



Intel® Integrated Performance Primitives

Reference Manual, Volume 4: Cryptography

IPP 7.0

Document Number: 303881-20US

Legal Information

Contents

Legal Information.....19

Chapter 1: Volume Overview

What's New.....	21
Notational Conventions.....	21
Basic Features.....	22

Chapter 2: Symmetric Cryptography Primitive Functions

Block Cipher Modes of Operation.....	25
DES/TDES Functions.....	26
DESGetSize	28
DESInit.....	28
DESPack, DESUnpack.....	29
DESEncryptECB.....	29
DESDecryptECB.....	30
DESEncryptCBC.....	31
DESDecryptCBC.....	31
DESEncryptCFB.....	32
DESDecryptCFB.....	33
DESEncryptOFB.....	34
DESDecryptOFB.....	34
DESEncryptCTR.....	35
DESDecryptCTR.....	36
TDESEncryptECB.....	37
TDESDecryptECB.....	37
TDESEncryptCBC.....	38
TDESDecryptCBC.....	39
TDESEncryptCFB.....	40
TDESDecryptCFB.....	41
TDESEncryptOFB.....	42
TDESDecryptOFB.....	43
TDESEncryptCTR.....	43
TDESDecryptCTR.....	44

Example of Using DES/TDES Functions.....	46
Rijndael Functions.....	47
Rijndael128GetSize.....	51
Rijndael128Init, SafeRijndael128Init.....	51
Rijndael128Pack, Rijndael128Unpack.....	52
Rijndael128EncryptECB.....	53
Rijndael128DecryptECB.....	53
Rijndael128EncryptCBC.....	54
Rijndael128DecryptCBC.....	55
Rijndael128EncryptCFB.....	56
Rijndael128DecryptCFB.....	56
Rijndael128EncryptOFB.....	57
Rijndael128DecryptOFB.....	58
Rijndael128EncryptCTR.....	59
Rijndael128DecryptCTR.....	60
Rijndael128EncryptCCM.....	60
Rijndael128EncryptCCM_u8.....	61
Rijndael128DecryptCCM.....	62
Rijndael128DecryptCCM_u8.....	63
Rijndael192GetSize.....	64
Rijndael192Init.....	65
Rijndael192Pack, Rijndael192Unpack.....	65
Rijndael192EncryptECB.....	66
Rijndael192DecryptECB.....	67
Rijndael192EncryptCBC.....	67
Rijndael192DecryptCBC.....	68
Rijndael192EncryptCFB.....	69
Rijndael192DecryptCFB.....	70
Rijndael192EncryptOFB.....	71
Rijndael192DecryptOFB.....	71
Rijndael192EncryptCTR.....	72
Rijndael192DecryptCTR	73
Rijndael256GetSize.....	74
Rijndael256Init.....	74
Rijndael256Pack, Rijndael256Unpack.....	75
Rijndael256EncryptECB.....	75
Rijndael256DecryptECB.....	76
Rijndael256EncryptCBC.....	77
Rijndael256DecryptCBC.....	78
Rijndael256EncryptCFB.....	78
Rijndael256DecryptCFB.....	79

Rijndael256EncryptOFB.....	80
Rijndael256DecryptOFB.....	81
Rijndael256EncryptCTR.....	82
Rijndael256DecryptCTR.....	82
Example of Using Rijndael Functions.....	84
AES-CCM Functions.....	85
Rijndael128CCMEncryptMessage.....	87
Rijndael128CCMDecryptMessage.....	88
Rijndael128CCMGetSize.....	89
Rijndael128CCMInit.....	89
Rijndael128CCMStart.....	90
Rijndael128CCMEncrypt.....	90
Rijndael128CCMDecrypt.....	91
Rijndael128CCMGetTag.....	92
Rijndael128CCMMessageLen.....	92
Rijndael128CCMTagLen.....	93
AES-GCM Functions.....	93
Rijndael128GCMEncryptMessage.....	95
Rijndael128GCMDecryptMessage.....	96
Rijndael128GCMGetSize.....	97
Rijndael128GCMGetSizeManaged.....	97
Rijndael128GCMInit.....	98
Rijndael128GCMInitManaged.....	99
Rijndael128GCMStart.....	101
Rijndael128GCMReset.....	101
Rijndael128GCMProcessIV.....	102
Rijndael128GCMProcessAAD.....	103
Rijndael128GCMEncrypt.....	103
Rijndael128GCMDecrypt.....	104
Rijndael128GCMGetTag.....	104
Blowfish Functions.....	105
BlowfishGetSize.....	106
BlowfishInit.....	106
BlowfishPack, BlowfishUnpack.....	107
BlowfishEncryptECB.....	108
BlowfishDecryptECB.....	108
BlowfishEncryptCBC.....	109
BlowfishDecryptCBC.....	110
BlowfishEncryptCFB.....	111
BlowfishDecryptCFB.....	111
BlowfishEncryptOFB.....	112

BlowfishDecryptOFB.....	113
BlowfishEncryptCTR.....	114
BlowfishDecryptCTR.....	115
Example of Using Blowfish Functions.....	116
Twofish Functions.....	117
TwofishGetSize.....	118
TwofishInit.....	119
TwofishPack, TwofishUnpack.....	119
TwofishEncryptECB.....	120
TwofishDecryptECB.....	120
TwofishEncryptCBC.....	121
TwofishDecryptCBC.....	122
TwofishEncryptCFB.....	123
TwofishDecryptCFB.....	123
TwofishEncryptOFB.....	124
TwofishDecryptOFB.....	125
TwofishEncryptCTR.....	126
TwofishDecryptCTR.....	127
Example of Using Twofish Functions.....	128
RC5* Functions.....	129
RC5* Algorithm Functions for 64-bit Block Size.....	131
ARCFive64GetSize.....	131
ARCFive64Init.....	132
ARCFive64Pack, ARCFive64Unpack.....	132
ARCFive64EncryptECB.....	133
ARCFive64DecryptECB.....	134
ARCFive64EncryptCBC.....	134
ARCFive64DecryptCBC.....	135
ARCFive64EncryptCFB.....	136
ARCFive64DecryptCFB.....	137
ARCFive64EncryptOFB.....	137
ARCFive64DecryptOFB.....	138
ARCFive64EncryptCTR.....	139
ARCFive64DecryptCTR.....	140
RC5* Algorithm Functions for 128-bit Block Size.....	141
ARCFive128GetSize.....	141
ARCFive128Init.....	141
ARCFive128Pack, ARCFive128Unpack.....	142
ARCFive128EncryptECB.....	142
ARCFive128DecryptECB.....	143
ARCFive128EncryptCBC.....	144

ARCFive128DecryptCBC.....	145
ARCFive128EncryptCFB.....	145
ARCFive128DecryptCFB.....	146
ARCFive128EncryptOFB.....	147
ARCFive128DecryptOFB.....	148
ARCFive128EncryptCTR.....	148
ARCFive128DecryptCTR.....	149
ARCfour Functions.....	150
ARCfourGetSize.....	151
ARCfourCheckKey.....	151
ARCfourInit.....	152
ARCfourPack, ARCfourUnpack.....	152
ARCfourEncrypt.....	153
ARCfourDecrypt.....	153
ARCfourReset.....	154

Chapter 3: One-Way Hash Primitives

Hash Functions.....	157
MD5GetSize.....	158
MD5Init.....	158
MD5Pack, MD5Unpack.....	159
MD5Duplicate.....	159
MD5Update.....	160
MD5Final.....	160
MD5GetTag.....	161
SHA1GetSize.....	161
SHA1Init.....	162
SHA1Pack, SHA1Unpack.....	162
SHA1Duplicate.....	163
SHA1Update.....	163
SHA1Final.....	164
SHA1GetTag.....	165
SHA224GetSize.....	165
SHA224Init.....	166
SHA224Pack, SHA224Unpack.....	166
SHA224Duplicate.....	167
SHA224Update.....	167
SHA224Final.....	168
SHA224GetTag.....	168
SHA256GetSize.....	169
SHA256Init.....	169

SHA256Pack, SHA256Unpack.....	170
SHA256Duplicate.....	170
SHA256Update.....	171
SHA256Final.....	172
SHA256GetTag.....	172
SHA384GetSize.....	173
SHA384Init.....	173
SHA384Pack, SHA384Unpack.....	174
SHA384Duplicate.....	174
SHA384Update.....	175
SHA384Final.....	175
SHA384GetTag.....	176
SHA512GetSize.....	176
SHA512Init.....	177
SHA512Pack, SHA512Unpack.....	177
SHA512Duplicate.....	178
SHA512Update.....	178
SHA512Final.....	179
SHA512GetTag.....	180
Generalized Hash Functions for Non-Streaming Messages.....	180
General Definition of a Hash Function.....	180
MD5MessageDigest.....	181
SHA1MessageDigest.....	182
SHA224MessageDigest.....	184
SHA256MessageDigest.....	184
SHA384MessageDigest.....	185
SHA512MessageDigest.....	185
Mask Generation Functions.....	186
User's Implementation of a Mask Generation Function.....	186
MGF_MD5.....	187
MGF_SHA1.....	187
MGF_SHA224.....	188
MGF_SHA256.....	188
MGF_SHA384.....	189
MGF_SHA512.....	190

Chapter 4: Data Authentication Primitive Functions

Message Authentication Functions.....	191
Keyed Hash Functions.....	191
HMACSHA1GetSize	195
HMACSHA1Init.....	195

HMACSHA1Pack, HMACSHA1Unpack.....	196
HMACSHA1Duplicate	196
HMACSHA1Update.....	197
HMACSHA1Final.....	197
HMACSHA1GetTag.....	198
HMACSHA1MessageDigest.....	199
HMACSHA224GetSize	199
HMACSHA224Init	200
HMACSHA224Pack, HMACSHA224Unpack.....	200
HMACSHA224Duplicate	201
HMACSHA224Update	201
HMACSHA224Final	202
HMACSHA224GetTag.....	203
HMACSHA224MessageDigest	203
HMACSHA256GetSize	204
HMACSHA256Init.....	204
HMACSHA256Pack, HMACSHA256Unpack.....	205
HMACSHA256Duplicate	205
HMACSHA256Update.....	206
HMACSHA256Final.....	207
HMACSHA256GetTag.....	207
HMACSHA256MessageDigest.....	208
HMACSHA384GetSize	209
HMACSHA384Init.....	210
HMACSHA384Pack, HMACSHA384Unpack.....	210
HMACSHA384Duplicate	211
HMACSHA384Update.....	211
HMACSHA384Final.....	212
HMACSHA384GetTag.....	213
HMACSHA384MessageDigest.....	213
HMACSHA512GetSize	214
HMACSHA512Init.....	214
HMACSHA512Pack, HMACSHA512Unpack.....	215
HMACSHA512Duplicate	215
HMACSHA512Update.....	216
HMACSHA512Final.....	217
HMACSHA512GetTag.....	217
HMACSHA512MessageDigest.....	218
HMACMD5GetSize	218
HMACMD5Init.....	219
HMACMD5Pack, HMACMD5Unpack.....	219

HMACMD5Duplicate	220
HMACMD5Update.....	221
HMACMD5Final.....	221
HMACMD5GetTag.....	222
HMACMD5MessageDigest.....	223
CMAC Functions.....	224
CMACRijndael128GetSize.....	224
CMACRijndael128Init, CMACSafeRijndael128Init.....	225
CMACRijndael128Update.....	225
CMACRijndael128Final.....	226
CMACRijndael128MessageDigest.....	227
AES-XCBC Functions	227
XCBCRijndael128GetSize.....	228
XCBCRijndael128Init.....	228
XCBCRijndael128Update.....	229
XCBCRijndael128GetTag.....	230
XCBCRijndael128Final.....	230
XCBCRijndael128MessageTag.....	231
Data Authentication Functions.....	231
DAADESGetSize	234
DAADESInit	234
DAADESUpdate.....	235
DAADESFinal.....	235
DAADESMessageDigest.....	236
DAATDESGetSize	237
DAATDESInit.....	237
DAATDESUpdate.....	238
DAATDESFinal.....	238
DAATDESMessageDigest.....	239
DAARijndael128GetSize	240
DAARijndael128Init, DAASafeRijndael128Init.....	240
DAARijndael128Update.....	241
DAARijndael128Final.....	241
DAARijndael128MessageDigest.....	242
DAARijndael192GetSize	243
DAARijndael192Init.....	243
DAARijndael192Update.....	244
DAARijndael192Final.....	244
DAARijndael192MessageDigest.....	245
DAARijndael256GetSize	246
DAARijndael256Init.....	246

DAARijndael256Update.....	247
DAARijndael256Final.....	247
DAARijndael256MessageDigest.....	248
DAABlowfishGetSize	249
DAABlowfishInit.....	249
DAABlowfishUpdate.....	250
DAABlowfishFinal.....	250
DAABlowfishMessageDigest.....	251
DAATwofishGetSize	252
DAATwofishInit.....	252
DAATwofishUpdate.....	253
DAATwofishFinal.....	253
DAATwofishMessageDigest.....	254

Chapter 5: Public Key Cryptography Functions

Big Number Arithmetic.....	255
Add_BNU.....	256
Sub_BNU.....	257
MulOne_BNU.....	258
MACOne_BNU_I.....	259
Mul_BNU4.....	259
Mul_BNU8.....	260
Div_64u32u.....	260
Sqr_32u64u.....	261
Sqr_BNU4.....	262
Sqr_BNU8.....	262
SetOctString_BNU	263
GetOctString_BNU	263
BigNumGetSize.....	264
BigNumInit.....	264
Set_BN.....	265
SetOctString_BN.....	266
GetSize_BN	267
Get_BN.....	268
ExtGet_BN.....	268
Ref_BN.....	269
GetOctString_BN.....	270
Cmp_BN	271
CmpZero_BN.....	272
Add_BN.....	272
Sub_BN.....	274

