSMRGetDlyVal 440  
 FIRLMSMRGetTapsPointer 437  
 FIRLMSMRInitAlloc 431  
 FIRLMSMROne 441  
 FIRLMSMROneVal 442  
 FIRLMSMRPutVal 441  
 FIRLMSMRSetDlyLine 439

FIRLMSMRSetMu 433  
 FIRLMSMRSetTaps 436  
 FIRLMSMRUpdateTaps 434  
 FIRLMSOne\_Direct 428  
 FIRLMSSetDlyLine 425  
 FIRMR\_Direct 402  
 FIRMRGetStateSize 372  
 FIRMRInit 367  
 FIRMRInitAlloc 357  
 FIRMRStreamGetStateSize 375  
 FIRMRStreamInit 370  
 FIRMRStreamInitAlloc 360  
 FIROne 383  
 FIROne\_Direct 394  
 FIRSetDlyLine 382  
 FIRSetTaps 378  
 FIRSparse 409  
 FIRSparseGetStateSize 408  
 FIRSparseInit 406  
 FIRStreamGetStateSize 375  
 FIRStreamInit 366  
 FIRStreamInitAlloc 355  
 FIRSubband\_EC 1162  
 FIRSubbandCoeffUpdate\_EC 1163  
 FIRSubbandLow\_EC 1162  
 FIRSubbandLowCoeffUpdate\_EC 1163  
 fixed-accuracy arithmetic functions  
   Abs 1509  
   Acos 1561  
   Add 1496  
   Arg 1511  
   arithmetic 1496, 1498, 1500, 1502, 1505, 1507,  
   1509, 1511  
   Atan2 1569  
   Atanh 1584  
   CdfNorm 1591  
   CdfNormInv 1599  
   Ceil 1604  
   CIS 1557  
   Conj 1507  
   Cos 1550  
   Cosh 1571  
   Div 1515  
   Erfc 1588  
   ErfcInv 1596  
   ErfInv 1593  
   Exp 1539  
   Expm1 1542



fixed-accuracy arithmetic functions (*continued*)  
   exponential and logarithmic 1539, 1542, 1544, 1546, 1549  
   Floor 1602  
   hyperbolic 1571, 1574, 1576, 1578, 1581, 1584  
   Hypot 1537  
   Inv 1513  
   InvCbrt 1525  
   InvSqrt 1521  
   Ln 1544  
   Log10 1546  
   Log1p 1549  
   Modf 1612  
   Mul 1502  
   MulByConj 1505  
   NearbyInt 1608  
   Pow 1530  
   Pow2o3 1527  
   Pow3o2 1528  
   power and root 1513, 1515, 1518, 1521, 1523, 1525, 1527, 1528, 1530, 1533, 1537  
   Powx 1533  
   Rint 1610  
   rounding 1602, 1604, 1605, 1607, 1608, 1610, 1612  
   Sin 1552  
   SinCos 1555  
   Sinh 1574  
   special 1586, 1588, 1591, 1593, 1596, 1599  
   Sqrt 1518  
   Sub 1498  
   Tan 1559  
   Tanh 1576  
   trigonometric 1550, 1552, 1555, 1557, 1559, 1561, 1564, 1566, 1569  
   Trunc 1605  
 FixedCodebookDecode\_GSMAMR 1011  
 FixedCodebookSearch\_G729 893  
 FixedCodebookSearch\_RTA 1201  
 FixedCodebookSearchRandom\_RTA 1201  
 Flip 281  
 Floor 1602  
 font conventions 61  
 FormVector 791  
 FormVectorVQ 804  
 fraction delay 891  
 frequency domain prediction 1247  
 FullbandController\_EC 1159  
 FullbandControllerGetSize\_EC 1156

FullbandControllerInit\_EC 1157  
 FullbandControllerReset\_EC 1160  
 FullbandControllerUpdate\_EC 1158  
 function descriptions 55

## G

G.722 speech codec  
   QMFDcode\_G722 1184  
   QMFEcode\_G722 1181  
   SBADPCMDcode\_G722 1183  
   SBADPCMDcodeInit\_G722 1182  
   SBADPCMDcodeStateSize 1182  
   SBADPCMEncode\_G722 1180  
   SBADPCMEncodeInit\_G722 1179  
   SBADPCMEncodeStateSize\_G722 1179  
 G.722.1 speech codec  
   DCTFwd\_G722, DCTFwd\_G7221 1105  
   DCTInv\_G722, DCTInv\_G7221 1106  
   DecomposeDCTToMLT 1108  
   DecomposeMLTToDCT 1107  
   HuffmanEncode\_G722 1110  
 G.723.1 speech codec  
   codebook search functions 960, 961, 963, 964, 966  
   filter functions 970, 971, 973, 974, 975, 976, 977  
   gain quantization functions 967, 969  
   LP analysis functions 952, 953, 954, 956, 957, 958, 959  
 G.726 speech codec  
   Decode\_G726 1115  
   DecodeGetStateSize\_G726 1113  
   DecodeInit\_G726 1114  
   Encode\_G726 1112  
   EncodeGetStateSize\_G726 1111  
   EncodeInit\_G726 1112  
 G.728 speech codec  
   CodebookSearch\_G728 1133  
   CodebookSearchTCQ\_G728 1134  
   CombinedFilter\_G728 1122  
   CombinedFilterGetStateSize\_G728 1121  
   CombinedFilterInit\_G728 1121  
   IIR\_G728 1118  
   IIR\_Init\_G728 1117  
   IIRGetStateSize\_G728 1117  
   ImpulseResponseEnergy\_G728 1135  
   LevinsonDurbin\_G728 1132  
   LPCInverseFilter\_G728 1127  
   PitchPeriodExtraction\_G728 1128

G.728 speech codec (*continued*)

- PostFilter\_G728 1124
- PostFilterAdapterGetStateSize\_G728 1126
- PostFilterAdapterStateInit\_G728 1126
- PostFilterGetStateSize\_G728 1123
- PostFilterInit\_G728 1124
- SynthesisFilter\_G728 1120
- SynthesisFilterGetStateSize\_G728 1119
- SynthesisFilterInit\_G728 1119
- WinHybrid\_G728 1130
- WinHybridGetStateSize\_G728 1129
- WinHybridInit\_G728 1129

G.729 Functions 869

G.729 speech codec

- basic functions 871, 872
- codebook search functions 889, 890, 891, 892, 893, 898, 904, 905
- filter functions 906, 908, 910, 911, 913, 914, 915, 916, 917, 919, 920, 921, 922, 923, 925, 926
- gain quantization functions 900, 901, 903
- LP analysis functions 873, 875, 877, 879, 880, 881, 882, 883, 884, 888