Mul_BN.....	274
MAC_BN_I.....	275
Div_BN.....	276
Mod_BN.....	277
Gcd_BN.....	277
ModInv_BN.....	278
Montgomery Reduction Scheme Functions.....	279
MontGetSize.....	281
MontInit.....	281
MontSet.....	282
MontGet.....	282
MontForm.....	283
MontMul.....	284
Example of Using Montgomery Reduction Scheme Functions.....	285
MontExp.....	286
Pseudorandom Number Generation Functions.....	286
User's Implementation of a Pseudorandom Number Generator.....	287
PRNGGetSize	288
PRNGInit.....	288
PRNGSetSeed.....	289
PRNGSetAugment	290
PRNGSetModulus	290
PRNGSetH0	291
PRNGen.....	291
PRNGen_BN	292
Example of Using Pseudorandom Number Generation Functions.....	293
Prime Number Generation Functions.....	294
PrimeGetSize	295
PrimeInit.....	296
PrimeGen.....	296
PrimeTest.....	297
PrimeSet.....	298
PrimeSet_BN	298
PrimeGet.....	299
PrimeGet_BN	299
Example of Using Prime Number Generation Functions.....	301
RSA Algorithm Functions.....	302
Functions for Building RSA System.....	303
RSAGetSize.....	303
RSAInit.....	304
RSAPack, RSAUnpack.....	305

RSASetKey.....	305
RSAGetKey.....	307
RSAGenerate.....	308
RSAValidate.....	309
RSA Primitives.....	311
RSAEncrypt.....	312
RSADecrypt.....	312
Example of Using RSA Primitive Functions.....	313
RSA Encryption Schemes.....	316
RSA-OAEP Scheme Functions.....	316
PKCS V1.5 Encryption Scheme Functions.....	326
RSA Signature Schemes.....	329
RSASSA-PSS Scheme Functions.....	329
PKCS V1.5 Signature Scheme Functions.....	340
Discrete-Logarithm-Based Cryptography Functions.....	351
DLPGetSize	352
DLPInit	352
DLPPack, DLPUnpack.....	353
DLPSet	353
DLPGet	354
DLPSetDP	355
DLPGetDP	356
DLPGenKeyPair	357
DLPPublicKey	357
DLPValidateKeyPair	358
DLPSetKeyPair	359
DLPGenerateDSA	359
DLPValidateDSA	360
DLPSignDSA	361
DLPVerifyDSA.....	362
Example of Using RSA Primitive Functions.....	364
DLPGenerateDH	366
DLPValidateDH	367
DLPSharedSecretDH	368
Elliptic Curve Cryptography Functions.....	369
Functions Based on GF(p).....	371
ECCPGetSize.....	372
ECCPInit	372
ECCPSet	373
ECCPSetStd	374
ECCPGet	375

ECCPGetOrderBitSize	376
ECCPValidate	376
ECCPPointGetSize	378
ECCPPointInit	378
ECCPSetPoint	379
ECCPSetPointAtInfinity	379
ECCPGetPoint	380
ECCPCheckPoint	381
ECCPComparePoint	381
ECCPNegativePoint	382
ECCPAddPoint	383
ECCPMulPointScalar	383
ECCPGenKeyPair	384
ECCPPublicKey	385
ECCPValidateKeyPair	386
ECCPSetKeyPair	387
ECCPSharedSecretDH	388
ECCPSharedSecretDHC.....	389
ECCPSignDSA	390
ECCPVerifyDSA	391
ECCPSignNR.....	392
ECCPVerifyNR	393
Signing/Verification Using the Elliptic Curve Cryptography Functions over a Prime Finite Field.....	395
Functions Based on GF(2 ^m).....	398
ECCBGetSize.....	398
ECCBInit.....	399
ECCBSet.....	400
ECCBSetStd	401
ECCBGet	402
ECCBGetOrderBitSize	403
ECCBValidate	403
ECCBPointGetSize	405
ECCBPointInit	405
ECCBSetPoint	406
ECCBSetPointAtInfinity	407
ECCBGetPoint	407
ECCBCheckPoint	408
ECCBComparePoint	409
ECCBNegativePoint	409
ECCBAddPoint	410

ECCBMulPointScalar	411
ECCBGenKeyPair.....	411
ECCBPublicKey	412
ECCBValidateKeyPair	413
ECCBSetKeyPair	414
ECCBSharedSecretDH	415
ECCBSharedSecretDHC	416
ECCBSignDSA	417
ECCBVerifyDSA	418
ECCBSignNR	419
ECCBVerifyNR	420
Finite Field Arithmetic.....	421
Functions for the GF(p) Field.....	424
GFPGetSize.....	424
GFPInit.....	425
GFPGet.....	425
GFPElementGetSize.....	426
GFPElementInit.....	426
GFPSetElement.....	427
GFPSetElementZero.....	428
GFPSetElementPower2.....	428
GFPSetElementRandom.....	429
GFPCmpElement.....	429
GFPCpyElement.....	430
GFPGetElement.....	430
GFPNeg.....	431
GFPInv.....	432
GFPSqrt.....	432
GFPAAdd.....	433
GFPSub.....	433
GFPMul.....	434
GFPExp.....	435
GFPMontEncode.....	435
GFPMontDecode.....	436
Functions for the GF(p^d) Field.....	437
GFPXGetSize.....	437
GFPXInit.....	437
GFPXGet.....	438
GFPXElementGetSize.....	439
GFPXElementInit.....	439
GFPXSetElement.....	440

GFPXSetElementZero.....	441
GFPXSetElementPowerX.....	441
GFPXSetElementRandom.....	442
GFPXCmpElement.....	442
GFPXCpyElement.....	443
GFPXGetElement.....	444
GFPXNeg.....	444
GFPXInv.....	445
GFPXAdd.....	445
GFPXAdd_GFP.....	446
GFPXSub.....	447
GFPXSub_GFP.....	447
GFPXMul.....	448
GFPXMul_GFP.....	449
GFPXExp.....	450
GFPXDiv.....	450
Functions for the GF(p^d) Field.....	451
GFPXQGetSize.....	451
GFPXQInit.....	452
GFPXQGet.....	453
GFPXQElementGetSize.....	453
GFPXQElementInit.....	454
GFPXQSetElement.....	454
GFPXQSetElementZero.....	455
GFPXQSetElementPowerX.....	456
GFPXQSetElementRandom.....	456
GFPXQCmpElement.....	457
GFPXQCpyElement.....	457
GFPXQGetElement.....	458
GFPXQNeg.....	459
GFPXQInv.....	459
GFPXQAdd.....	460
GFPXQSub.....	461
GFPXQMul.....	461
GFPXQMul_GFP.....	462
GFPXQExp.....	463
Arithmetic of the Group of Elliptic Curve Points.....	463
Functions for the Elliptic Curve over GF(p).....	465
GFPECGetSize.....	465
GFPECInit.....	466
GFPECSet.....	467

GFPECCGet.....	467
GFPECCVerify.....	468
GFPECCPointGetSize.....	469
GFPECCPointInit.....	469
GFPECCSetPoint.....	470
GFPECCSetPointAtInfinity.....	471
GFPECCSetPointRandom.....	471
GFPECCopyPoint.....	472
GFPECCGetPoint.....	472
GFPECCVerifyPoint.....	473
GFPECCmpPoint.....	474
GFPECCNegPoint.....	474
GFPECCAddPoint.....	475
GFPECCMulPointScalar.....	476
Functions for the Elliptic Curve over $GF(p^d)$	476
GFPXECGetSize.....	476
GFPXECInit.....	477
GFPXECSet.....	478
GFPXECGet.....	479
GFPXECVerify.....	479
GFPXECPointGetSize.....	480
GFPXECPointInit.....	481
GFPXECSetPoint.....	481
GFPXECSetPointAtInfinity.....	482
GFPXECSetPointRandom.....	482
GFPXECCopyPoint.....	483
GFPXECGetPoint.....	484
GFPXECVerifyPoint.....	484
GFPXECCmpPoint.....	485
GFPXECNegPoint.....	486
GFPXECAddPoint.....	486
GFPXECMulPointScalar.....	487
Tate Pairing.....	487
TatePairingDE3GetSize.....	488
TatePairingDE3Init.....	488
TatePairingDE3Get.....	489
TatePairingDE3Apply.....	490
 Appendix A: Support Functions and Classes	
Version Information Function.....	491
GetLibVersion.....	491

Classes and Functions Used in Examples.....	492
BigNumber Class.....	492
Functions for Creation of Cryptographic Contexts.....	501

Appendix B: Calling the Cryptography Functions from Fortran-90

Bibliography

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 go to

<http://www.intel.com/design/literature.htm>

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 for details.

MPEG-1, MPEG-2, MPEG-4, H.261, H.263, H.264, MP3, DV, VC-1, MJPEG, AC3, and AAC are international standards promoted by ISO, IEC, ITU, ETSI and other organizations. Implementations of these standards, or the standard enabled platforms may require licenses from various entities, including Intel Corporation.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to <http://www.intel.com/performance>.

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.

Microsoft, Windows, Visual Studio, Visual C++, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Copyright© 2001-2012, Intel Corporation. All rights reserved.

Volume Overview

This manual describes the structure, operation, and functions of Intel® Integrated Performance Primitives (Intel® IPP) for cryptography. The manual provides a background for cryptography concepts used in the Intel IPP software as well as detailed description of the respective Intel IPP functions. The Intel IPP functions are combined in groups by their functionality. Each group of functions is described in a separate chapter.

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

What's New

This Reference Manual documents Intel® Integrated Performance Primitives 7.0 Update 7.

The document contains bugfixes of the documentation for [RSA Signature Schemes](#).

Notational Conventions

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.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

In this manual, notational conventions include:

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

Font Conventions

The following font conventions are used throughout this manual:

This type style

Mixed with the uppercase in function names, code examples, and call statements, for example, `ippsAdd_BNU`.

This type style

Parameters in function prototype parameters and parameters description, for example, `pCtx`, `pSrcMsg`.

Naming Conventions

The naming conventions for different items are the same as used by the Intel IPP software.

- All names of the functions used for cryptographic operations 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, each function is introduced by its short name (without the `ipps` prefix and descriptors) and a brief description of its purpose.

The `ipps` prefix in function names is always used in code examples and function prototypes. In the text, this prefix is omitted when referring to the function group.

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

Basic Features

Like other members of Intel® Performance Libraries, Intel Integrated Performance Primitives 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 conventions and similar composition of prototypes for primitives that refer to different application domains.
- Intel IPP functions use an abstraction level which is best suited to achieve superior performance figures by the application programs.

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

Intel IPP is well-suited for cross-platform applications. For example, functions developed for the IA-32 architecture can be readily ported to the Intel® 64 architecture-based platform. In addition, each Intel IPP function has its reference code written in ANSI C, which clearly presents the algorithm used and provides for compatibility with different operating systems.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Symmetric Cryptography Primitive Functions

2

In the context of secure data communication, symmetric cryptography primitive functions protect messages transferred over open communication media by offering adequate security strength to meet application security requirement, as well as algorithmic efficiency to enable secure communication in real time.

Intel® Integrated Performance Primitives (Intel® IPP) for cryptography offer operations using the following symmetric cryptography algorithms:

- Block ciphers: DES, Triple DES (TDES) [[FIPS PUB 46-3](#)], Rijndael [[AES](#)], including AES-CCM [[NIST SP 800-38C](#)] and AES-GCM [[NIST SP 800-38D](#)], Twofish [[TF](#)], Blowfish [[BF](#)], and RC5* [[RC5](#)]
- Stream ciphers: ARCFour [[AC](#)], producing the same encryption/decryption as the RC4* proprietary cipher of RSA Security Inc.

Block Cipher Modes of Operation

Most of Symmetric Cryptography Algorithms implemented in Intel IPP are Block Ciphers, which operate on data blocks of the fixed size. Block Ciphers encrypt a plaintext block into a ciphertext block or decrypts a ciphertext block into a plaintext block. The size of the data blocks depends on the specific algorithm. [Table “Block Sizes in Symmetric Algorithms”](#) shows the correspondence between Block Ciphers applied and their data block size.

Block Sizes in Symmetric Algorithms

Block Cipher Name	Data Block Size (bits)
DES	64
TDES	64
Rijndael128	128
Rijndael192	192
Rijndael256	256
Twofish	128
Blowfish	64
RC5	64 or 128

Block Cipher modes of executing the operation of encryption/decryption are applied in practice more frequently than “pure” Block Ciphers. On one hand, the modes enable you to process arbitrary length data stream. On the other hand, they provide additional security strength.

Intel IPP for cryptography supports five widely used modes, as specified in [[NIST SP 800-38A](#)]:

- Electronic Code Book (ECB) mode
- Cipher Block Chain (CBC) mode
- Cipher Feedback (CFB) mode
- Output Feedback (OFB) mode
- Counter (CTR) mode.



NOTE. For simplicity and consistency, the mathematical expression and pseudocode in this chapter describes the behaviour of each function.

The cryptographic functions described in this chapter require the application to specify both the plaintext message and the ciphertext message lengths as multiples of block size of the respective algorithm (see [Table “Block Sizes in Symmetric Algorithms”](#)). To meet this requirement in ciphering the message, the application may use any padding scheme, for example, the scheme defined in [PKCS7]. In case padding is used, the application is responsible for correct interpretation and processing of the last deciphered message block. So of the three padding schemes available for earlier releases,

```
typedef enum {
    NONE = 0, IppsCPPaddingNONE = 0,
    PKCS7 = 1, IppsCPPaddingPKCS7 = 1,
    ZEROS = 2, IppsCPPaddingZEROS = 2
} IppsCPPadding;
```

only IppsCPPaddingNONE remains acceptable.

DES/TDES Functions

Data Encryption Standard (DES) is a well-known symmetric cipher and also the first modern commercial-grade algorithm with open and fully specified implementation details. DES consists of a Feistel network iterated 16 times with the block size of 64 bits and the effective key size of 56 bits.

Triple Data Encryption Standard (TDES) is a revised symmetric algorithm scheme built on the DES system. TDES encryption process includes three consecutive DES operations in the encryption, decryption, and encryption (E-D-E) sequence again in accordance with the American standard FIPS 46-3.

Although the functions that support TDES operations require three sets of round keys, the functions can operate under TDES cipher system with a two-set round keys by simply setting the third set of round keys to be the same as the first set.

You can use the functions described in this section for performing various operational modes under the DES/TDES cipher systems.

[Table “Intel IPP DES/TDES Functions”](#) lists Intel IPP DES/TDES functions:

Intel IPP DES/TDES Functions

Function Base Name	Operation
DES Functions	
DESGetSize	Gets the size of the IppsDESSpec context.
DESInit	Initializes user-supplied memory as IppsDESSpec context for future use.
DESPack , DESUnpack	Packs/unpacks the IppsDESSpec context into/from a user-defined buffer.
DESEncryptECB	Encrypts a variable length data stream in the ECB mode.
DESDecryptECB	Decrypts a variable length data stream in the ECB mode.
DESEncryptCBC	Encrypts a variable length data stream in the CBC mode.
DESDecryptCBC	Decrypts a variable length data stream in the CBC mode.
DESEncryptCFB	Encrypts a variable length data stream in the CFB mode.
DESDecryptCFB	Decrypts a variable length data stream in the CFB mode.
DESEncryptOFB	Encrypts a variable length data stream in the OFB mode.
DESDecryptOFB	Decrypts a variable length data stream in the OFB mode.
DESEncryptCTR	Encrypts a variable length data stream in the CTR mode.