G.729.1 speech codec

- AdaptiveCodebookGain\_G7291 936
- AdaptiveCodebookSearch\_G7291 934
- AlgebraicCodebookSearchL1\_G7291 937
- AlgebraicCodebookSearchL2\_G7291 937
- CompressEnvelopTime\_G7291 944
- EnvelopFrequency\_G7291 940
- EnvelopTime\_G7291 939, 942
- FilterHighpass\_G7291 929
- FilterHighpassGetStateSize\_G7291 928
- FilterHighpassInit\_G7291 928
- FilterLowpass\_G7291 930
- GainControl\_G7291 948
- GainQuant\_G7291 938
- GenerateExcitationGetStateSize\_G729 940
- GenerateExcitationInit\_G7291 941
- LSFDecode\_G7291 934
- MDCTFwd\_G7291 945
- MDCTInv\_G7291 945, 946
- MDCTPostProcess\_G7291 947
- MDCTQuantInv\_G7291 947
- QMFDecode\_G7291 933
- QMFEncode\_G7291 932
- QMFGetStateSize\_G7291 931
- QMFInit\_G7291 931
- ShapeEnvelopTime\_G7291 942, 943, 950
- TiltCompensation\_G7291 949

gain

- estimation 967
- of the adaptive-codebook vector 905
- optimum 903
- GainControl\_G723 969
- GainControl\_G729 901
- GainControl\_G7291 948
- GainDecodeTCX\_AMRWBE 1094
- GainQuant\_AMRWB 1059
- GainQuant\_AMRWBE 1090
- GainQuant\_G723 967
- GainQuant\_G729 903
- GainQuant\_G7291 938
- GainQuantTCX\_AMRWBE 1092
- Gaussian distribution functions 148, 149, 150, 151, 152
- RandGauss 151
- RandGauss\_Direct 152
- RandGaussFree 149
- RandGaussGetSize 151
- RandGaussInit 150
- RandGaussInitAlloc 148
- GaussianDist 765
- GaussianMerge 767
- GaussianSplit 766
- Generalized Interval Transformation coding 1723
- GenerateExcitationGetStateSize\_G729 940
- GenerateExcitationInit\_G7291 941
- GetCdbkSize 799
- GetCodebook 799
- GetSizeMCRA 838
- GetVarPointDV 284
- GFAAdd 1770
- GFDiv 1773
- GFEExpAlpha 1778
- GFFGet\_Size 1768
- GFINit 1769
- GFINv 1775
- GFLogAlpha 1777
- GFMul 1772
- GFNeg 1776
- GFPow 1774
- GFSUB 1771
- GITFree 1725
- Goertz 557
- GoertzTwo 560
- GSM FR speech codec
  - Deemphasize\_GSMFR 1098
  - HighPassFilter\_GSMFR 1101

GSM FR speech codec (*continued*)

- Preemphasize\_GSMFR 1103
- RPEQuantDecode\_GSMFR 1097
- Schur\_GSMFR 1102
- ShortTermAnalysisFilter\_GSMFR 1099
- ShortTermSynthesisFilter\_GSMFR 1100
- WeightingFilter\_GSMFR 1102
- GSM-AMR speech codec 850, 978, 980, 981, 982, 984, 985, 986, 988, 990, 991, 992, 994, 996, 999, 1002, 1004, 1005, 1007, 1009, 1011, 1012, 1013, 1015, 1018, 1019, 1021, 1022
  - adaptive codebook functions 991
  - AdaptiveCodebookDecode\_GSMAMR 1005
  - AdaptiveCodebookGain\_GSMAMR 1007
  - AdaptiveCodebookSearch\_GSMAMR 1004
  - AlgebraicCodebookSearch\_GSMAMR 1009
  - AutoCorr\_GSMAMR 982
  - data structures 850
  - DecDTXBuffer\_GSMAMR 1021
  - discontinuous transmission (DTX) primitives 1012
  - EncDTXBuffer\_GSMAMR 1021
  - EncDTXHandler\_GSMAMR 1019
  - EncDTXSID\_GSMAMR 1018
  - FFTFwd\_RToPerm\_GSMAMR 981
  - fixed codebook primitives 1009
  - FixedCodebookDecode\_GSMAMR 1011
  - ImpulseResponseTarget\_GSMAMR 1002
  - Interpolate\_GSMAMR 980
  - LevinsonDurbin\_GSMAMR 984
  - LP analysis and quantization functions 982
  - LPCToLSP\_GSMAMR 985
  - LSFToLSP\_GSMAMR 988
  - LSPQuant\_GSMAMR 988
  - LSPToLPC\_GSMAMR 986
  - open-loop pitch search 991, 992
  - OpenLoopPitchSearchDTXVAD1\_GSMAMR 996
  - OpenLoopPitchSearchDTXVAD2\_GSMAMR 999
  - OpenLoopPitchSearchNonDTX\_GSMAMR 994
  - PostFilter\_GSMAMR 1022
  - Preemphasize\_GSMAMR 1012
  - QuantLSPDecode\_GSMAMR 990
  - VAD1\_GSMAMR 1013
  - VAD2\_GSMAMR 1015
- GZIP 1666

**H**

- Hamming window 952
- hardware and software requirements 51
- HarmonicFilter 914
- HarmonicNoiseSubtract\_G723 975
- HarmonicSearch\_G723 975
- Hartley 562
- Hash 1462
- HighBandCoding\_Aurora 826
- HighPassFilter\_AMRWB 1034
- HighPassFilter\_G723 970
- HighPassFilter\_G729 916
- HighPassFilter\_GSMFR 1101
- HighPassFilter\_RTA 1204
- HighPassFilterGetDlyLine\_AMRWB 1035
- HighPassFilterGetSize\_AMRWB 1032
- HighPassFilterInit\_AMRWB 1033
- HighPassFilterInit\_G729 916
- HighPassFilterSetDlyLine\_AMRWB 1036
- HighPassFilterSize\_G729 915
- Hilbert 588
- Hilbert transform functions
  - Hilbert 588
  - HilbertFree 587
  - HilbertInitAlloc 586
- HilbertFree 587
- HilbertInitAlloc 586
- HuffFree 1642
- HuffGetLenCodeTable 1646
- HuffGetSize 1643
- HuffLenCodeTablePack 1651
- HuffLenCodeTableUnpack 1652
- Huffman coding
  - DecodeHuff 1649
  - DecodeHuffInit 1648
  - DecodeHuffInitAlloc 1647
  - DecodeHuffOne 1648
  - EncodeHuff 1645
  - EncodeHuffFinal 1646
  - EncodeHuffInit 1643
  - EncodeHuffInitAlloc 1641
  - EncodeHuffOne 1644
  - HuffFree 1642
  - HuffGetDstBuffSize 1650
  - HuffGetLenCodeTable 1646
  - HuffGetSize 1643
  - HuffLenCodeTablePack 1651
  - HuffLenCodeTableUnpack 1652

HuffmanDecode\_MP3 1323  
HuffmanEncode\_G722 1110  
HuffmanEncode\_MP3 1310  
Hypot 1537

## I

IIR 466  
IIR filter functions  
    IIR 466  
    IIRGetDlyLine 461  
    IIRInitAlloc 446  
    IIRInitAlloc\_BiQuad 448  
    IIRInitAlloc\_BiQuad\_DF1 448  
    IIROne 464  
    IIRSetDlyLine 463  
    IIR\_BiQuadDirect 478  
    IIR\_Direct 477  
    IIRFree 450  
    IIRGetStateSize 457  
    IIRGetStateSize\_BiQuad 458  
    IIRGetStateSize\_BiQuad\_DF1 458  
    IIRInit 452  
    IIRInit\_BiQuad 454  
    IIRInit\_BiQuad\_DF1 454  
    IIROne\_BiQuadDirect 475  
    IIROne\_Direct 474  
    IIRSetTaps 459  
    IIRSparse 482  
    IIRSparseGetStateSize 481  
    IIRSparseInit 479  
IIR\_BiQuadDirect 478  
IIR\_Direct 477  
IIR\_G728 1118  
IIR\_Init\_G728 1117  
IIR16s\_G723 971  
IIR16s\_G729 917  
IIRFree 450  
IIRGenHighpass 484  
IIRGetDlyLine 461  
IIRGetStateSize\_BiQuad 458  
IIRGetStateSize\_BiQuad\_DF1 458  
IIRGetStateSize\_G728 1117  
IIRInit 452  
IIRInit\_BiQuad 454  
IIRInit\_BiQuad\_DF1 454  
IIRInitAlloc 446  
IIRInitAlloc\_BiQuad 448