Function Base Name	Operation
DESDecryptCTR	Decrypts a variable length data stream in the CTR mode.
TDES Functions	
TDESEncryptECB	Encrypts variable length data stream in the ECB mode.
TDESDecryptECB	Decrypts variable length data stream in the ECB mode.
TDESEncryptCBC	Encrypts variable length data stream in the CBC mode.
TDESDecryptCBC	Decrypts variable length data stream in the CBC mode.
TDESEncryptCFB	Encrypts variable length data stream in the CFB mode.
TDESDecryptCFB	Decrypts variable length data stream in the CFB mode.
TDESEncryptOFB	Encrypts variable length data stream in the OFB mode.
TDESDecryptOFB	Decrypts variable length data stream in the OFB mode.
TDESEncryptCTR	Encrypts a variable length data stream in the CTR mode.
TDESDecryptCTR	Decrypts a variable length data stream in the CTR mode.



NOTE. Intel IPP functions for cryptography operations do not allocate memory internally. The function `GetSize` does not require allocated memory. You need to call the function `GetSize` to find out how much available memory you need to have to work with the selected algorithm and after that you call the initialization function to create a memory buffer and initialize it.

Intel IPP for cryptography supports ECB, CBC, CFB, and CTR modes. You can tell which algorithm a given function supports from the function base name, for example, the function [DESEncryptECB](#) operates under the ECB mode for DES encryption and the function [TDESEncryptECB](#) operates under the ECB mode under the TDES scheme.

The encryption functions [DESEncryptCBC](#) and [TDESEncryptCBC](#) operate under the CBC mode using their respective cipher scheme and require to have an initialization vector *iv*. Since there exists a number of ways to initialize the initialization vector *iv*, you should remember which of these ways you used to be able to decrypt the message when needed.

Functions [DESEncryptCFB](#) and [TDESEncryptCFB](#) operate under CFB mode for encryption using their respective cipher scheme, both require having the initialization vector *pIV*, and CFB block size *cfbBlkSize*.

All functions described in this section use the context `IppsDESSpec` to serve as an operational vehicle that carries a set of round keys.

The application code for conducting a typical encryption under CBC mode using the TDES scheme must perform the following sequence of operations:

1. Get the size required to configure the context `IppsDESSpec` by calling the function [DESGetSize](#).
2. Call operating system memory allocation service function to allocate three buffers whose sizes are not less than the one specified by the function [DESGetSize](#). Initialize pointers to contexts *pCtx1*, *pCtx2*, and *pCtx3* by calling the function [DESInit](#) three times, each with the allocated buffer and the respective DES key.
3. Specify the initialization vector and then call the function [TDESEncryptCBC](#) to encrypt the input data stream under CBC mode using TDES scheme.
4. Free the memory allocated to the buffer once TDES encryption under the CBC mode has been completed and the data structures allocated for set of round keys are no longer required.



NOTE. Similar procedure can be applied for ECB, CFB, and CTR mode operation.

The `IppsDESSpec` context is position-dependent. The `DESpack/DESUnpack` functions transform the position-dependent context to a position-independent form and vice versa.

DESGetSize

Gets the size of the `IppsDESSpec` context.

Syntax

```
IppStatus ippsDESGetSize(int* pSize);
```

Parameters

pSize Pointer to the `IppsDESSpec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsDESSpec` context size in bytes and stores it in **pSize*.

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 <code>NULL</code> .

DESInit

Initializes user-supplied memory as the `IppsDESSpec` context for future use.

Syntax

```
IppStatus ippsDESInit(const Ipp8u* pKey, IppsDESSpec* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the DES key.
<i>pCtx</i>	Pointer to the <code>IppsDESSpec</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsDESSpec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .

DESPack, DESUnpack

Packs/unpacks the IppsDESSpec context into/from a user-defined buffer.

Syntax

```
IppStatus ippSDESPack (const IppsDESSpec* pCtx, Ipp8u* pBuffer);
IppStatus ippSDESPack (const Ipp8u* pBuffer, IppsDESSpec* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the IppsDESSpec context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `DESPack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `DESPack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal IppsDESSpec context. The `DESPack` and `DESPack` functions enable replacing the position-dependent IppsDESSpec context in the memory.

Call the `DESPackSize` function prior to `DESPack/DESPack` to determine the size of the buffer.

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 <code>NULL</code> .

DESEncryptECB

Encrypts a variable length data stream in the ECB mode.

Syntax

```
IppStatus ippSDESEncryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const IppsDESSpec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>DECSpec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by cipher block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DESDecryptECB

Decrypts a variable length data stream in the ECB mode.

Syntax

```
IppStatus ippSDESDecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsDESSpec* pCtx, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream of variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream of variable length.
<code>srclen</code>	Length of the ciphertext data stream in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDESSpec</code> context.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [[NIST SP 800-38A](#)].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by cipher block size.

DESEncryptCBC

Encrypts a variable length data stream in the CBC mode.

Syntax

```
IppStatus ippsDESEncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsDESSpec*
pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the plaintext data stream length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsDESSpec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [[NIST SP 800-38A](#)].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DESDecryptCBC

Decrypts a variable length data stream in the CBC mode.

Syntax

```
IppStatus ippsDESDecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsDESSpec*
pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of the variable length.

<code>srcLen</code>	Length of the ciphertext data stream length in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDESSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for CBC mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by cipher block size.

DESEncryptCFB

Encrypts a variable length data stream in the CFB mode.

Syntax

```
IppStatus ippseDESEncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int cfbBlkSize,  
const IppsDESSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>srcLen</code>	Length of the plaintext data stream in bytes.
<code>cfbBlkSize</code>	Size of the CFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDESSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the CFB mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by <code>cfbBlkSize</code> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DESDecryptCFB

Decrypts a variable length data stream in the CFB mode.

Syntax

```
IppStatus ippDESDecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int cfbBlkSize,
const IppsDESSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of variable length.
<code>srclen</code>	Length of the ciphertext data stream in bytes.
<code>cfbBlkSize</code>	Size of the CFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDESSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the CFB mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by cipher block size.

DESEncryptOFB

Encrypts a variable length data stream according to the DES algorithm in the OFB mode.

Syntax

```
IppStatus ippsDESEncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize,
const IppsDESSpec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>len</i>	Length of the plaintext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <i>IppsDESSpec</i> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DESDecryptOFB

Decrypts a variable length data stream according to the DES algorithm in the OFB mode.

Syntax

```
IppStatus ippsDESDecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize,
const IppsDESSpec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
-------------	---

<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>srcLen</i>	Length of the ciphertext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <i>IppsDESSpec</i> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DESEncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
ippStatus ippSDESEncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const IppsDESSpec* pCtx, Ipp8u*pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <i>IppsDESSpec</i> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DESDecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippDESDecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const IppsDESSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of a variable length.
<code>srcLen</code>	Length of the plaintext data stream in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDESSpec</code> context.
<code>pCtrValue</code>	Pointer to the counter data block.
<code>ctrNumBitSize</code>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TDESEncryptECB

Encrypts variable length data stream in ECB mode.

Syntax

```
IppStatus ippsTDESEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec *pCtx3, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Input plaintext data stream of a variable length.
<i>pDst</i>	Resulting ciphertext data stream.
<i>srclen</i>	Input data stream length in bytes.
<i>pCtx1</i>	First set of round keys scheduled for TDES internal operations.
<i>pCtx2</i>	Second set of round keys scheduled for TDES internal operations.
<i>pCtx3</i>	Third set of round keys scheduled for TDES internal operations.
<i>padding</i>	IppsCPPaddingNONE padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of supplied round keys in the ECB mode. The function returns the ciphertext result.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if the input data stream length is not divisible by cipher block size .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptECB

Decrypts variable length data stream in the ECB mode.

Syntax

```
IppStatus ippsTDESDecryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec *pCtx3, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Input ciphertext data stream of variable length.
-------------	--

<i>pDst</i>	Resulting plaintext data stream.
<i>srclen</i>	Input data stream length in bytes.
<i>pCtx1</i>	First set of round keys scheduled for TDES internal operations.
<i>pCtx2</i>	Second set of round keys scheduled for TDES internal operations.
<i>pCtx3</i>	Third set of round keys scheduled for TDES internal operations.
<i>padding</i>	IppsCPPaddingNONE padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of supplied round keys in the ECB mode. The function returns the ciphertext result and validates the final plaintext block.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the decrypted plaintext data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by cipher block size.

TDESEncryptCBC

Encrypts variable length data stream in the CBC mode.

Syntax

```
IppStatus ippstTDESEncryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec *pCtx3, const Ipp8u *pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Input plaintext data stream of a variable length.
<i>pDst</i>	Resulting ciphertext data stream.
<i>pIV</i>	Initialization vector for TDES CBC mode operation.
<i>srclen</i>	Input data stream length in bytes.
<i>pCtx1</i>	First set of round keys scheduled for TDES internal operations.
<i>pCtx2</i>	Second set of round keys scheduled for TDES internal operations.
<i>pCtx3</i>	Third set of round keys scheduled for TDES internal operations.
<i>padding</i>	IppsCPPaddingNONE padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of the supplied round keys in the Cipher Block Chaining (CBC) mode with the initialization vector. The function returns the ciphertext result.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if the input data stream length is not divisible by cipher block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptCBC

Decrypts variable length data stream in the CBC mode.

Syntax

```
IppStatus ippstDESDecryptCBC(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec *pCtx3, const Ipp8u *pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Input ciphertext data stream of a variable length.
<code>pDst</code>	Resulting plaintext data stream.
<code>pIV</code>	Initialization vector for TDES CBC mode operation.
<code>srclen</code>	Input data stream length in bytes.
<code>pCtx1</code>	First set of round keys scheduled for TDES internal operations.
<code>pCtx2</code>	Second set of round keys scheduled for TDES internal operations.
<code>pCtx3</code>	Third set of round keys scheduled for TDES internal operations.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of the supplied round keys in the Cipher Block Chaining (CBC) mode with the initialization vector. The function returns the ciphertext result and validates the final plaintext block.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the decrypted plaintext data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by cipher block size.

TDESEncryptCFB

Encrypts variable length data stream in the CFB mode.

Syntax

```
IppStatus ippstDESEncryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, int cfbBlkSize,
const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec *pCtx3, const Ipp8u
*pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Input plaintext data stream of variable length.
<code>pDst</code>	Resulting ciphertext data stream.
<code>pIV</code>	Initialization vector for TDES CFB mode operation.
<code>srclen</code>	Input data stream length in bytes.
<code>pCtx1</code>	First set of round keys scheduled for TDES internal operations.
<code>pCtx2</code>	Second set of round keys scheduled for TDES internal operations.
<code>pCtx3</code>	Third set of round keys scheduled for TDES internal operations.
<code>cfbBlkSize</code>	CFB block size in bytes.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of the supplied round keys in the Cipher Feedback (CFB) mode with the initialization vector. The function returns the ciphertext result.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.

<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by <code>cfbBlkSize</code> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptCFB

Decrypts variable length data stream in the CFB mode.

Syntax

```
IppStatus ippSTDESDecryptCFB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, int cfbBlkSize,
const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec *pCtx3, const Ipp8u
*pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Input ciphertext data stream of variable length.
<code>pDst</code>	Resulting plaintext data stream.
<code>pIV</code>	Initialization vector for TDES CFB mode operation.
<code>srclen</code>	Ciphertext data stream length in bytes.
<code>pCtx1</code>	First set of round keys scheduled for TDES internal operations.
<code>pCtx2</code>	Second set of round keys scheduled for TDES internal operations.
<code>pCtx3</code>	Third set of round keys scheduled for TDES internal operations.
<code>cfbBlkSize</code>	CFB block size in bytes.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A]. The function uses three sets of the supplied round keys in the Cipher Feedback (CFB) mode with the initialization vector. The function returns the ciphertext result and validates the final plaintext block.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the decrypted plaintext data stream length is less than or equal to zero.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by cipher block size.

TDSEncryptOFB

Encrypts a variable length data stream according to the TDES algorithm in the OFB mode.

Syntax

```
IppStatus ippSTDESEncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize,
const IppsDESSpec *pCtx1, const IppsDESSpec * pCtx2, const IppsDESSpec *pCtx3, Ipp8u*
pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the plaintext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx1</i>	First set of round keys scheduled for TDES internal operations.
<i>pCtx2</i>	Second set of round keys scheduled for TDES internal operations.
<i>pCtx3</i>	Third set of round keys scheduled for TDES internal operations.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptOFB

Decrypts a variable length data stream according to the TDES algorithm in the OFB mode.

Syntax

```
IppStatus ippsTDESDecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize,
const IppsDESSpec *pCtx1, const IppsDESSpec * pCtx2, const IppsDESSpec *pCtx3, Ipp8u*
pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>srclen</i>	Length of the ciphertext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx1</i>	First set of round keys scheduled for TDES internal operations.
<i>pCtx2</i>	Second set of round keys scheduled for TDES internal operations.
<i>pCtx3</i>	Third set of round keys scheduled for TDES internal operations.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TDESEncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsTDESEncryptCTR(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const IppsDESSpec
*pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec *pCtx3, Ipp8u *pCtrValue, int
ctrNumBitSize);
```

Parameters

<code>pSrc</code>	Input plaintext data stream of a variable length.
<code>pDst</code>	Resulting ciphertext data stream.
<code>srcLen</code>	Input data stream length in bytes.
<code>pCtx1</code>	First set of round keys scheduled for TDES internal operations.
<code>pCtx2</code>	Second set of round keys scheduled for TDES internal operations.
<code>pCtx3</code>	Third set of round keys scheduled for TDES internal operations.
<code>pCtrValue</code>	Counter.
<code>ctrNumBitSize</code>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in the [NIST SP 800-38A] recommendation. The function uses three sets of the supplied round keys. The standard incrementing function is applied to increment counter value. The function returns the ciphertext result.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TDSEDecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippstDESEDecryptCTR(const Ipp8u *pSrc, Ipp8u *pDst, int srcLen, const IppsDESSpec *pCtx1, const IppsDESSpec *pCtx2, const IppsDESSpec *pCtx3, Ipp8u *pCtrValue, int ctrNumBitSize);
```

Parameters

<code>pSrc</code>	Input ciphertext data stream of a variable length.
<code>pDst</code>	Resulting plaintext data stream.
<code>srcLen</code>	Length of the plaintext data stream in bytes.
<code>pCtx1</code>	First set of round keys scheduled for TDES internal operations.
<code>pCtx2</code>	Second set of round keys scheduled for TDES internal operations.