IIRInitAlloc\_BiQuad\_DF1 448  
IIROne 464  
IIROne\_BiQuadDirect 475  
IIROne\_Direct 474  
IIRSetDlyLine 463  
IIRSetTaps 459  
IIRSparse 482  
IIRSparseGetStateSize 481  
IIRSparseInit 479  
Imag 253  
ImpulseResponseEnergy\_G728 1135  
ImpulseResponseTarget\_GSMAMR 1002  
Inflate 1703  
InflateBuildHuffTable 1702  
InitMCRA 838  
Insert 1448  
Intel Performance Primitives software 51  
Interleave 1213  
interleaved to multi-row format conversion 1213  
internationalization functions  
    ippGetMessageStatusI18n 109  
    ippMessageCatalogCloseI18n 108  
    ippOpenMessageCatalogI18n 107  
    ippStatusToMessageIdI18n 110  
Interpolate\_G729 872  
Interpolate\_GSMAMR 980  
InterpolateC\_NR 853  
Inv 1513  
InvCbrt 1525  
Inversion frequencies 1723  
InvSqrt 860, 1521  
ippAlignPtr 100  
ippCloseMessageCatalogI18n 108  
ippEnableCpu 106  
ippFree 103  
ippGetCpuClocks 94  
ippGetCpuFeatures 95  
ippGetCpuFreqMhz 95  
ippGetCpuType 92  
ippGetGetMaxCacheSizeB 98  
ippGetMessageStatusI18n 109  
ippGetNumCoresOnDie 98  
ippGetNumThreads 102  
ippGetStatusString 91  
ippgHartley 562  
ippgWHT 564  
ippInit 104  
ippInitCpu 106  
ippMalloc 102

---

ippMessageCatalogOpenI18n 107  
ippsAbs 1509  
ippsAcos 1561  
ippsAcosh 1578  
ippsAdaptiveCodebookGain\_G7291() 936  
ippsAdaptiveCodebookSearch\_G7291 934  
ippsAdd 1496  
ippsALC\_G169() 1177  
ippsALCGetStateSize\_G169() 1175  
ippsALCInit\_G169() 1175  
ippsALCSetGain\_G169() 1177  
ippsALCSetLevel\_G169() 1176  
ippsAlgebraicCodebookSearchL1\_G7291 937  
ippsAlgebraicCodebookSearchL2\_G7291 937  
ippsAnalysisFilterInitAlloc\_SBR 1400  
ippsArg 1511  
ippsAsin 1564  
ippsAsinh 1581  
ippsAtan 1566  
ippsAtan2 1569  
ippsAtanh 1584  
ippsCbrt 1523  
ippsCdfNorm 1591  
ippsCdfNormInv 1599  
ippsCeil 1604  
ippsCIS 1557  
ippsCompressEnvelopTime\_G7291() 944  
ippsConj 1507  
ippsConvBiased 337  
ippsCos 1550  
ippsCosh 1571  
ippsDecodeChanPairElt\_AAC 1350  
ippsDecodeChanPairElt\_MP4\_AAC 1376  
ippsDecodeDatStrElt\_AAC 1355  
ippsDecodeExtensionHeader\_AAC 1373  
ippsDecodeIsStereo\_AAC 1361  
ippsDecodeMainHeader\_AAC 1372  
ippsDecodeMsStereo\_AAC 1359  
ippsDecodePNS\_AAC 1374, 1375  
ippsDecodePrgCfgElt\_AAC 1350  
ippsDecodeTNS\_AAC 1364  
ippsDecodeZ1Z2\_BZ 1752  
ippsDeinterleaveSpectrum\_AAC 1363  
ippsDiv 1515  
ippsEncodeTNS\_AAC 1383  
ippsEnvelopFrequency\_G7291() 940  
ippsEnvelopTime\_G7291() 939  
ippsErf 1586  
ippsErfc 1588  
ippsErfcInv 1596  
ippsErfInv 1593  
ippsSetDenormAreZeros 100  
ippsSetFlushToZero 99  
ippsSetNumThreads 101  
ippsExp 1539  
ippsExpm1 1542  
ippsFilterHighpass\_G7291() 929  
ippsFilterHighpassGetStateSize\_G7291 928  
ippsFilterHighpassInit\_G7291 928  
ippsFilterLowpass\_G7291 930  
ippsFIRGetDlyLine 380  
ippsFIRSetDlyLine 382  
ippsFloor 1602  
ippsFree 90  
ippsGainControl\_G7291() 948  
ippsGainQuant\_G7291() 938  
ippsGenerateExcitation\_G7291() 942  
ippsGenerateExcitationGetStateSize\_G7291() 940  
ippsGenerateExcitationInit\_G7291() 941  
ippsGetLibVersion 87  
ippsGFAdd 1770  
ippsGFDiv 1773  
ippsGFExpAlpha 1778  
ippsGFGetSize 1768  
ippsGFInit 1769  
ippsGFInv 1775  
ippsGFLogAlpha 1777  
ippsGFMul 1772  
ippsGFNeg 1776  
ippsGFPow 1774  
ippsGFSub 1771  
ippsHuffmanEncode\_MP3 1310  
ippsHypot 1537  
ippsInitBitReservoir\_MP3 1316  
ippsInv 1513  
ippsInvCbrt 1525  
ippsInvSqrt 1521  
ippsLevinsonDurbin\_GSMAMR 984  
ippsLn 1544  
ippsLog10 1546  
ippsLog1p 1549  
ippsLongTermPredict\_AAC 1384  
ippsLongTermReconstruct\_AAC 1378  
ippsLPCToLSP\_GSMAMR\_16s 985  
ippsLSFDecode\_G7291() 934  
ippsLSPQuant\_GSMAMR\_16s 988  
ippsLSPToLPC\_GSMAMR\_16s 986  
ippsLtpUpdate\_AAC 1387

ippsMalloc 89  
ippsMDCTFwd\_AAC 1379, 1380  
ippsMDCTFwd\_G7291() 945  
ippsMDCTInv\_AAC 1368  
ippsMDCTInv\_G7291() 945, 946  
ippsMDCTPostProcess\_G7291() 947  
ippsMDCTQuantInv\_G7291() 947  
ippsModf 1612  
ippsMul 1502  
ippsMulByConj 1505  
ippsNearbyInt 1608  
ippsNoiselessDecode\_AAC 1385  
ippsNoiselessDecoder\_LC\_AAC 1352  
ippsPackFrameHeader\_MP3 1313  
ippsPolyGFAdd 1785  
ippsPolyGFCopy 1783  
ippsPolyGFDerive 1792  
ippsPolyGFDiv 1788  
ippsPolyGFGCD 1794  
ippsPolyGFGetRef 1784  
ippsPolyGFGetSize 1799  
ippsPolyGFInit 1780  
ippsPolyGFIrreducible 1790  
ippsPolyGFMod 1786  
ippsPolyGFMul 1787  
ippsPolyGFPrimitive 1791  
ippsPolyGFRoots 1793  
ippsPolyGFSetCoeffs 1781  
ippsPolyGFSetDegree 1782  
ippsPolyGFShlC 1789  
ippsPolyGFShrC 1789  
ippsPolyGFSub\_8u() 1785  
ippsPolyGFValue 1792  
ippsPow 1530  
ippsPow2o3 1527  
ippsPow3o2 1528  
ippsPowx 1533  
ippsQMF\_G7291 931  
ippsQMFFDecode\_G7291 933  
ippsQMFFEncode\_G7291 932  
ippsQMFFGetStateSize\_G7291 931  
ippsQuantInv\_AAC\_32s\_I 1357  
ippsQuantLSPDecode\_GSMAMR\_16s 990  
ippsQuantParam\_G7291() 950  
ippsRegExpMultiAdd 1480  
ippsRegExpMultiDelete 1481  
ippsRegExpMultiFind 1483  
ippsRegExpMultiFree 1480  
ippsRegExpMultiGetSize 1478

ippsRegExpMultiInit 1478  
ippsRegExpMultiInitAlloc 1479  
ippsRegExpMultiModify 1482  
ippsRegExpReplace 1490  
ippsRegExpReplaceGetSize 1488  
ippsRegExpReplaceInit 1489  
ippsRint 1610  
ippsRound 1607  
ippsRSDecodeBMGetBufferSize 1801  
ippsRSDecodeEEGetBufferSize 1801  
ippsRSDecodeGetSize 1799  
ippsRSDecodeInit 1800  
ippsRSEncode 1798  
ippsRSEncodeGetBufferSize 1797  
ippsRSEncodeGetSize 1795  
ippsRSEncodeInit 1796  
ippsShapeEnvelopTime\_G7291() 942, 943  
ippsSin 1552  
ippsSinCos 1555  
ippsSinh 1574  
ippsSqr 1500  
ippsSqrt 1518  
ippsSub 1498  
ippsTan 1559  
ippsTanh 1576  
ippStaticInit 103  
ippStaticInitCpu 105  
ippStatusToMessageIdI18n 110  
ippsTiltCompensation\_G7291() 949  
ippsTrunc 1605  
ippsUnpackADIFHeader\_AAC 1348  
ippsUnpackADTSFrameHeader\_AAC 1349  
ISFQuant\_AMRWB 1055  
ISFQuantDecode\_AMRWB 1057  
ISFQuantDecode\_AMRWBE 1087  
ISFQuantDecodeDTX\_AMRWB 1059  
ISFQuantDecodeHighBand\_AMRWBE 1088  
ISFQuantDTX\_AMRWB 1058  
ISFQuantHighBand\_AMRWBE 1089  
ISFTToISP\_AMRWB 1030  
ISPTToISF\_Norm\_AMRWB 1029  
ISPTToLPC\_AMRWB 1028

## J

Join 239  
JoinScaled 241

**L**

LagWindow\_G729 889  
LevinsonDurbin\_G723 954  
LevinsonDurbin\_G728 1132  
LevinsonDurbin\_G729 875  
LevinsonDurbin\_GSMAMR 984  
LevinsonDurbin\_RTA 1194  
line spectral frequencies 879  
linear predictors 1242  
LinearPrediction 646  
LinearToMel 666  
LinToALaw 1189  
LinToMuLaw 1187  
Ln 214, 1544  
Log10 1546  
Log1p 1549  
LogAdd 714  
LogGauss 724  
LogGaussAdd 735  
LogGaussAddMultiMix 737  
LogGaussMax 729  
LogGaussMaxMultiMix 732  
LogGaussMixture 739  
LogGaussMixtureSelect 742  
LogGaussMultiMix 727  
LogGaussSingle 721  
Logical and shift functions  
    And 167  
    AndC 166  
    LShiftC 173  
    Not 172  
    Or 169  
    OrC 168  
    RShiftC 174  
    Xor 171  
    XorC 170  
LogSub 715  
LogSum 716  
LongTermPostFilter\_G729 910  
LongTermPredict\_AAC 1384  
LongTermReconstruct\_AAC 1378  
Lowercase 1460  
LowercaseLatin 1460  
LowHighFilter\_Aurora 825  
LP coefficients 883  
LPCInverseFilter\_G728 1127  
LPCToISP\_AMRWB 1026  
LPCToISP\_AMRWBE 1070  
LPCToLSF\_G723 956  
LPCToLSP\_G729 877  
LPCToLSP\_GSMAMR 985  
LPCToLSP\_RTA 1193  
LPToCepstrum 653  
LPToLSP 663  
LPToReflection 655  
LPToSpectrum 652  
LSFDecode\_G723 958  
LSFDecode\_G729 881  
LSFDecodeErased\_G729 882  
LSFQuant\_G723 959  
LSFQuant\_G729 880  
LSFToLPC\_G723 957  
LSFToLSP\_G729 879  
LSFToLSP\_GSMAMR 988  
LShiftC 173  
LSP coefficients 888  
LSPQuant\_G729 884  
LSPQuant\_GSMAMR 988  
LSPQuant\_RTA 1203  
LSPToLP 664  
LSPToLPC\_G729 883  
LSPToLPC\_GSMAMR 986  
LSPToLPC\_RTA 1194  
LSPToLSF\_G729 888  
LtpUpdate\_AAC 1387  
LZ77 dictionary-based compression 1666  
LZ77Free 1673  
LZO compression  
    DecodeLZO 1710  
    DecodeLZOSafe 1711  
    EncodeLZO 1709  
    EncodeLZOGetSize 1707  
    EncodeLZOInit 1708  
LZSS compression  
    EncodeLZSSInitAlloc 1653  
LZSS compression algorithm 1653  
LZSSFree 1654  
LZSSGetSize 1655

**M**

MA predictor 880  
Magnitude 246  
MagSquared 248  
MahDist 718  
MahDistMultiMix 720

MahDistSingle 717  
MakeFloat 1230  
mantissa conversion and scaling functions 1228  
manual organization 53  
MatVecMul 640  
Max 303  
MaxAbs 305  
MaxAbsIndx 306  
MaxEvery 326  
MaxIndx 304  
MaxOrder 279  
MDCTFwd\_AAC 1379, 1380  
MDCTFwd\_G7291 945  
MDCTFwd\_MP3 1293  
MDCTFwd, MDCTInv 1238  
MDCTFwdFree, MDCTInvFree 1235  
MDCTFwdGetBufSize, MDCTInvGetBufSize 1237  
MDCTFwdGetSize, MDCTInvGetSize 1236  
MDCTFwdInit, MDCTInvInit 1234  
MDCTFwdInitAlloc, MDCTInvInitAlloc 1232  
MDCTInv\_AAC 1370  
MDCTInv\_AAC\_32s16s 1368  
MDCTInv\_G7291 945, 946  
MDCTInv\_MP3 1327  
MDCTPostProcess\_G7291 947  
MDCTQuantInv\_G7291 947  
Mean 313  
MeanColumn 752  
MeanSrdDev 316  
MeanVarAcc 764  
MeanVarColumn 754  
Median filter functions  
    FilterMedian 487  
MelFBankGetSize 668  
MelFBankInit 669  
MelFBankInitAlloc 670  
MelFBankInitAlloc\_Aurora 822  
MelLinFBankInitAlloc 672  
MelToLinear 665  
memory allocation functions  
    ippsFree 90  
    ippsMalloc 89  
Min 307  
MinAbs 309  
MinAbsIndx 310  
MinEvery 326  
MinIndx 308  
MinMax 311  
MinMaxIndx 312  
  