<code>pCtx3</code>	Third set of round keys scheduled for TDES internal operations.
<code>pCtrValue</code>	Counter.
<code>ctrNumBitSize</code>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the cipher scheme specified in the [NIST SP 800-38A] recommendation. The function uses three sets of the supplied round keys. The standard incrementing function is applied to increment value of counter. The function returns the ciphertext result.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the decrypted plaintext data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Example of Using DES/TDES Functions

DES/TDES Encryption and Decryption

```
// use of the ECB mode
void DES_sample(void){
    // size of the DES algorithm block is equal to 8
    const int desBlkSize = 8;

    // get size of the context needed for the encryption/decryption operation
    int ctxSize;
    ippsDESGetSize(&ctxSize);
    // and allocate one
    IppsDESSpec* pCtx = (IppsDESSpec*)( new Ipp8u [ctxSize] );

    // define the key
    Ipp8u key[] = {0x01,0x2,0x3,0x4,0x5,0x6,0x7,0x8};
    // and prepare the context for the DES usage
    ippsDESInit(key, pCtx);
    // define the message to be encrypted
    Ipp8u ptext[] = {"quick brown fox jum over lazy dog"};

    // allocate enough memory for the ciphertext
    // note that
    // the size of ciphertext is always multiple of cipher block size
    Ipp8u ctext[(sizeof(ptext)+desBlkSize-1) &~(desBlkSize-1)];
    // encrypt (ECB mode) ptext message
    // pay attention to the 'length' parameter
    // it defines the number of bytes to be encrypted
    ippsDESEncryptECB(ptext, ctext, sizeof(ctext),
                      pCtx,
                      IppsCPPaddingNONE);

    // allocate memory for the decrypted message
    Ipp8u rtext[sizeof(ctext)];
```

```

// decrypt (ECB mode) ctext message
// pay attention to the 'length' parameter
// it defines the number of bytes to be decrypted
IppsDESDecryptECB(ctext, rtext, sizeof(ctext),
                  pCtx,
                  IppsCPPaddingNONE);
delete (Ipp8u*)pCtx;
}

```

Rijndael Functions

Rijndael cipher scheme is an iterated block cipher with a variable block size and a variable key length. You can independently specify the lengths of the data block and the key as 128, 192, or 256 bits.

This section describes the functions operating in various operational modes under the various Rijndael cipher systems. The functions in this section are categorized by their data block sizes of the baseline Rijndael cipher functions:

- `Rijndael128` refers to the Rijndael cipher scheme with 128-bit data block size
- `Rijndael192` refers to the Rijndael cipher scheme with 192-bit data block size
- `Rijndael256` refers to the Rijndael cipher scheme with 256-bit data block size.

To specify the key length for these baseline Rijndael cipher schemes, all the functions in this section use the following enumeration

```

typedef enum {
IppsRijndaelKey128 = 128, // 128-bit key
IppsRijndaelKey192 = 192, // 192-bit key
IppsRijndaelKey256 = 256, // 256-bit key
} IppsRijndaelKeyLength;

```

The functions for Rijndael128 with the 128-bit key length described in this section are, in fact, American Encryption Standard (AES) cipher functions implemented in the way to comply with the American Standard FIPS 197. All other functions for various other Rijndael block cipher schemes fully comply to the respective cipher schemes documented by Joan Daeman and Vincent Rijmen.

Table “Intel IPP Rijndael Algorithm Functions” lists Intel IPP Rijndael functions:

Intel IPP Rijndael Algorithm Functions

Function Base Name	Operation
<code>Rijndael128GetSize</code>	Gets the size of the <code>IppsRijndael128Spec</code> context.
<code>Rijndael128Init</code> , <code>SafeRijndael128Init</code>	Initialize user-supplied memory as <code>IppsRijndael128Spec</code> context for future use.
<code>Rijndael128Pack</code> , <code>Rijndael128Unpack</code>	Packs/unpacks the <code>IppsRijndael128Spec</code> context into/from a user-defined buffer.

Function Base Name	Operation
Rijndael128EncryptECB	Encrypts plaintext message using Rijndael128 algorithm in the ECB encryption mode.
Rijndael128DecryptECB	Decrypts byte data stream using Rijndael128 algorithm in the ECB mode.
Rijndael128EncryptCBC	Encrypts byte data stream according to Rijndael128 in the CBC mode.
Rijndael128DecryptCBC	Decrypts byte data stream according to Rijndael128 in the CBC mode.
Rijndael128EncryptCFB	Encrypts byte data stream according to Rijndael128 in the CFB mode.
Rijndael128DecryptCFB	Decrypts byte data stream according to Rijndael128 in the CFB mode.
Rijndael128EncryptOFB	Encrypts byte data stream according to Rijndael128 in the OFB mode.
Rijndael128DecryptOFB	Decrypts byte data stream according to Rijndael128 in the OFB mode.
Rijndael128EncryptCTR	Encrypts a variable length data stream according to Rijndael128 in the CTR mode.
Rijndael128DecryptCTR	Decrypts a variable length data stream according to Rijndael128 in the CTR mode.
Rijndael128EncryptCCM	DEPRECATED. Use Rijndael128CCMEncryptMessage instead of this function. Encrypts a variable length data stream and generates its authentication tag in the CCM mode.
Rijndael128DecryptCCM	DEPRECATED. Use Rijndael128CCMDecryptMessage instead of this function. Decrypts and verifies a variable length data stream in the CCM mode.
Rijndael128EncryptCCM_u8	DEPRECATED. Use Rijndael128CCMEncryptMessage instead of this function. Encrypts a variable length data stream and generates its authentication tag in the CCM mode using enhanced interface.
Rijndael128DecryptCCM_u8	DEPRECATED. Use Rijndael128CCMDecryptMessage instead of this function. Decrypts and verifies a variable length data stream in the CCM mode using enhanced interface.
Rijndael192GetSize	Gets the size of the <code>IppsRijndael192Spec</code> context.

Function Base Name	Operation
<code>Rijndael192Init</code>	Initializes user-supplied memory as <code>IppsRijndael192Spec</code> context for future use.
<code>Rijndael192Pack</code> , <code>Rijndael192Unpack</code>	Packs/unpacks the <code>IppsRijndael192Spec</code> context into/from a user-defined buffer.
<code>Rijndael192EncryptECB</code>	Encrypts plaintext message using Rijndael192 algorithm in the ECB encryption mode.
<code>Rijndael192DecryptECB</code>	Decrypts byte data stream using Rijndael192 algorithm in the ECB mode.
<code>Rijndael192EncryptCBC</code>	Encrypts byte data stream according to Rijndael192 in the CBC mode.
<code>Rijndael192DecryptCBC</code>	Decrypts byte data stream according to Rijndael192 in the CBC mode.
<code>Rijndael192EncryptCFB</code>	Encrypts byte data stream according to Rijndael192 in the CFB mode.
<code>Rijndael192DecryptCFB</code>	Decrypts byte data stream according to Rijndael192 in the CFB mode.
<code>Rijndael192EncryptOFB</code>	Encrypts byte data stream according to Rijndael192 in the OFB mode.
<code>Rijndael192DecryptOFB</code>	Decrypts byte data stream according to Rijndael192 in the OFB mode.
<code>Rijndael192EncryptCTR</code>	Encrypts a variable length data stream according to Rijndael192 in the CTR mode.
<code>Rijndael192DecryptCTR</code>	Decrypts a variable length data stream according to Rijndael192 in the CTR mode.
<code>Rijndael256GetSize</code>	Gets the size of the <code>IppsRijndael256Spec</code> context.
<code>Rijndael256Init</code>	Initializes user-supplied memory as <code>IppsRijndael256Spec</code> context for future use.
<code>Rijndael256Pack</code> , <code>Rijndael256Unpack</code>	Packs/unpacks the <code>IppsRijndael256Spec</code> context into/from a user-defined buffer.
<code>Rijndael256EncryptECB</code>	Encrypts plaintext message using Rijndael256 algorithm in the ECB encryption mode.
<code>Rijndael256DecryptECB</code>	Decrypts byte data stream using Rijndael256 algorithm in the ECB mode.
<code>Rijndael256EncryptCBC</code>	Encrypts byte data stream according to Rijndael256 in the CBC mode.

Function Base Name	Operation
Rijndael256DecryptCBC	Decrypts byte data stream according to Rijndael256 in the CBC mode.
Rijndael256EncryptCFB	Encrypts byte data stream according to Rijndael256 in the CFB mode.
Rijndael256DecryptCFB	Decrypts byte data stream according to Rijndael256 in the CFB mode.
Rijndael256EncryptOFB	Encrypts byte data stream according to Rijndael256 in the OFB mode.
Rijndael256DecryptOFB	Decrypts byte data stream according to Rijndael256 in the OFB mode.
Rijndael256EncryptCTR	Encrypts a variable length data stream according to Rijndael256 in the CTR mode.
Rijndael256DecryptCTR	Decrypts a variable length data stream according to Rijndael256 in the CTR mode.

¹ Obsolete. Use [AES-CCM Functions](#).

Throughout this section, the functions for Rijndael128 baseline cipher schemes employ the context `IppsRijndael128Spec`, the functions for Rijndael192 baseline cipher schemes employ the context `IppsRijndael192Spec`, and the functions for Rijndael256 baseline cipher schemes employ the context `IppsRijndael256Spec`. They serve as operational vehicles to carry not only a set of round keys and a set of round inverse keys at the same time, but also the key management information.

Once the respective initialization function generates the round keys, the functions for ECB, CBC, CFB, and other modes are ready for the execution of either encrypting or decrypting the streaming data with the specified padding scheme.

The Intel IPP versions 5.3 or lower employed the implementation of AES (that is, Rijndael128) based on the use of large pre-calculated tables (S-boxes). This implementation provides the best performance. However, the research done in recent years proved vulnerability of this solution to various attacks, for example, timing and cache-behavior attacks. To provide a proper level of protection, IPP 6.0 introduces a safe implementation of the AES algorithm. Though 1.3 times slower than the existing one, the safe implementation is invulnerable to the known implementations of timing and cache-behavior attacks. To use Rijndael128 functions with the safe implementation of the algorithm, call initialization function [SafeRijndael128Init](#). If performance is the priority, call [Rijndael128Init](#).

The application code for conducting a typical encryption under CBC mode using the AES scheme, that is, the Rijndael128 with a 128-bit key, should follow the sequence of operations as outlined below:

1. Get the size required to configure the context `IppsRijndael128Spec` by calling the function [Rijndael128GetSize](#).
2. Call the operating system memory-allocation service function to allocate a buffer whose size is no less than the one specified by the function [Rijndael128GetSize](#).
3. Initialize the context `IppsRijndael128Spec *pCtx` by calling the function [Rijndael128Init](#) with the allocated buffer and the respective 128-bit AES key.
4. Specify the initialization vector and the padding scheme, then call the function [Rijndael128EncryptCBC](#) to encrypt the input data stream using the AES encryption function with CBC mode.

5. Call the operating system memory free service function to release the buffer allocated for the context `IppsRijndael128Spec`, if needed.

The `IppsRijndael128Spec`, `IppsRijndael192Spec`, and `IppsRijndael224Spec` contexts are position-dependent. The [Rijndael128Pack/Rijndael128Unpack](#), [Rijndael192Pack/Rijndael192Unpack](#), and [Rijndael256Pack/Rijndael256Unpack](#) functions transform the respective position-dependent context to a position-independent form and vice versa.

See Also

- [AES-CCM Functions](#)
- [AES-GCM Functions](#)

Rijndael128GetSize

Gets the size of the `IppsRijndael128Spec` context.

Syntax

```
IppStatus ippsRijndael128GetSize(int* pSize);
```

Parameters

pSize Pointer to the `IppsRijndael128Spec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsRijndael128Spec` context size in bytes and stores it in **pSize*.

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 <code>NULL</code> .

Rijndael128Init, SafeRijndael128Init

Initialize user-supplied memory as `IppsRijndael128Spec` context for future use.

Syntax

```
IppStatus ippsRijndael128Init(const Ipp8u *pKey, IppsRijndaelKeyLength keylen,
IppsRijndael128Spec* pCtx);
```

```
IppStatus ippsSafeRijndael128Init(const Ipp8u *pKey, IppsRijndaelKeyLength keylen,
IppsRijndael128Spec* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the Rijndael128 key.
<i>keylen</i>	Key byte stream length in bytes defined by the <code>IppsRijndaelKeyLength</code> enumerator.

pCtx Pointer to the `IppsRijndael128Spec` context being initialized.

Description

These functions are declared in the `ippcp.h` file. Each function initializes the memory pointed by *pCtx* as `IppsRijndael128Spec`. In addition, each function uses the key to provide all necessary key material for both encryption and decryption operations. Depending upon whether you wish to employ fast or safe implementation of the AES algorithm, call `Rijndael128Init` or `SafeRijndael128Init`, 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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Returns an error condition if <i>keyLen</i> is not set to <code>IppsRijndaelKey128</code> , <code>IppsRijndaelKey192</code> or <code>IppsRijndaelKey256</code> .

Rijndael128Pack, Rijndael128Unpack

Packs/unpacks the `IppsRijndael128Spec` context into/from a user-defined buffer.

Syntax

```
IppStatus ippRijndael128Pack (const IppsRijndael128Spec* pCtx, Ipp8u* pBuffer);  
IppStatus ippRijndael128Unpack (const Ipp8u* pBuffer, IppsRijndael128Spec* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

These functions are declared in the `ippcp.h` file. The `Rijndael128Pack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `Rijndael128Unpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal `IppsRijndael128Spec` context. The `Rijndael128Pack` and `Rijndael128Unpack` functions enable replacing the position-dependent `IppsRijndael128Spec` context in the memory.

Call the [Rijndael128GetSize](#) function prior to `Rijndael128Pack`/`Rijndael128Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

Rijndael128EncryptECB

Encrypts plaintext message by using ECB encryption mode.

Syntax

```
IppStatus ippsRijndael128EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const
IppsRijndael128Spec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the input plaintext data in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by cipher block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptECB

Decrypts byte data stream by using Rijndael algorithm in the ECB mode.

Syntax

```
IppStatus ippsRijndael128DecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const
IppsRijndael128Spec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream of variable length.
<i>srclen</i>	Length of the ciphertext data stream in bytes.

<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by cipher block size.

Rijndael128EncryptCBC

Encrypts byte data stream according to Rijndael in the CBC mode.

Syntax

```
ippStatus ippRijndael128EncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsRijndael128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the plaintext data stream length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .

<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCBC

Decrypts byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippRijndael128DecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const
IppsRijndael128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of the variable length.
<code>srclen</code>	Length of the ciphertext data stream length in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael128Spec</code> context.
<code>pIV</code>	Pointer to the initialization vector for CBC mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by cipher block size.

Rijndael128EncryptCFB

Encrypts byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippsRijndael128EncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int cfbBlkSize, const IppsRijndael128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by <i>cfbBlkSize</i> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <i>cfbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCFB

Decrypts byte data stream according to Rijndael in CFB mode.

Syntax

```
IppStatus ippsRijndael128DecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int cfbBlkSize, const IppsRijndael128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```


Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of variable length.
<i>srcLen</i>	Length of the ciphertext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <i>cfbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by cipher block size.

Rijndael128EncryptOFB

Encrypts a variable length data stream according to Rijndael128 in the OFB mode.

Syntax

```
IppStatus ippRijndael128EncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int ofbBlkSize, const IppsRijndael128Spec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by the <code>ofbBlkSize</code> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <code>ofbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptOFB

Decrypts a variable length data stream according to Rijndael128 in the OFB mode.

Syntax

```
IppStatus ippRijndael128DecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize, const IppsRijndael128Spec* pCtx, Ipp8u* pIV);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream of variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream.
<code>srclen</code>	Length of the ciphertext data stream in bytes.
<code>ofbBlkSize</code>	Size of the OFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael128Spec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.

<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128EncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippRijndael128EncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const
IppsRijndael128Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <i>ctrNumBitSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippsRijndael128DecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const  
IppsRijndael128Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of a variable length.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128EncryptCCM

DEPRECATED. Encrypts a variable length data stream and generates its authentication tag in the CCM mode.

Syntax

```
IppStatus ippsRijndael128EncryptCCM(const Ipp8u* pNonce, Ipp32u nonceLen, const Ipp8u*  
pAssc, Ipp64u asscLen, const Ipp8u* pSrc, Ipp64u srcLen, int macLen, Ipp8u* pDst, const  
IppsRijndael128Spec* pCtx);
```

Parameters

<code>pNonce</code>	Pointer to the nonce.
<code>nonceLen</code>	Length of the nonce <code>*pNonce</code> (in octets).
<code>pAssc</code>	Pointer to the associated data.
<code>asscLen</code>	Length of the associated data <code>*pAssc</code> (in octets).
<code>pSrc</code>	Pointer to the input plaintext data.
<code>srcLen</code>	Length of the input plaintext <code>*pSrc</code> (in octets).
<code>macLen</code>	Length of the authentication tag in octets.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>pCtx</code>	Pointer to the <code>IppsRijndael128Spec</code> context.

Description

This function is deprecated. Use [Rijndael128CCMEncryptMessage](#) instead of it.

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length and computes the authentication tag according to the Counter with Cipher Block Chaining-Message Authentication Code (CCM) mode, as specified in [NIST SP 800-38C]. Unless you have successfully implemented the encryption with this function, you are encouraged to use the newer function [Rijndael128EncryptCCM_u8](#) having the same functionality.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the length input data does not meet any of the following conditions: $7 \leq \text{nonceLen} \leq 13$ $\text{asscLen} \geq 0$ $\text{srcLen} \geq 1$ $\text{macLen} = 2n, 1 \leq n \leq 8$.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128EncryptCCM_u8

DEPRECATED. *Encrypts a variable length data stream and generates its authentication tag in the CCM mode.*

Syntax

```
IppStatus ippRijndael128EncryptCCM_u8(const Ipp8u* pNonce, int nonceLen, const Ipp8u*
pAssc, int asscLen, const Ipp8u* pSrc, int srcLen, int macLen, Ipp8u* pDst, const
IppsRijndael128Spec* pCtx);
```

Parameters

<i>pNonce</i>	Pointer to the nonce.
<i>nonceLen</i>	Length of the nonce <i>*pNonce</i> (in octets).
<i>pAssc</i>	Pointer to the associated data.
<i>asscLen</i>	Length of the associated data <i>*pAssc</i> (in octets).
<i>pSrc</i>	Pointer to the input plaintext data.
<i>srcLen</i>	Length of the input plaintext <i>*pSrc</i> (in octets).
<i>macLen</i>	Length of the authentication tag in octets.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.

Description

This function is deprecated. Use [Rijndael128CCMEncryptMessage](#) instead of it.

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length and computes the authentication tag according to the CCM mode, as specified in [NIST SP 800-38C]. Unlike [Rijndael128EncryptCCM](#), `Rijndael128EncryptCCM_u8` employs the `int` data type for *all* length parameters. Use this function rather than `Rijndael128EncryptCCM`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the length input data does not meet any of the following conditions: $7 \leq \textit{nonceLen} \leq 13$ $\textit{asscLen} \geq 0$ $\textit{srcLen} \geq 1$ $\textit{macLen} = 2n, 1 \leq n \leq 8$.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCCM

DEPRECATED. Decrypts and verifies a variable length data stream in the CCM mode.

Syntax

```
IppStatus ippRijndael128DecryptCCM(const Ipp8u* pNonce, Ipp32u nonceLen, const Ipp8u* pAssc, Ipp64u asscLen, const Ipp8u* pSrc, Ipp64u srcLen, int macLen, Ipp8u* pDst, IppBool* pResult, const IppsRijndael128Spec* pCtx);
```

Parameters

<code>pNonce</code>	Pointer to the nonce of length.
<code>nonceLen</code>	Length of the nonce <code>*pNonce</code> (in octets).
<code>pAssc</code>	Pointer to the associated data.
<code>asscLen</code>	Length of the associated data <code>*pAssc</code> (in octets).
<code>pSrc</code>	Pointer to the input ciphertext data.
<code>srcLen</code>	Length of the input ciphertext <code>*pSrc</code> (in octets).
<code>macLen</code>	Length of the authentication tag in octets.
<code>pDst</code>	Pointer to the resulting plaintext data stream.
<code>pResult</code>	Pointer to the result of verification.
<code>pCtx</code>	Pointer to the <code>IppsRijndael128Spec</code> context.

Description

This function is deprecated. Use [Rijndael128CCMDecryptMessage](#) instead of this function.

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length and verifies the authentication tag according to the CCM mode, as specified in [NIST SP 800-38C]. Unless you have successfully implemented the decryption with this function, you are encouraged to use the newer function [Rijndael128DecryptCCM_u8](#) having the same functionality.

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>ippStsLengthErr</code>	Indicates an error condition if the length input data does not meet any of the following conditions: $7 \leq \text{nonceLen} \leq 13$ $\text{srcLen} > \text{macLen}$ $\text{macLen} = 2n, 1 \leq n \leq 8$.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128DecryptCCM_u8

DEPRECATED. Decrypts and verifies a variable length data stream in the CCM mode.

Syntax

```
IppStatus ippRijndael128DecryptCCM_u8(const Ipp8u* pNonce, int nonceLen, const Ipp8u* pAssc, int asscLen, const Ipp8u* pSrc, int srcLen, int macLen, Ipp8u* pDst, IppBool* pResult, const IppsRijndael128Spec* pCtx);
```

Parameters

<code>pNonce</code>	Pointer to the nonce of length.
---------------------	---------------------------------

<i>nonceLen</i>	Length of the nonce <i>*pNonce</i> (in octets).
<i>pAssc</i>	Pointer to the associated data.
<i>asscLen</i>	Length of the associated data <i>*pAssc</i> (in octets).
<i>pSrc</i>	Pointer to the input ciphertext data.
<i>srcLen</i>	Length of the input ciphertext <i>*pSrc</i> (in octets).
<i>macLen</i>	Length of the authentication tag in octets.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>pResult</i>	Pointer to the result of verification.
<i>pCtx</i>	Pointer to the <code>IppsRijndael128Spec</code> context.

Description

This function is deprecated. Use [Rijndael128CCMDecryptMessage](#) instead of it.

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length and verifies the authentication tag according to the CCM mode, as specified in [NIST SP 800-38C]. Unlike [Rijndael128DecryptCCM](#), `Rijndael128DecryptCCM_u8` employs the `int` data type for *all* length parameters. Use this function rather than `Rijndael128DecryptCCM`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the length input data does not meet any of the following conditions: $7 \leq \textit{nonceLen} \leq 13$ $\textit{asscLen} \geq 0$ $\textit{srcLen} > \textit{macLen}$ $\textit{macLen} = 2n, 1 \leq n \leq 8$.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael192GetSize

Gets the size of the `IppsRijndael192Spec` context.

Syntax

```
IppStatus ippRijndael192GetSize(int* pSize);
```

Parameters

<i>pSize</i>	Pointer to the <code>IppsRijndael192Spec</code> context size value.
--------------	---

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsRijndael192Spec` context size in bytes and stores it in **pSize*.

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 <code>NULL</code> .

Rijndael192Init

Initializes user-supplied memory as the `IppsRijndael192Spec` context for future use.

Syntax

```
IppStatus ippRijndael192Init(const Ipp8u *pKey, IppsRijndaelKeyLength keylen,
IppsRijndael192Spec* pCtx);
```

Parameters

<code>pKey</code>	Pointer to the Rijndael192 key.
<code>keylen</code>	Key byte stream length in bytes as defined by the <code>IppsRijndaelKeyLength</code> enumerator.
<code>pCtx</code>	Pointer to the <code>IppsRijndael192Spec</code> context.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as the `IppsRijndael192Spec` context. In addition, the function uses the key to provide all the necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keylen</code> is not set to <code>IppsRijndaelKey128</code> , <code>IppsRijndaelKey192</code> or <code>IppsRijndaelKey256</code> .

Rijndael192Pack, Rijndael192Unpack

Packs/unpacks the `IppsRijndael192Spec` context into/from a user-defined buffer.

Syntax

```
IppStatus ippRijndael192Pack (const IppsRijndael192Spec* pCtx, Ipp8u* pBuffer);
IppStatus ippRijndael192Unpack (const Ipp8u* pBuffer, IppsRijndael192Spec* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsRijndael192Spec</code> context.
-------------------	--

pBuffer Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `Rijndael192Pack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `Rijndael192Unpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal `IppsRijndael192Spec` context. The `Rijndael192Pack` and `Rijndael192Unpack` functions enable replacing the position-dependent `IppsRijndael192Spec` context in the memory.

Call the `Rijndael192GetSize` function prior to `Rijndael192Pack`/`Rijndael192Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

Rijndael192EncryptECB

Encrypts a byte data stream according to Rijndael in the ECB mode.

Syntax

```
IppStatus ippRijndael192EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const IppsRijndael192Spec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the input data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael192Spec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.

<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptECB

Decrypts byte data stream according to Rijndael in the ECB mode.

Syntax

```
IppStatus ippRijndael192DecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const
IppsRijndael192Spec *pCtx, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream of variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream of variable length.
<code>srclen</code>	Length of the ciphertext data stream in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael192Spec</code> context.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by cipher block size.

Rijndael192EncryptCBC

Encrypts a byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippRijndael192EncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const
IppsRijndael192Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the plaintext data stream length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael192Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptCBC

Decrypts a byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippRijndael192DecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsRijndael192Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of the variable length.
<i>srclen</i>	Length of the ciphertext data stream length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael192Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by cipher block size.

Rijndael192EncryptCFB

Encrypts a byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippRijndael192EncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int
cfbBlkSize, const IppsRijndael192Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>srcLen</code>	Length of the plaintext data stream in bytes.
<code>cfbBlkSize</code>	Size of the CFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael192Spec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the CFB mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.

<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by <code>cfbBlkSize</code> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptCFB

Decrypts a byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippRijndael192DecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int cfbBlkSize, const IppsRijndael192Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of variable length.
<code>srcLen</code>	Length of the ciphertext data stream in bytes.
<code>cfbBlkSize</code>	Size of the CFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael192Spec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the CFB mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by cipher block size.

Rijndael192EncryptOFB

Encrypts a variable length data stream according to Rijndael192 in the OFB mode.

Syntax

```
IppStatus ippsRijndael192EncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize, const IppsRijndael192Spec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the plaintext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael192Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptOFB

Decrypts a variable length data stream according to Rijndael192 in the OFB mode.

Syntax

```
IppStatus ippsRijndael192DecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize, const IppsRijndael192Spec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
-------------	---

<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>srcLen</i>	Length of the ciphertext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael192Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael192EncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippRijndael192EncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const
IppsRijndael192Spec* pCtx, Ipp8u* pCtrValue , int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael192Spec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael192DecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippRijndael192DecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const
IppsRijndael192Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of a variable length.
<code>srcLen</code>	Length of the plaintext data stream in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael192Spec</code> context.
<code>pCtrValue</code>	Pointer to the counter data block.
<code>ctrNumBitSize</code>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael256GetSize

Gets the size of the `IppsRijndael256Spec` context.

Syntax

```
IppStatus ippsRijndael256GetSize(int* pSize);
```

Parameters

pSize Pointer to the `IppsRijndael256Spec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsRijndael256Spec` context size in bytes and stores it in **pSize*.

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 <code>NULL</code> .

Rijndael256Init

Initializes user-supplied memory as `IppsRijndael256Spec` context for future use.

Syntax

```
IppStatus ippsRijndael256Init(const Ipp8u *pKey, IppsRijndaelKeyLength keylen,  
IppsRijndael256Spec* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the Rijndael256 key.
<i>keylen</i>	Key byte stream length in bytes defined by the <code>IppsRijndaelKeyLength</code> enumerator.
<i>pCtx</i>	Pointer to the <code>IppsRijndael256Spec</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsRijndael256Spec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .

`ippStsLengthErr` Indicates an error condition if `keylen` is not set to
`IppsRijndaelKey128`, `IppsRijndaelKey192` or
`IppsRijndaelKey256`.

Rijndael256Pack, Rijndael256Unpack

Packs/unpacks the `IppsRijndael256Spec` context into/from a user-defined buffer.