MMX technology 51  
Model adaptation functions  
    AddMulColumn 780  
    AddMulRow 781  
    DotProdColumn 783  
    MulColumn 784  
    QRTransColumn 782  
    SumColumnAbs 785  
    SumColumnSqr 786  
    SumRowAbs 787  
    SumRowSqr 787  
    SVD 788  
    SVDSort 788  
    WeightedSum 790  
Model estimation functions  
    BhatDist 771  
    DcsClustLAccumulate 778  
    DcsClustLCompute 779  
    Entropy 768  
    ExpNegSqr 770  
    GaussianDist 765  
    GaussianMerge 767  
    GaussianSplit 766  
    MeanColumn 752  
    MeanVarAcc 764  
    MeanVarColumn 754  
    NormalizeColumn 761  
    NormalizeInRange 762  
    OutProbPreCalc 776  
    SinC 769  
    UpdateGConst 775  
    UpdateMean 772  
    UpdateVar 773  
    UpdateWeight 774  
    VarColumn 753  
    WeightedMeanColumn 755  
    WeightedMeanVarColumn 757  
    WeightedVarColumn 756  
Model evaluation functions  
    AddNRows 712  
    BuildSignTable 744  
    DTW 750  
    FillShortlist\_Column 748  
    FillShortlist\_Row 745  
    LogAdd 714  
    LogGauss 724  
    LogGaussAdd 735  
    LogGaussAddMultiMix 737  
    LogGaussMax 729



- Model evaluation functions (*continued*)
    - LogGaussMaxMultiMix 732
    - LogGaussMixture 739
    - LogGaussMixtureSelect 742
    - LogGaussMultiMix 727
    - LogGaussSingle 721
    - LogSub 715
    - LogSum 716
    - MahDist 718
    - MahDistMultiMix 720
    - MahDistSingle 717
    - ScaleLM 713
  - Modf 1612
  - modified discrete cosine transform (MDCT) 1232
  - Move 117
  - Move To Front data transform method 1738
  - MP3 audio decoder 1317
  - MP3 audio decoder functions
    - HuffmanDecode\_MP3 1323
    - MDCTInv\_MP3 1327
    - ReQuantize\_MP3 1325
    - SynthesisFilter\_PQMF\_MP3 1333
    - SynthesisFilterFree\_PQMF\_MP3 1332
    - SynthesisFilterGetSize\_PQMF\_MP3 1332
    - SynthesisFilterInit\_PQMF\_MP3 1330
    - SynthesisFilterInitAlloc\_PQMF\_MP3 1331
    - SynthPQMF\_MP3 1329
    - UnpackFrameHeader\_MP3 1318
    - UnpackScaleFactors\_MP3 1321
    - UnpackSideInfo\_MP3 1319
  - MP3 audio encoder
    - bit reservoir structure 1281
    - psychoacoustic model 1281
    - psychoacoustic model structure 1281
  - MP3 audio encoder API
    - enumerated types 1286
  - MP3, data structures 1281
  - MP3, macro and constant definitions 1280
  - MPEG-2
    - global macros 1334
  - MPMLQFixedCodebookSearch\_G723 964
  - MTF data transform
    - MTFFree 1739
    - MTFGetSize 1740
    - MTFInit 1740
    - MTFInitAlloc 1739
  - MTFFree 1739
  - MTFFwd 1741
  - MTFGetSize 1740
  - MTFInit 1740
  - MTFInitAlloc 1739
  - MTFInv 1742
  - Mul 186, 1502
  - Mul\_NR 854
  - MuLawToALaw 1190
  - MuLawToLin 1186
  - MulByConj 1505
  - MulC 184
  - MulC\_NR 856
  - MulColumn 784
  - MulPack 503
  - MulPackConj 507
  - MulPerm 505
  - MulPowerC\_NR 857
  - multi-rate FIR LMS filter functions
    - FIRLMSMRFree 433
    - FIRLMSMRGetDlyLine 438
    - FIRLMSMRGetDlyVal 440
    - FIRLMSMRGetTaps 435
    - FIRLMSMRGetTapsPointer 437
    - FIRLMSMRInitAlloc 431
    - FIRLMSMROne 441
    - FIRLMSMROneVal 442
    - FIRLMSMRPutVal 441
    - FIRLMSMRSetDlyLine 439
    - FIRLMSMRSetMu 433
    - FIRLMSMRSetTaps 436
    - FIRLMSMRUpdateTaps 434
  - Multiplication of packed data
    - MulPackConj 507
    - MulPerm 503, 505
- ## N
- NearbyInt 1608
  - NewVar 692
  - NLMS\_EC 1165
  - noise reduction
    - FilterNoise 1172
    - FilterNoiseDetect 1169
    - FilterNoiseDetectModerate 1170
    - FilterNoiseGetStateSize 1166
    - FilterNoiseInit 1167
    - FilterNoiseLevel 1168
    - FilterNoiseSetMode 1171
  - NoiselessDecode\_AAC 1385
  - NoiselessDecoder\_LC\_AAC 1352

NoiseSpectrumUpdate\_Aurora 820  
Norm 318  
Normalize 219  
NormalizeColumn 761  
NormalizeInRange 762  
NormDiff 320  
NormEnergy 688  
Not 172  
notational conventions 60  
NthMaxElement 637

## O

OpenLoopPitchSearch\_AMRWB 1031  
OpenLoopPitchSearch\_AMRWB 1069  
OpenLoopPitchSearch\_G723 960  
OpenLoopPitchSearch\_G729 890  
OpenLoopPitchSearchDTXVAD1\_GSMAMR 996  
OpenLoopPitchSearchDTXVAD2\_GSMAMR 999  
OpenLoopPitchSearchNonDTX\_GSMAMR 994  
Or 169  
OrC 168  
out-of-order DFT 532  
OutProbPreCalc 776

## P

PackBits 116  
packed format for Fourier transform 494  
    pack format 494  
    CCS format 494  
    perm format 494  
PackFrameHeader\_MP3 1313  
PackHuffContext\_BZ2 1758  
PackScalefactors\_MP3 1308  
PackSideInfo\_MP3 1314  
parallelism 51  
Periodicity 845  
PeriodicityLSPE 843  
Phase, complex 249  
PhaseDispersion\_G729D 920  
PhaseDispersionGetStateSize\_G729D 919  
PhaseDispersionInit\_G729D 919  
PhaseDispersionUpdate\_G729D 920  
pitch 890, 960, 963  
    close loop 963  
    open loop 960

pitch super resolution functions  
    CrossCorrCoeff 709  
    CrossCorrCoeffDecim 707  
    CrossCorrCoeffInterpolation 710  
PitchmarkToF0Cand 660  
PitchPeriodExtraction\_G728 1128  
PitchPostFilter\_G723 977  
platforms supported 52  
PolarToCart, complex 278  
PolyGFAdd 1785  
PolyGFCopy 1783  
PolyGFDiv 1788  
PolyGFGCD 1794  
PolyGFGetRef 1784  
PolyGFGetSize 1779  
PolyGFInit 1780  
PolyGFIrreducible 1790  
PolyGFMod 1786  
PolyGFMul 1787  
PolyGFPrimitive 1791  
PolyGFRoots 1793  
PolyGFSetCoeffs 1781  
PolyGFSetDegree 1782  
PolyGFSetDerive 1792  
PolyGFShlC 1789  
PolyGFShrC 1789  
PolyGFSub 1785  
PolyGFValue 1792  
polynomial coefficients 956  
Polyphase functions  
    ResamplePolyphase 815  
    ResamplePolyphaseFree 814  
    ResamplePolyphaseGetFilter 809  
    ResamplePolyphaseGetSize 807  
    ResamplePolyphaseInit 806  
    ResamplePolyphaseInitAlloc 812  
    ResamplePolyphaseSetFilter 808  
post filter 977  
PostFilter\_G728 1124  
PostFilter\_GSMAMR 1022  
PostFilter\_RTA 1199  
PostFilterAdapterGetStateSize\_G728 1126  
PostFilterAdapterStateInit\_G728 1126  
PostFilterGetStateSize\_G728 1123  
PostFilterGetStateSize\_RTA 1198  
PostFilterInit\_G728 1124  
PostFilterInit\_RTA 1199  
PostFilterLowBand\_AMRWB 1073  
Pow 1530