Syntax

```
IppStatus ippRijndael256Pack (const IppsRijndael256Spec* pCtx, Ipp8u* pBuffer);
IppStatus ippRijndael256Unpack (const Ipp8u* pBuffer, IppsRijndael256Spec* pCtx);
```

Parameters

`pCtx` Pointer to the `IppsRijndael256Spec` context.
`pBuffer` Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `Rijndael256Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the `*pBuffer` buffer. The `Rijndael256Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsRijndael256Spec` context. The `Rijndael256Pack` and `Rijndael256Unpack` functions enable replacing the position-dependent `IppsRijndael256Spec` context in the memory.

Call the `Rijndael256GetSize` function prior to `Rijndael256Pack`/`Rijndael256Unpack` to determine the size of the buffer.

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`.

Rijndael256EncryptECB

Encrypts a byte data stream according to Rijndael in the ECB mode.

Syntax

```
IppStatus ippRijndael256EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const
IppsRijndael256Spec* pCtx, IppsCPPadding padding);
```

Parameters

`pSrc` Pointer to the input plaintext data stream of variable length.
`pDst` Pointer to the resulting ciphertext data stream.
`srclen` Length of the input data stream in bytes.

<i>pCtx</i>	Pointer to the <code>IppsRijndael256Spec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptECB

Decrypts a byte data stream according to Rijndael in the ECB mode.

Syntax

```
IppStatus ippRijndael256DecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsRijndael256Spec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream of variable length.
<i>srclen</i>	Length of the ciphertext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael256Spec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .

<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by cipher block size.

Rijndael256EncryptCBC

Encrypts byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippRijndael256EncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const
IppsRijndael256Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>srclen</code>	Length of the plaintext data stream length in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael256Spec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the CBC mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptCBC

Decrypts byte data stream according to Rijndael in the CBC mode.

Syntax

```
IppStatus ippRijndael256DecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const IppsRijndael256Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of the variable length.
<i>srcLen</i>	Length of the ciphertext data stream length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael256Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by cipher block size.

Rijndael256EncryptCFB

Encrypts byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippRijndael256EncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int cfbBlkSize, const IppsRijndael256Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
-------------	--

<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael256Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by <i>cfbBlkSize</i> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <i>cfbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptCFB

Decrypts byte data stream according to Rijndael in the CFB mode.

Syntax

```
IppStatus ippRijndael256DecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int
cfbBlkSize, const IppsRijndael256Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of variable length.
<i>srclen</i>	Length of the ciphertext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael256Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by cipher block size.

Rijndael256EncryptOFB

Encrypts a variable length data stream according to Rijndael256 in the OFB mode.

Syntax

```
IppStatus ippRijndael256EncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize, const IppsRijndael256Spec* pCtx, Ipp8u* pIV);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>srclen</code>	Length of the plaintext data stream in bytes.
<code>ofbBlkSize</code>	Size of the OFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael256Spec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.

<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptOFB

Decrypts a variable length data stream according to Rijndael256 in the OFB mode.

Syntax

```
IppStatus ippRijndael256DecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int
ofbBlkSize, const IppsRijndael256Spec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>srcLen</i>	Length of the ciphertext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael256Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael256EncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippRijndael256EncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const IppsRijndael256Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsRijndael256Spec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael256DecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippRijndael256DecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const IppsRijndael256Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of a variable length.

<code>srcLen</code>	Length of the plaintext data stream in bytes.
<code>pCtx</code>	Pointer to the <code>IppsRijndael1256Spec</code> context.
<code>pCtrValue</code>	Pointer to the counter data block.
<code>ctrNumBitSize</code>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [\[NIST SP 800-38A\]](#).

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Example of Using Rijndael Functions

AES Encryption and Decryption

```
// use of the CTR mode
void AES_sample(void){
    // size of Rijndael-128 algorithm block is equal to 16
    const int aesBlkSize = 16;

    // get the size of the context needed for the encryption/decryption operation
    int ctxSize;
    ippsRijndael128GetSize(&ctxSize);

    // and allocate one
    IppsRijndael128Spec* pCtx = (IppsRijndael128Spec*)( new Ipp8u [ctxSize] );
    // define the key
    Ipp8u key[16] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                    0x08,0x09,0x10,0x11,0x12,0x13,0x14,0x15};

    // and prepare the context for Rijndael128 usage
    ippsRijndael128Init(key,IppsRijndaelKey128, pCtx);

    // define the message to be encrypted
    Ipp8u ptext[] = {"quick brown fox jum over lazy dog"};

    // define an initial vector
    Ipp8u crt0[aesBlkSize] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
                             0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
    Ipp8u crt[aesBlkSize];

    // counter the variable number of bits
    int ctrNumBitSize = 64;

    // allocate enough memory for the ciphertext
    // note that
```

```
// the size of the ciphertext is always equal to that of the plaintext
Ipp8u ctext[sizeof(ptext)];

// init the counter
memcpy(crt, crt0, sizeof(crt0));
// encrypt (CTR mode) ptext message
// pay attention to the 'length' parameter
// it defines the number of bytes to be encrypted
memcpy(crt, crt0, sizeof(crt0));
ippsRijndael128EncryptCTR(ptext, ctext, sizeof(ptext),
                           pCtx,
                           crt, ctrNumBitSize);

// allocate memory for the decrypted message
Ipp8u rtext[sizeof(ptext)];

// init the counter
memcpy(crt, crt0, sizeof(crt0));

// decrypt (ECTR mode) ctext message
// pay attention to the 'length' parameter
// it defines the number of bytes to be decrypted
ippsRijndael128DecryptCTR(ctext, rtext, sizeof(ptext),
                           pCtx,
                           crt, ctrNumBitSize);

delete (Ipp8u*)pCtx;
}
```

AES-CCM Functions

This section describes functions for authenticated encryption/decryption using the Counter with Cipher Block Chaining-Message Authentication Code (CCM) mode [\[NIST SP 800-38C\]](#) of the AES (Rijndael128) block cipher.

[Table "Intel IPP AES-CCM Functions"](#) lists Intel IPP AES-CCM functions:

Intel IPP AES-CCM Functions

Function Base Name	Operation
Rijndael128CCMEncryptMessage	Encrypts an entire message and generates its authentication tag in the CCM mode.
Rijndael128CCMDecryptMessage	Decrypts an entire message and generates its authentication tag in the CCM mode.
Rijndael128CCMGetSize	Gets the size of the <code>IppsRijndael128CCMState</code> context.
Rijndael128CCMInit	Initializes user-supplied memory as the <code>IppsRijndael128CCMState</code> context for future use.
Rijndael128CCMStart	Starts the process of authenticated encryption/decryption for a new message.
Rijndael128CCMEncrypt	Encrypts a data buffer in the CCM mode.
Rijndael128CCMDecrypt	Decrypts a data buffer in the CCM mode.
Rijndael128CCMGetTag	Generates the message authentication tag in the CCM mode.
Rijndael128CCMMessageLen	Sets up the length of the message to be processed.
Rijndael128CCMTagLen	Sets up the length of the required authentication tag.

The Intel IPP AES-CCM function set includes:

- Functions for complete message operations, to be used for authenticated encryption/decryption of an entire message, if available: [Rijndael128CCMEncryptMessage](#) and [Rijndael128CCMDecryptMessage](#)
- Incremental functions, to be used if the message is too long to be processed in one step.

The AES-CCM incremental functions enable authenticated encryption/decryption of several messages using one key that the [Rijndael128CCMInit](#) function sets. The processing of each new message starts with a call to the [Rijndael128CCMStart](#) function. The application code for conducting a typical AES-CCM authenticated encryption should follow the sequence of operations as outlined below:

1. Get the size required to configure the context `IppsRijndael128CCMState` by calling the function [Rijndael128CCMGetSize](#).
2. Call the system memory-allocation service function to allocate a buffer whose size is not less than the function [Rijndael128CCMGetSize](#) specifies.
3. Initialize the context `IppsRijndael128CCMState *pCtx` by calling the function [Rijndael128CCMInit](#) with the allocated buffer and the respective AES key.
4. Optionally call [Rijndael128CCMMessageLen](#) and/or [Rijndael128CCMTagLen](#) to set up message and tag parameters.
5. Call [Rijndael128CCMStart](#) to start authenticated encryption of the first/next message.
6. Keep calling [Rijndael128CCMEncrypt](#) until the entire message is processed.
7. Request the authentication tag by calling [Rijndael128CCMGetTag](#).
8. Proceed to the next message, if any, that is, go to step 5.
9. Call the system memory free service function to release the buffer allocated for the context `IppsRijndael128CCMState`, if needed.

Rijndael128CCMEncryptMessage

Encrypts an entire message and generates its authentication tag in the CCM mode.

Syntax

```
IppStatus ippsRijndael128CCMEncryptMessage(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,
const Ipp8u* pIV, Ipp32u ivLen, const Ipp8u* pAAD, Ipp32u aadLen, const Ipp8u* pSrc, Ipp8u*
pDst, Ipp32u len, Ipp8u* pTag, Ipp32u tagLen);
```

Parameters

<i>pKey</i>	Pointer to the secret key.
<i>keyLength</i>	Length of the secret key.
<i>pIV</i>	Pointer to the initialization vector.
<i>ivLen</i>	Length of the initialization vector <i>*pIV</i> (in bytes).
<i>pAAD</i>	Pointer to the additional authenticated data (not to be encrypted).
<i>aadLen</i>	Length of the additional authenticated data <i>*pAAD</i> (in bytes).
<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>len</i>	Length of the plaintext and ciphertext data stream (in bytes).
<i>pTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the authentication tag <i>*pTag</i> (in bytes).

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length and computes the authentication tag in the CCM mode as specified in [NIST SP 800-38C]. Use this function if the entire message is available. Otherwise, follow the typical sequence of invoking the incremental functions (see [AES-CCM Functions](#)).

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>ippStsLengthErr</code>	Indicates an error condition if the length parameters do not meet any of the following conditions: $keyLen = \text{IppsRijndaelKey128}$ or $keyLen = \text{IppsRijndaelKey192}$ or $keyLen = \text{IppsRijndaelKey256}$ $7 \leq ivLen \leq 13$ $4 \leq tagLen \leq 16$ and $tagLen$ is even.

Rijndael128CCMDecryptMessage

Decrypts an entire message and generates its authentication tag in the CCM mode.

Syntax

```
IppStatus ippsRijndael128CCMDecryptMessage(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,
const Ipp8u* pIV, Ipp32u ivLen, const Ipp8u* pAAD, Ipp32u aadLen, const Ipp8u* pSrc, Ipp8u*
pDst, Ipp32u len, Ipp8u* pTag, Ipp32u tagLen);
```

Parameters

<i>pKey</i>	Pointer to the secret key.
<i>keyLength</i>	Length of the secret key.
<i>pIV</i>	Pointer to the initialization vector.
<i>ivLen</i>	Length of the initialization vector <i>*pIV</i> (in bytes).
<i>pAAD</i>	Pointer to the additional authenticated data (not encrypted).
<i>aadLen</i>	Length of additional authenticated data <i>*pAAD</i> (in bytes).
<i>pSrc</i>	Pointer to the input ciphertext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>len</i>	Length of the plaintext and ciphertext data stream (in bytes).
<i>pTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the authentication tag <i>*pTag</i> (in bytes).

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length and verifies the authentication tag in the CCM mode as specified in [NIST SP 800-38C]. Use this function if the entire message is available. Otherwise, follow the typical sequence of invoking the incremental functions (see [AES-CCM Functions](#)).

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>ippStsLengthErr</code>	Indicates an error condition if the length parameters do not meet any of the following conditions: <i>keyLen</i> = <code>IppsRijndaelKey128</code> or <i>keyLen</i> = <code>IppsRijndaelKey192</code> or <i>keyLen</i> = <code>IppsRijndaelKey256</code> $7 \leq ivLen \leq 13$ $4 \leq tagLen \leq 16$ and <i>tagLen</i> is even.

Rijndael128CCMGetSize

Gets the size of the `IppsRijndael128CCMState` context.

Syntax

```
IppStatus ippsRijndael128CCMGetSize(Ipp32u* pSize);
```

Parameters

pSize Pointer to the size of the `IppsRijndael128CCMState` context.

Description

This function is declared in the `ippcp.h` file. The function gets the size of the `IppsRijndael128CCMState` context in bytes and stores it in **pSize*.

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 specified pointer is <code>NULL</code> .

Rijndael128CCMInit

Initializes user-supplied memory as the `IppsRijndael128CCMState` context for future use.

Syntax

```
IppStatus ippsRijndael128CCMInit(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,  
IppsRijndael128CCMState* pState);
```

Parameters

pKey Pointer to the secret key.

keyLength Length of the secret key.

pState Pointer to the `IppsRijndael128CCMState` context.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pState* as the `IppsRijndael128CCMState` context. In addition, the function uses the initialization variable and additional authenticated data to provide all necessary key material for both encryption and decryption.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

`ippStsLengthErr` Indicates an error condition an error condition if `keyLen` is not set to `IppsRijndaelKey128`, `IppsRijndaelKey192` or `IppsRijndaelKey256`.

Rijndael128CCMStart

Starts the process of authenticated encryption/decryption for a new message.

Syntax

```
IppStatus ippsRijndael128CCMStart(const Ipp8u* pIV, Ipp32u ivLen, const Ipp8u* pAAD, Ipp32u aadLen, IppsRijndael128CCMState* pState);
```

Parameters

<code>pIV</code>	Pointer to the initialization vector.
<code>ivLen</code>	Length of the initialization vector <code>*pIV</code> (in bytes).
<code>pAAD</code>	Pointer to the additional authenticated data.
<code>aadLen</code>	Length of additional authenticated data <code>*pAAD</code> (in bytes).
<code>pState</code>	Pointer to the <code>IppsRijndael128CCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function resets internal counters and buffers of the `*pState` context.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>pState</code>	Indicates an error condition if any of the specified pointers is <code>NULL</code> .
<code>pState</code>	Indicates an error condition if the context parameter does not match the operation.
<code>pState</code>	Indicates an error condition if <code>ivLen < 7</code> or <code>ivLen > 13</code> .

Rijndael128CCMEncrypt

Encrypts a data buffer in the CCM mode.

Syntax

```
IppStatus ippsRijndael128CCMEncrypt(const Ipp8u* pSrc, Ipp8u* pDst, Ipp32u len, IppsRijndael128CCMState* pState);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of a variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>len</code>	Length of the plaintext and ciphertext data stream in bytes.

pState Pointer to the `IppsRijndael128CCMState` context.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the CCM mode as specified in [NIST SP 800-38C].