Pow2o3 1527  
Pow34 1217  
Pow3o2 1528  
Pow43 1221  
Pow43Scale 1222  
power weighting 857  
PowerSpectr, complex 250  
Powx 1533  
PredictCoef\_SBR 1429  
prediction error 903  
prediction, in frequency domain 1247  
PredictOneCoef\_SBR 1431  
Preemphasize 280  
Preemphasize\_AMRWB 1037  
Preemphasize\_G729 921  
Preemphasize\_GSMAMR 1012  
Preemphasize\_GSMFR 1103  
prequantization 1217  
PreSelect\_VQ 1272  
PsychoacousticModelTwo\_MP3 1295

## Q

QMFDcode\_G722 1184  
QMFDcode\_G7291 933  
QMFDcode\_RTA 1196  
QMFEcode\_G722 1181  
QMFEcode\_G7291 932, 934, 936  
QMFEcode\_RTA 1197  
QMFGgetStateSize\_G7291 931  
QMFGgetStateSize\_RTA 1195  
QMFINit\_G7291 931  
QMFINit\_RTA 1196  
QRTransColumn 782  
QuantInv\_AAC 1357  
quantization  
    inverse 958  
    in speech codecs 880  
    process 884  
QuantLSPDecode\_GSMAMR 990  
QuantParam\_G7291 950  
QuantTCX\_AMRWBE 1091

## R

RandGauss 151  
RandGauss\_Direct 152

RandGaussFree 149  
RandGaussGetSize 151  
RandGaussInit 150  
RandGaussInitAlloc 148  
RandomNoiseExcitation\_G729B 925  
RandUniform 144  
RandUniform\_Direct 147  
RandUniformFree 142  
RandUniformGetSize 144  
RandUniformInit 143  
RandUniformInitAlloc 141  
Real 252  
RealToCplx 254  
RecSqrt 693  
ReduceDictionary 1753  
ReflectionToAR 657  
ReflectionToLAR 657  
ReflectionToLP 656  
ReflectionToTilt 659  
ReflectionToTrueAR 657  
RegExpFind 1473  
RegExpFree 1470  
RegExpGetSize 1472  
RegExpInit 1470  
RegExpInitAlloc 1469  
RegExpMultiAdd 1480  
RegExpMultiDelete 1481  
RegExpMultiFind 1483  
RegExpMultiFree 1480  
RegExpMultiGetSize 1478  
RegExpMultiInit 1478  
RegExpMultiInitAlloc 1479  
RegExpMultiModify 1482  
RegExpReplace 1490  
RegExpReplaceGetSize 1488  
RegExpReplaceInit 1489  
RegExpSetMatchLimit 1472  
regular expressions 1468  
related publications 60  
Remove 1450  
ReplaceC 1458  
ReQuantize\_MP3 1325  
ResamplePolyphase 815  
ResamplePolyphase\_AMRWBE 1085  
ResamplePolyphaseFree 814  
ResamplePolyphaseGetFilter 809  
ResamplePolyphaseGetSize 807  
ResamplePolyphaseInit 806  
ResamplePolyphaseInitAlloc 812

ResamplePolyphaseSetFilter 808  
residual signal 906  
ResidualFilter\_AMRWB 1025  
ResidualFilter\_Aurora 824  
ResidualFilter\_G729 906  
RFC1950 1666, 1679, 1685, 1688  
RFC1951 1666, 1679, 1685, 1688, 1689, 1702, 1703  
RFC1952 1666, 1679, 1685, 1688  
Rice-Colomb coding 1723  
Rint 1610  
RLEFree\_BZ2 1747  
RLEGetInUseTable 1750  
RLEGetSize\_BZ2 1748  
Round 1607  
rounding mode, in speech codec functions 848  
RPEQuantDecode\_GSMFR 1097  
RSDDecodeBM 1802  
RSDDecodeBMGetBufferSize 1801  
RSDDecodeEE 1802  
RSDDecodeEEGetBufferSize 1801  
RSDDecodeGetSize 1799  
RSDDecodeInit 1800  
RSEncode 1798  
RSEncodeGetBufferSize 1797  
RSEncodeGetSize 1795  
RSEncodeInit 1796  
RShiftC 174  
RT Audio  
    AdaptiveCodebookSearch\_RTA 1200  
    BandPassFilter\_RTA 1205  
    FixedCodebookSearch\_RTA 1201  
    FixedCodebookSearchRandom\_RTA 1201  
    HighPassFilter\_RTA 1204  
    LevinsonDurbin\_RTA 1194  
    LPCToLSP\_RTA 1193  
    LSPQuant\_RTA 1203  
    LSPToLPC\_RTA 1194  
    PostFilter\_RTA 1199  
    PostFilterGetStateSize\_RTA 1198  
    PostFilterInit\_RTA 1199  
    QMFDDecode\_RTA 1196  
    QMFEncode\_RTA 1197  
    QMFGGetStateSize\_RTA 1195  
    QMFINit\_RTA 1196  
run length encoding (RLE)  
    DecodeRLE 1744  
    EncodeRLE 1743

## S

sample-generating Functions 121  
SampleDown 332  
SampleUp 330  
Sampling functions  
    SampleDown 332  
    SampleUp 330  
SBADPCMDecode\_G722 1183  
SBADPCMDecodeInit\_G722 1182  
SBADPCMDecodeStateSize 1182  
SBADPCMEncode\_G722 1180  
SBADPCMEncodeInit\_G722 1179  
SBADPCMEncodeStateSize\_G722 1179  
Scale 1228  
scale factor bands 1228  
scale factors calculation 1226  
ScaleLM 713  
Schur 650  
Schur\_GSMFR 1102  
Set 119  
Shift functions 173  
ShortTermAnalysisFilter\_GSMFR 1099  
ShortTermPostFilter\_G729 911  
ShortTermSynthesisFilter\_GSMFR 1100  
SignChangeRate 645  
SIMD instructions 51  
Sin 1552  
SinC 769  
SinCos 1555  
single-rate FIR LMS filter functions  
    FIRLMS 426  
    FIRLMSFree 422  
    FIRLMSGetDlyLine 424  
    FIRLMSGetTaps 423  
    FIRLMSInitAlloc 421  
    FIRLMSOne\_Direct 428  
    FIRLMSSetDlyLine 425  
Sinh 1574  
sliding window technique 1685  
SmoothedPowerSpectrumAurora 819  
SNR\_AMRWB 1067  
SortAscend 223  
SortDescend 223  
SortIndexAscend 224  
SortIndexDescend 224  
SortRadixAscend 226  
SortRadixDescend 226  
SortRadixIndexAscend 228