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128CCMDecrypt

Decrypts a data buffer in the CCM mode.

Syntax

```
IppStatus ippRijndael128CCMDecrypt(const Ipp8u* pSrc, Ipp8u* pDst, Ipp32u len,
IppsRijndael128CCMState* pState);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>len</i>	Length of the plaintext and ciphertext data stream in bytes.
<i>pState</i>	Pointer to the <code>IppsRijndael128CCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input ciphered data stream of a variable length in the CCM mode as specified in [NIST SP 800-38C].

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128CCMGetTag

Generates the message authentication tag in the CCM mode.

Syntax

```
IppStatus ippsRijndael128CCMGetTag (Ipp8u* pTag, Ipp32u tagLen, const  
IppsRijndael128CCMState* pState);
```

Parameters

<i>pTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the authentication tag <i>*pTag</i> (in bytes).
<i>pState</i>	Pointer to the <code>IppsRijndael128CCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function generates and computes the authentication tag of length *tagLen* bytes in the CCM mode as specified in [NIST SP 800-38C]. The `ippsRijndael128GCMGetTag` function does not stop the encryption/decryption and authentication process.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>tagLen</i> < 1 or <i>tagLen</i> does not exceed the tag length specified in the previous call to <code>Rijndael128CCMStart</code> .

Rijndael128CCMMessageLen

Sets up the length of the message to be processed.

Syntax

```
IppStatus ippsRijndael128CCMMessageLen(Ipp32u msgLen, IppsRijndael128CCMState* pState);
```

Parameters

<i>msgLen</i>	Length of the message to be processed (in bytes).
<i>pState</i>	Pointer to the <code>IppsRijndael128CCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function assigns the value of *msgLen* to the length of the message to be processed in the **pState* context.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <code>msgLen=0</code> .

Rijndael128CCMTagLen

Sets up the length of the required authentication tag.

Syntax

```
ippStatus ippRijndael128CCMTagLen(Ipp32u tagLen, IppsRijndael128CCMState* pState);
```

Parameters

<code>tagLen</code>	Length of the required authentication tag (in bytes).
<code>pState</code>	Pointer to the <code>IppsRijndael128CCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function assigns the value of `tagLen` to the length of the required authentication tag in the `*pState` context.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 4</code> or <code>tagLen > 16</code> or <code>tagLen</code> is odd.

AES-GCM Functions

The Galois/Counter Mode (GCM) is a mode of operation of the AES algorithm. GCM [NIST SP 800-38D] uses a variation of the Counter mode of operation for encryption. GCM assures authenticity of the confidential data (of up to about 64 GB per invocation) using a universal hash function defined over a binary finite field (the Galois field).

GCM can also provide authentication assurance for additional data (of practically unlimited length per invocation) that is not encrypted. If the GCM input contains only data that is not to be encrypted, the resulting specialization of GCM, called GMAC, is simply an authentication mode for the input data.

GCM provides stronger authentication assurance than a (non-cryptographic) checksum or error detecting code. In particular, GCM can detect both accidental modifications of the data and intentional, unauthorized modifications.

Table "Intel IPP AES-GCM Functions" lists Intel IPP AES-GCM functions:

Intel IPP AES-GCM Functions

Function Base Name	Operation
Rijndael128GCMEncryptMessage	Encrypts an entire message and generates its authentication tag in the GCM mode.
Rijndael128GCMDecryptMessage	Decrypts an entire message and generates its authentication tag in the GCM mode.
Rijndael128GCMGetSize	DEPRECATED. Use Rijndael128GCMGetSizeManaged instead of this function. Gets the size of the <code>IppsRijndael128GCMState</code> context.
Rijndael128GCMGetSizeManaged	Gets the size of the <code>IppsRijndael128GCMState</code> context for use of the AES-GCM implementation that meets specified requirements.
Rijndael128GCMInit	DEPRECATED. Use Rijndael128GCMInitManaged instead of this function. Initializes user-supplied memory as the <code>IppsRijndael128GCMState</code> context for future use.
Rijndael128GCMInitManaged	Initializes user-supplied memory as the <code>IppsRijndael128GCMState</code> context for use of the AES-GCM implementation that meets specified requirements.
Rijndael128GCMStart	Starts the process of authenticated encryption/decryption for a new message.
Rijndael128GCMReset	Resets the <code>IppsRijndael128GCMState</code> context for authenticated encryption/decryption of a new message.
Rijndael128GCMProcessIV	Processes an initial vector of a given length according to the GCM specification.
Rijndael128GCMProcessAAD	Processes additional authentication data of a given length according to the GCM specification.
Rijndael128GCMEncrypt	Encrypts a data buffer in the GCM mode.
Rijndael128GCMDecrypt	Decrypts a data buffer in the GCM mode.
Rijndael128GCMGetTag	Generates the message authentication tag in the GCM mode.

The AES-GCM function set includes:

- Functions for complete message operations, to be used for authenticated encryption/decryption of an entire message, if available: [Rijndael128GCMEncryptMessage](#) and [Rijndael128GCMDecryptMessage](#).
- Incremental functions, to be used if the message is too long to be processed in one step.

The AES-GCM incremental functions enable authenticated encryption/decryption of several messages using one key that the [Rijndael128GCMInit](#) function sets. The application code for conducting a typical AES-GCM authenticated encryption should follow the sequence of operations as outlined below:

1. Get the size required to configure the context `IppsRijndael128GCMState` by calling the function `Rijndael128GCMGetSizeManaged`.
2. Call the system memory-allocation service function to allocate a buffer whose size is not less than the function `Rijndael128GCMGetSize` specifies.
3. Initialize the context `IppsRijndael128GCMState *pCtx` by calling the function `Rijndael128GCMInitManaged` with the allocated buffer and the respective AES key.
4. Call `Rijndael128GCMStart` to start authenticated encryption of the first/next message.
5. Keep calling `Rijndael128GCMEncrypt` until the entire message is processed.
6. Request the authentication tag by calling `Rijndael128GCMGetTag`.
7. Proceed to the next message, if any, that is, go to step 4.
8. Call the system memory free service function to release the buffer allocated for the context `IppsRijndael128GCMState`, if needed.

If the size of the initial vector and/or additional authenticated data (*IV* and *AAD* parameters of the `Rijndael128GCMStart` function, respectively) is large or any of these parameters is placed in a disconnected memory buffer, replace step 4 above with the following sequence:

1. Call `Rijndael128GCMReset` to prepare the `IppsRijndael128GCMState` context for authenticated encryption of the first/new message.
2. Keep calling `Rijndael128GCMProcessIV` for successive parts of *IV* until the entire *IV* is processed.
3. Keep calling `Rijndael128GCMProcessAAD` for successive parts of *AAD* until the entire *AAD* is processed.

Rijndael128GCMEncryptMessage

Encrypts an entire message and generates its authentication tag in the GCM mode.

Syntax

```
IppStatus ippsRijndael128GCMEncryptMessage(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,
const Ipp8u* pIV, Ipp32u ivLen, const Ipp8u* pAAD, Ipp32u aadLen, const Ipp8u* pSrc, Ipp8u*
pDst, Ipp32u len, Ipp8u* pTag, Ipp32u tagLen);
```

Parameters

<i>pKey</i>	Pointer to the secret key.
<i>keyLen</i>	Length of the secret key.
<i>pIV</i>	Pointer to the initialization vector.
<i>ivLen</i>	Length of the initialization vector <i>*pIV</i> (in bytes).
<i>pAAD</i>	Pointer to the additional authenticated data (not to be encrypted).
<i>aadLen</i>	Length of the additional authenticated data <i>*pAAD</i> (in bytes).
<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>len</i>	Length of the plaintext and ciphertext data stream (in bytes).
<i>pTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the authentication tag <i>*pTag</i> (in bytes).

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length and computes the authentication tag according to GCM as specified in [NIST SP 800-38D]. Use this function if the entire message is available. Otherwise, use the incremental functions.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the length parameters do not meet any of the following conditions: $keyLen = \text{IppsRijndaelKey128}$ or $keyLen = \text{IppsRijndaelKey192}$ or $keyLen = \text{IppsRijndaelKey256}$ $ivLen > 0$ $1 \leq tagLen \leq 16$.

Rijndael128GCMDecryptMessage

Decrypts an entire message and generates its authentication tag in the GCM mode.

Syntax

```
IppStatus ippsRijndael128GCMDecryptMessage(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,
const Ipp8u* pIV, Ipp32u ivLen, const Ipp8u* pAAD, Ipp32u aadLen, const Ipp8u* pSrc, Ipp8u*
pDst, Ipp32u len, Ipp8u* pTag, Ipp32u tagLen);
```

Parameters

<code>pKey</code>	Pointer to the secret key
<code>keyLen</code>	Length of the secret key.
<code>pIV</code>	Pointer to the initialization vector.
<code>ivLen</code>	Length of the initialization vector $*pIV$ (in bytes).
<code>pAAD</code>	Pointer to the additional authenticated data (not encrypted).
<code>aadLen</code>	Length of additional authenticated data $*pAAD$ (in bytes).
<code>pSrc</code>	Pointer to the input ciphertext data stream of a variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream.
<code>len</code>	Length of the plaintext and ciphertext data stream (in bytes).
<code>pTag</code>	Pointer to the authentication tag.
<code>tagLen</code>	Length of the authentication tag $*pTag$ (in bytes).

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length and verifies the authentication tag according to GCM, as specified in [NIST SP 800-38D]. Use this function if the entire message is available. Otherwise, use the incremental functions.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the length parameters do not meet any of the following conditions: $keyLen = \text{IppsRijndaelKey128}$ or $keyLen = \text{IppsRijndaelKey192}$ or $keyLen = \text{IppsRijndaelKey256}$ $ivLen > 0$ $1 \leq tagLen \leq 16$.

Rijndael128GCMGetSize

DEPRECATED. Gets the size of the `IppsRijndael128GCMState` context.

Syntax

```
IppStatus ippRijndael128GCMGetSize(Ipp32u* pSize);
```

Parameters

pSize Pointer to the size of the `IppsRijndael128GCMState` context.

Description

This function is deprecated. Use [Rijndael128GCMGetSizeManaged](#) instead of it.

This function is declared in the `ippcp.h` file. The function gets the size of the `IppsRijndael128GCMState` context in bytes and stores it in **pSize*.

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 specified pointer is <code>NULL</code> .

Rijndael128GCMGetSizeManaged

Gets the size of the `IppsRijndael128GCMState` context for use of the AES-GCM implementation with the specified characteristics.

Syntax

```
IppStatus ippRijndael128GCMGetSizeManaged(IppAESGCMbehaviour flag, Ipp32u* pSize);
```

Parameters

flag A flag that defines characteristics of the AES-GCM implementation.

pSize Pointer to the size of the `IppsRijndael128GCMState` context.

Description

This function is declared in the `ippcp.h` file. The function takes characteristics of the AES-GCM implementation from the input parameter `flag`, gets the size of the `IppsRijndael128GCMState` context (in bytes) needed to perform encryption and/or decryption that uses the specified AES-GCM implementation, and stores the size in `*pSize`. The `flag` parameter is the following enumerator:

```
typedef enum {
    ippAESGCMdefault,
    ippAESGCMsafe,
    ippAESGCMtable2K
}
IppAESGCMbehaviour;
```

where the values define the following requirements for the characteristics of the AES-GCM implementation:

<code>ippAESGCMdefault</code>	Minimum memory, that is, the minimum size of the context
<code>ippAESGCMsafe</code>	Maximum protection against timing attacks
<code>ippAESGCMtable2K</code>	Maximum performance of the implementation



NOTE. If your system is based on microprocessors that support an AES instruction set, the AES-GCM implementation will meet all the above requirements regardless of the value of `flag`.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

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 specified pointer is <code>NULL</code> .

Rijndael128GCMInit

DEPRECATED. *Initializes user-supplied memory as the `IppsRijndael128GCMState` context for future use.*

Syntax

```
IppStatus ippRijndael128GCMInit(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,
IppsRijndael128GCMState* pState);
```

Parameters

<i>pKey</i>	Pointer to the secret key.
<i>keyLen</i>	Length of the secret key.
<i>pState</i>	Pointer to the <code>IppsRijndael128GCMState</code> context.

Description

This function is deprecated. Use `Rijndael128GCMInitManaged` instead of it.

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pState* as `IppsRijndael128GCMState` context. In addition, the function uses the initialization variable and additional authenticated data to provide all necessary key material for both encryption and decryption.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>keyLen</i> is not set to <code>IppsRijndaelKey128</code> , <code>IppsRijndaelKey192</code> or <code>IppsRijndaelKey256</code> .

Rijndael128GCMInitManaged

Initializes user-supplied memory as the `IppsRijndael128GCMState` context for use of the AES-GCM implementation with the specified characteristics.

Syntax

```
IppStatus ippRijndael128GCMInitManaged(IppAESGCMbehaviour flag, const Ipp8u* pKey,
IppsRijndaelKeyLength keyLen, IppsRijndael128GCMState* pState);
```

Parameters

<i>flag</i>	A flag that defines characteristics for the AES-GCM implementation.
<i>pKey</i>	Pointer to the secret key.
<i>keyLen</i>	Length of the secret key.
<i>pState</i>	Pointer to the <code>IppsRijndael128GCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function takes characteristics of the AES-GCM implementation from the input parameter `flag` and initializes the memory pointed by `pState` as the `IppsRijndael128GCMState` context needed to perform encryption and/or decryption that uses the specified AES-GCM implementation. In addition, the function uses the initialization variable and additional authenticated data to provide all necessary key material for both encryption and decryption. The `flag` parameter is the following enumerator:

```
typedef enum {
    ippAESGCMdefault,
    ippAESGCMsafe,
    ippAESGCMtable2K
}
IppAESGCMbehaviour;
```

where the values define the following requirements for the characteristics of the AES-GCM implementation:

<code>ippAESGCMdefault</code>	Minimum memory, that is, the minimum size of the context
<code>ippAESGCMsafe</code>	Maximum protection against timing attacks
<code>ippAESGCMtable2K</code>	Maximum performance of the implementation

Call `Rijndael128GCMInitManaged` with the same value of `flag` as used in the previous call to `Rijndael128GCMGetSizeManaged`.



NOTE. If your system is based on microprocessors that support an AES instruction set, the AES-GCM implementation will meet all the above requirements regardless of the value of `flag`.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keyLen</code> is not set to <code>IppsRijndaelKey128</code> , <code>IppsRijndaelKey192</code> or <code>IppsRijndaelKey256</code> .

See Also

Rijndael128GCMStart

Starts the process of authenticated encryption/decryption for new message.

Syntax