- 
- SortRadixIndexDescend 228
  - special vector functions 154, 155, 157
    - VectorJaehne 154
    - VectorRamp 157
    - VectorSlope 155
  - spectral data prequantization 1217
  - spectral values restoration 1228
  - speech codecs common functions
    - AutoCorr 861
    - AutoCorr\_NormE 863
    - AutoCorrLagMax 862
    - AutoScale 858
    - ConvPartial 852
    - CrossCorr 865
    - CrossCorrLagMax 866
    - DotProdAutoScale 859
    - InterpolateC\_NR 853
    - InvSqrt 860
    - Mul\_NR 854
    - MulC\_NR 856
    - MulPowerC\_NR 857
    - SynthesisFilter\_G723 867
  - speech, synthesized 908
  - SplitC 1466
  - SplitScaled 243
  - SplitVQ 803
  - Spread 1267
  - Sqr 206, 1500
  - Sqrt 208, 1518
  - Statistical functions
    - CountInRange 329
    - DotProd 323
    - Max 303
    - MaxAbs 305
    - MaxAbsIndx 306
    - MaxEvery 326
    - MaxIndx 304
    - Mean 313
    - MeanStdDev 316
    - Min 307
    - MinAbs 309
    - MinAbsIndx 310
    - MinEvery 326
    - MinIndx 308
    - MinMax 311
    - MinMaxIndx 312
    - Norm 318
    - NormDiff 320
    - PowerSpectr, complex 250
  - Statistical functions (*continued*)
    - StdDev 315
    - Sum 301
    - ZeroCrossing 327
  - StdDev 315
  - Streaming SIMD Extensions 51
  - string functions
    - Compare 1451
    - CompareIgnoreCase 1452
    - CompareIgnoreCaseLatin 1452
    - Concat 1463
    - ConcatC 1465
    - Equal 1454
    - Find 1443
    - FindC 1445
    - FindCAny, 1447
    - FindRev 1443
    - FindRevC 1445
    - FindRevCAny 1447
    - Hash 1462
    - Insert 1448
    - Lowercase 1460
    - LowercaseLatin 1460
    - RegExpFind 1473
    - RegExpFree 1470
    - RegExpGetSize 1472
    - RegExpInit 1470
    - RegExpSetMatchLimit 1472
    - Remove 1450
    - ReplaceC 1458
    - SplitC 1466, 1469
    - TrimC 1455
    - TrimCAny 1457
    - TrimEndCAny 1457
    - TrimStartCAny 1457
    - Uppercase 1459
    - UppercaseLatin 1459
  - Sub 194, 1498
  - SubbandController\_EC 1146
  - SubbandControllerDT\_EC 1150
  - SubbandControllerDTGetSize\_EC 1148
  - SubbandControllerDTInit\_EC 1149
  - SubbandControllerDTReset\_EC 1150
  - SubbandControllerDTUpdate\_EC 1152
  - SubbandControllerGetSize\_EC 1143
  - SubbandControllerInit\_EC 1144
  - SubbandControllerReset\_EC 1147
  - SubbandControllerUpdate\_EC 1145
  - SubbandProcessGetSize 1138

SubbandProcessInit 1139  
SubbandSynthesis 1141  
SubC 189  
SubCRev 192  
SubRow 633  
Sum 301  
SumColumn 629  
SumColumnAbs 785  
SumColumnSqr 786  
SumLn 217  
SumMeanVar 690  
SumRow 631  
SumRowAbs 787  
SumRowSqr 787  
SVD 788  
SVDSort 788  
SwapBytes 231  
SyntesisFilter\_G728 1120  
SynthesisDownFilter\_SBR 1424  
SynthesisDownFilterFree\_SBR 1405  
SynthesisDownFilterGetSize\_SBR 1409  
SynthesisDownFilterInit\_SBR 1413  
SynthesisDownFilterInitAlloc\_SBR 1402  
SynthesisFilter\_AMRWB 1039  
SynthesisFilter\_AMRWBE 1071  
SynthesisFilter\_DTS 1439  
SynthesisFilter\_G723 867, 973  
SynthesisFilter\_G729 908  
SynthesisFilter\_PQMF\_MP3 1333  
SynthesisFilter\_SBR 1419  
SynthesisFilterFree\_DTS 1438  
SynthesisFilterFree\_PQMF\_MP3 1332  
SynthesisFilterFree\_SBR 1404  
SynthesisFilterGetSize\_DTS 1437  
SynthesisFilterGetSize\_PQMF\_MP3 1332  
SynthesisFilterGetSize\_SBR 1407  
SynthesisFilterGetStateSize\_G728 1119  
SynthesisFilterInit\_DTS 1436  
SynthesisFilterInit\_G728 1119  
SynthesisFilterInit\_PQMF\_MP3 1330  
SynthesisFilterInit\_SBR 1412  
SynthesisFilterInitAlloc\_DTS 1437  
SynthesisFilterInitAlloc\_PQMF\_MP3 1331  
SynthesisFilterInitAlloc\_SBR 1401  
SynthPQMF\_MP3 1329

## T

TabsCalculation\_Aurora 823  
Tan 1559  
Tanh 1576  
Threshold 258  
Threshold\_GT 262  
Threshold\_GTAbs 266  
Threshold\_GTVAl 268  
Threshold\_LT 262  
Threshold\_LTAbs 266  
Threshold\_LTInv 273  
Threshold\_LTVAl 268  
Threshold\_LTVAlGTVAl 268  
tilt 913  
TiltCompensation\_G723 974  
TiltCompensation\_G729 913  
TiltCompensation\_G7291 949  
ToeplizMatrix\_G723 966  
ToeplizMatrix\_G729 898  
Tone\_Direct 128  
tone-generating functions 122, 123, 124, 126, 128, 129  
    Tone\_Direct 128  
    ToneFree 123  
    ToneGetStateSizeQ15 124  
    ToneInitAllocQ15 122  
    ToneInitQ15 124  
    ToneQ15 126  
    ToneQ15\_Direct 129  
ToneDetect\_EC 1155  
ToneDetectGetStateSize\_EC 1154  
ToneDetectInit\_EC 1154  
ToneFree 123  
ToneGetStateSizeQ15 124  
ToneInitAllocQ15 122  
ToneInitQ15 124  
ToneQ15 126  
ToneQ15\_Direct 129  
transcendental mathematical functions 1493  
Triangle\_Direct 137  
triangle-generating functions 130, 132, 133, 134, 135, 136, 137, 140  
    Triangle\_Direct 137  
    TriangleFree 133  
    TriangleGetStateSizeQ15 134  
    TriangleInitAllocQ15 132  
    TriangleInitQ15 135  
    TriangleQ15 136

triangle-generating functions (*continued*)

- TriangleQ15\_Direct 140
- TriangleFree 133
- TriangleGetStateSizeQ15 134
- TriangleInitAllocQ15 132
- TriangleInitQ15 135
- TriangleQ15 136
- TriangleQ15\_Direct 140
- TrimC 1455
- TrimCAny 1457
- TrimEndCAny 1457
- TrimStartCAny 1457
- Trunc 1605

**U**

## uniform distribution functions 141, 142, 143, 144, 147

- RandUniform 144
- RandUniformFree 142
- RandUniformGetSize 144
- RandUniformInit 143
- RandUniformInitAlloc 141
- RandUniform\_Direct 147

UnitCurve 661

## Unpack of packed data

- ConjCcs 500
- ConjPack 496
- ConjPerm 498

UnpackADIFHeader\_AAC 1348

UnpackADTSFrameHeader\_AAC 1349

UnpackFrameHeader\_MP3 1318

UnpackHuffContext\_BZ2 1763

UnpackScaleFactors\_MP3 1321

UnpackSideInfo\_MP3 1319

UpdateGConst 775

UpdateLinear 345

UpdateMean 772

UpdateNoisePSDMCRA 840

UpdatePathMetricsDV 287

UpdatePower 347

UpdateVar 773

UpdateWeight 774

Uppercase 1459

UppercaseLatin 1459

Upsample\_AMRWB 1080

**V**

VAD\_AMRWB 1041

VAD1\_GSMAMR 1013

VAD2\_GSMAMR 1015

VADDecision\_Aurora 832

VADFlush\_Aurora 833

VADGetBufSize\_Aurora 831

VADGetEnergyLevel\_AMRWB 1042

VADGetSize\_AMRWB 1040

VADInit\_AMRWB 1040

VADInit\_Aurora 831

VarColumn 753

Variable Length coding 1621, 1633

variable length coding table 1621, 1633

VecMatMul 638

## Vector correlation functions

- AutoCorr 339

## vector initialization functions 113, 116, 117, 119, 120

- Copy 113
- Move 117
- PackBits 116
- Set 119
- Zero 120

## Vector quantization functions

- CdbkFree 798
- CdbkGetSize 794
- CdbkInit 795
- CdbkInitAlloc 796
- FormVector 791
- FormVectorVQ 804
- GetCdbkSize 799
- GetCodebook 799
- SplitVQ 803
- VQ 800
- VQSingle\_Sort 801
- VQSingle\_Thresh 801

vector quantizer 884

VectorJaehne 154

VectorRamp 157

VectorSlope 155

## Viterbi decoder functions

- BuildSymbITableDV4D 286
- CalcStatesDV 285
- GetVarPointDV 284
- UpdatePathMetricsDV 287

VLC functions 1255

VLCCountBits 1627

VLCCountEscBits\_AAC 1263

VLCCountEscBits\_MP3 1262  
 VLCDecodeBlock 1631  
 VLCDecodeEscBlock\_AAC 1258  
 VLCDecodeEscBlock\_MP3 1255  
 VLCDecodeFree 1628  
 VLCDecodeGetSize 1630  
 VLCDecodeInit 1629  
 VLCDecodeInitAlloc 1627  
 VLCDecodeOne 1632  
 VLCDecodeUTupleBlock 1638  
 VLCDecodeUTupleEscBlock\_AAC 1261  
 VLCDecodeUTupleEscBlock\_MP3 1260  
 VLCDecodeUTupleFree 1635  
 VLCDecodeUTupleGetSize 1637  
 VLCDecodeUTupleInit 1636  
 VLCDecodeUTupleInitAlloc 1634  
 VLCDecodeUTupleOne 1639  
 VLCEncodeBlock 1625  
 VLCEncodeEscBlock\_AAC 1266  
 VLCEncodeEscBlock\_MP3 1264  
 VLCEncodeFree 1622  
 VLCEncodeGetSize 1624  
 VLCEncodeInit 1623  
 VLCEncodeInitAlloc 1622  
 VLCEncodeOne 1626  
 Voice Activity Detection functions  
     FindPeaks 842  
     Periodicity 845  
     PeriodicityLSPE 843  
 VQCodeBookFree 1271  
 VQCodeBookGetSize 1271  
 VQCodeBookInit 1270  
 VQCodeBookInitAlloc 1269  
 VQIndexSelect 1276  
 VQMainSelect 1274  
 VQReconstruction 1279  
 VQSingle\_Sort 801  
 VQSingle\_Thresh 801

## W

Walsh-Hadamard transform  
     WHT 564  
 Wavelet transform functions  
     WTFwd 603  
     WTFwdFree 601  
     WTFwdGetDlyLine 606  
     WTFwdInitAlloc 599

## Wavelet transform functions *(continued)*

    WTFwdSetDlyLine 606  
     WTHaarFwd 594  
     WTHaarInv 594  
     WTInv 608  
     WTInvFree 601  
     WTInvGetDlyLine 611  
     WTInvInitAlloc 599  
     WTInvSetDlyLine 611  
 WaveProcessing\_Aurora 824  
 WeightedMeanColumn 755  
 WeightedMeanVarColumn 757  
 WeightedSum 790  
 WeightedVarColumn 756  
 weighting matrix 959  
 WeightingFilter\_GSMFR 1102  
 WHT 564  
 WHT functions  
     WHTGetBufferSize 566  
 WHTGetBufferSize 566  
 WienerFilterDesign\_Aurora 821  
 WinBartlett 289  
 WinBlackman 291  
 Windowing functions  
     WinBartlett 289  
     WinBlackman 291  
     WinHamming 295  
     WinHann 297  
     WinKaiser 299  
 WinHamming 295  
 WinHann 297  
 WinHybrid\_G728 1130  
 WinHybrid\_G729E 923  
 WinHybridGetStateSize\_G728 1129  
 WinHybridGetStateSize\_G729E 922  
 WinHybridInit\_G728 1129  
 WinHybridInit\_G729E 923  
 WinKaiser 299  
 WTFwd 603  
 WTFwdFree 601  
 WTFwdGetDlyLine 606  
 WTFwdInitAlloc 599  
 WTFwdSetDlyLine 606  
 WTHaarFwd 594  
 WTHaarInv 594  
 WTInv 608  
 WTInvFree 601  
 WTInvGetDlyLine 611  
 WTInvInitAlloc 599



WTInvSetDlyLine 611

## X

Xor 171

XorC 170

## Z

Zero 120

ZeroCrossing 327

ZeroMean 642

ZLIB 1666

ZLIB functions

Adler32 1694

CRC32 1695

DecodeLZ77 1685

DecodeLZ77DynamicHuff 1689

DecodeLZ77FixedHuff 1688

DecodeLZ77GetBlockType 1686

DecodeLZ77GetPairs 1691

DecodeLZ77GetSize 1684

DecodeLZ77GetStatus 1692

DecodeLZ77Init 1683

ZLIB functions (*continued*)

DecodeLZ77InitAlloc 1685

DecodeLZ77Reset 1694

DecodeLZ77SetPairs 1692

DecodeLZ77SetStatus 1693

DecodeLZ77StoredBlock 1690

DeflateDictionarySet 1700

DeflateLZ77 1698

EncodeLZ77 1674

EncodeLZ77DynamicHuff 1677

EncodeLZ77FixedHuf 1676

EncodeLZ77Flush 1679

EncodeLZ77Free 1673

EncodeLZ77GetPairs 1680

EncodeLZ77GetSize 1672

EncodeLZ77GetStatus 1681

EncodeLZ77Init 1671

EncodeLZ77InitAlloc 1672

EncodeLZ77Reset 1683

EncodeLZ77SelectHuffMode 1675

EncodeLZ77SetPairs 1681

EncodeLZ77SetStatus 1682

EncodeLZ77StoredBlock 1678

Inflate 1703

InflateBuildHuffTable 1702