```
IppStatus ippsRijndael128GCMStart(const Ipp8u* pIV, Ipp32u ivLen, const Ipp8u* pAAD, Ipp32u aadLen, IppsRijndael128GCMState* pState);
```

Parameters

<i>pIV</i>	Pointer to the initialization vector.
<i>ivLen</i>	Length of the initialization vector <i>*pIV</i> (in bytes).
<i>pAAD</i>	Pointer to the additional authenticated data.
<i>aadLen</i>	Length of additional authenticated data <i>*pAAD</i> (in bytes).
<i>pState</i>	Pointer to the <code>IppsRijndael128GCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function resets internal counters and buffers of the **pState* context.

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 parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the length of the initialization vector is zero.

Rijndael128GCMReset

Resets the `IppsRijndael128GCMState` context for authenticated encryption/decryption of a new message.

Syntax

```
IppStatus ippsRijndael128GCMReset(IppsRijndael128GCMState* pState);
```

Parameters

<i>pState</i>	Pointer to the <code>IppsRijndael128GCMState</code> context.
---------------	--

Description

This function is declared in the `ippcp.h` file. The function resets the `*pState` context to prepare it for either of the following operations with a new message:

- encryption and tag generation
- decryption and tag authentication

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 parameter does not match the operation.

Rijndael128GCMProcessIV

Processes an initial vector of a given length according to the GCM specification.

Syntax

```
IppStatus ippRijndael128GCMProcessIV(const Ipp8u* pIV, Ipp32u ivLen,  
IppsRijndael128GCMState* pState);
```

Parameters

<code>pIV</code>	Pointer to the initialization vector.
<code>ivLen</code>	Length of the initialization vector <code>*pIV</code> (in bytes).
<code>pState</code>	Pointer to the <code>IppsRijndael128GCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function processes `ivLen` bytes of the initial vector `*pIV` as specified in [NIST SP 800-38D].

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 parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the length of the initialization vector is zero.

Rijndael128GCMProcessAAD

Processes additional authenticated data of a given length according to the GCM specification.

Syntax

```
IppStatus ippsRijndael128GCMProcessAAD(const Ipp8u* pAAD, Ipp32u aadLen,  
IppsRijndael128GCMState* pState);
```

Parameters

<i>pAAD</i>	Pointer to the additional authenticated data.
<i>aadLen</i>	Length of additional authenticated data <i>*pAAD</i> (in bytes).
<i>pState</i>	Pointer to the <code>IppsRijndael128GCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function processes *aadLen* bytes of additional authenticated data **pAAD* as specified in [NIST SP 800-38D].

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 parameter does not match the operation.

Rijndael128GCMEncrypt

Encrypts a data buffer in the GCM mode.

Syntax

```
IppStatus ippsRijndael128GCMEncrypt(const Ipp8u* pSrc, Ipp8u* pDst, Ipp32u len,  
IppsRijndael128GCMState* pState);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>len</i>	Length of the plaintext and ciphertext data stream in bytes.
<i>pState</i>	Pointer to the <code>IppsRijndael128GCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to GCM as specified in [NIST SP 800-38D].

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128GCMDecrypt

Decrypts a data buffer in the GCM mode.

Syntax

```
IppStatus ippRijndael128GCMDecrypt(const Ipp8u* pSrc, Ipp8u* pDst, Ipp32u len,
IppRijndael128GCMState* pState);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream of a variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream.
<code>len</code>	Length of the plaintext and ciphertext data stream in bytes.
<code>pState</code>	Pointer to the <code>IppRijndael128GCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input cipher data stream of a variable length according to GCM as specified in [NIST SP 800-38D].

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Rijndael128GCMGetTag

Generates the authentication tag in the GCM mode.

Syntax

```
IppStatus ippRijndael128GCMGetTag (Ipp8u* pTag, Ipp32u tagLen, const
IppRijndael128GCMState* pState);
```

Parameters

<code>pTag</code>	Pointer to the authentication tag.
<code>tagLen</code>	Length of the authentication tag *pTag (in bytes).
<code>pState</code>	Pointer to the <code>IppRijndael128GCMState</code> context.

Description

This function is declared in the `ippcp.h` file. The function generates and computes the authentication tag of length `tagLen` according to GCM as specified in [NIST SP 800-38D]. A call to `ippsRijndael128GCMGetTag` does not stop the process of authenticated encryption/decryption.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen > 16</code> .

Blowfish Functions

Blowfish is a 16-round Feistel block cipher. Under this algorithm, the block size is 64 bits and the key can be of any size up to 448 bits. Although the algorithm requires a computationally intensive key expansion process that creates a set of eighteen 32-bit subkeys plus four 8x32-bit S-boxes derived from the input key, for a total of 4168 bytes, the actual encryption of streaming data is very efficient for software implementation.

This section describes the functions performing various operational modes under the Blowfish cipher systems. The implementation of the functions described in this section complies with the Blowfish cipher schemes.

Table “Intel IPP Blowfish Algorithm Functions” lists Intel IPP Blowfish algorithm functions.

Intel IPP Blowfish Algorithm Functions

Function Base Name	Operation
<code>BlowfishGetSize</code>	Gets the size of the <code>IppsBlowfishSpec</code> context.
<code>BlowfishInit</code>	Initializes user-supplied memory as <code>IppsBlowfishSpec</code> context for future use.
<code>BlowfishPack</code> , <code>BlowfishUnpack</code>	Packs/unpacks the <code>IppsBlowfishSpec</code> context into/from a user-defined buffer.
<code>BlowfishEncryptECB</code>	Encrypts input plaintext according to Blowfish scheme in the ECB mode.
<code>BlowfishDecryptECB</code>	Decrypts byte data stream according to Blowfish scheme in the ECB mode.
<code>BlowfishEncryptCBC</code>	Encrypts byte data stream according to Blowfish scheme in the CBC mode.
<code>BlowfishDecryptCBC</code>	Decrypts byte data stream according to Blowfish scheme in the CBC mode.
<code>BlowfishEncryptCFB</code>	Encrypts byte data stream according to Blowfish scheme in the CFB mode.
<code>BlowfishDecryptCFB</code>	Decrypts byte data stream according to Blowfish scheme in the CFB mode.
<code>BlowfishEncryptOFB</code>	Encrypts byte data stream according to Blowfish scheme in the OFB mode.
<code>BlowfishDecryptOFB</code>	Decrypts byte data stream according to Blowfish scheme in the OFB mode.
<code>BlowfishEncryptCTR</code>	Encrypts a variable length data stream in the CTR mode.
<code>BlowfishDecryptCTR</code>	Decrypts a variable length data stream in the CTR mode.

Throughout this section, the functions for Blowfish baseline cipher scheme employ the context `IppsBlowfishSpec`. It serves as an operational vehicles to carry not only both a set of subkeys and a set of S-Boxes, but also the key management information.

Once the respective initialization function has generated a set of subkeys and S-Boxes, the functions for ECB, CBC, CFB, and CTR modes are ready for the execution of either encrypting or decrypting the streaming data with the selected padding scheme.

The application code for conducting a typical encryption under CBC mode using Blowfish scheme should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the context `IppsBlowfishSpec` by calling the function `BlowfishGetSize`.
2. Call the operating system memory allocation service function to allocate a buffer whose size is no less than the one specified by the function `BlowfishGetSize`.
3. Initialize the context `IppsBlowfishSpec * pCtx` by calling the function `BlowfishInit` with the allocated buffer and the respective Blowfish cipher key of the specified size.
4. Specify the initialization vector and the padding scheme, then call the function `BlowfishEncryptCBC` to encrypt the input data stream using the Blowfish encryption function with CBC mode.
5. Call the operating system memory free service function to release the buffer allocated for the context `IppsBlowfishSpec`, if needed.

The `IppsBlowfishSpec` context is position-dependent. The `BlowfishPack/BlowfishUnpack` functions transform the position-dependent context to a position-independent form and vice versa.

BlowfishGetSize

Gets the size of the `IppsBlowfishSpec` context.

Syntax

```
IppStatus ippsBlowfishGetSize(int* pSize);
```

Parameters

pSize Pointer to the `IppsBlowfishSpec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsBlowfishSpec` context size in bytes and stores it in **pSize*.

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 <code>NULL</code> .

BlowfishInit

Initializes user-supplied memory as the `IppsBlowfishSpec` context for future use.

Syntax

```
IppStatus ippsBlowfishInit(const Ipp8u *pKey, int keylen, IppsBlowfishSpec* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the Blowfish key.
<i>keylen</i>	Key byte stream length in bytes.

pCtx Pointer to the `IppsBlowfishSpec` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsBlowfishSpec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>keylen</i> is less than 1 or greater than 56.

BlowfishPack, BlowfishUnpack

Packs/unpacks the `IppsBlowfishSpec` context into/from a user-defined buffer.

Syntax

```
IppStatus ippsblofishPack (const IppsBlowfishSpec* pCtx, Ipp8u* pBuffer);
IppStatus ippsblofishUnpack (const Ipp8u* pBuffer, IppsBlowfishSpec* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the <code>IppsBlowfishSpec</code> context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `BlowfishPack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `BlowfishUnpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal `IppsBlowfishSpec` context. The `BlowfishPack` and `BlowfishUnpack` functions enable replacing the position-dependent `IppsBlowfishSpec` context in the memory.

Call the `BlowfishGetSize` function prior to `BlowfishPack/BlowfishUnpack` to determine the size of the buffer.

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 <code>NULL</code> .

BlowfishEncryptECB

Encrypts input plaintext in the ECB mode.

Syntax

```
IppStatus ippsBlowfishEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const IppsBlowfishSpec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the input data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsBlowfishSpec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptECB

Decrypts byte data stream according to Blowfish scheme in the ECB mode.

Syntax

```
IppStatus ippsBlowfishDecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsBlowfishSpec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream of variable length.
<i>srclen</i>	Length of the ciphertext data stream in bytes.

pCtx Pointer to the `IppsBlowfishSpec` context.
padding `IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

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`.
`ippStsLengthErr` Indicates an error condition if the output data stream length is less than or equal to zero.
`ippStsContextMatchErr` Indicates an error condition if the context parameter does not match the operation.
`ippStsUnderRunErr` Indicates an error condition if *srclen* is not divisible by cipher block size.

BlowfishEncryptCBC

Encrypts byte data stream according to Blowfish scheme in the CBC mode.

Syntax

```
IpStatus ippBlowfishEncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const
IppsBlowfishSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

pSrc Pointer to the input plaintext data stream of variable length.
pDst Pointer to the resulting ciphertext data stream.
srclen Length of the plaintext data stream in bytes.
pCtx Pointer to the `IppsBlowfishSpec` context.
pIV Pointer to the initialization vector for the CBC mode operation.
padding `IppsCPPaddingNONE` padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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`.

<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptCBC

Decrypts byte data stream according to Blowfish scheme in the CBC mode.

Syntax

```
IppStatus ippBlowfishDecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsBlowfishSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of the variable length.
<code>srclen</code>	Length of the ciphertext data stream length in bytes.
<code>pCtx</code>	Pointer to the <code>IppsBlowfishSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for CBC mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by cipher block size.

BlowfishEncryptCFB

Encrypts byte data stream according to Blowfish scheme in the CFB mode.

Syntax

```
IppStatus ippsBlowfishEncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int cfbBlkSize,
const IppsBlowfishSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <i>IppsBlowfishSpec</i> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<i>IppsCPPaddingNONE</i> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<i>ippStsLengthErr</i>	Indicates an error condition if the input data stream length is less than or equal to zero.
<i>ippStsUnderRunErr</i>	Indicates an error condition if <i>srcLen</i> is not divisible by <i>cfbBlkSize</i> parameter value.
<i>ippStsCFBSizeErr</i>	Indicates an error condition if the value for <i>cfbBlkSize</i> is illegal.
<i>ippStsContextMatchErr</i>	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptCFB

Decrypts byte data stream according to Blowfish scheme in the CFB mode.

Syntax

```
IppStatus ippsBlowfishDecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int cfbBlkSize,
const IppsBlowfishSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of variable length.
<code>srcLen</code>	Length of the ciphertext data stream in bytes.
<code>cfbBlkSize</code>	Size of the CFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsBlowfishSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the CFB mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by cipher block size.

BlowfishEncryptOFB

Encrypts a variable length data stream according to Blowfish scheme in the OFB mode.

Syntax

```
IppStatus ippBlowfishEncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int ofbBlkSize, const IppsBlowfishSpec* pCtx, Ipp8u* pIV);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>srcLen</code>	Length of the plaintext data stream in bytes.
<code>ofbBlkSize</code>	Size of the OFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsBlowfishSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by the <code>ofbBlkSize</code> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <code>ofbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptOFB

Decrypts a variable length data stream according to Blowfish scheme in the OFB mode.

Syntax

```
ippStatus ippSBlowfishDecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int
ofbBlkSize, const IppsBlowfishSpec* pCtx, Ipp8u* pIV);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream of variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream.
<code>srclen</code>	Length of the ciphertext data stream in bytes.
<code>ofbBlkSize</code>	Size of the OFB block in bytes.
<code>pCtx</code>	Pointer to the <i>IppsBlowfishSpec</i> context.
<code>pIV</code>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.

<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

BlowfishEncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippBlowfishEncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const  
IppsBlowfishSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsBlowfishSpec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <i>ctrNumBitSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

BlowfishDecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippSBlowfishDecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const  
IppsBlowfishSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of a variable length.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsBlowfishSpec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Example of Using Blowfish Functions

Blowfish Encryption and Decryption

```
// use of the CBC mode

void BF_sample(void){
    // size of Blowfish algorithm block is equal to 8
    const int bfBlkSize = 8;

    // get the size of the context needed for the encryption/decryption operation
    int ctxSize;
    ippsBlowfishGetSize(&ctxSize);

    // and allocate one
    IppsBlowfishSpec* pCtx = (IppsBlowfishSpec*)( new Ipp8u [ctxSize] );

    // define the key
    // note that the key length may vary from 1 to 56 bytes
    Ipp8u key[] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                   0x08,0x09,0x10,0x11,0x12,0x13,0x14,0x15,
                   0x16,0x17};

    // and prepare the context for Blowfish usage
    ippsBlowfishInit(key,sizeof(key), pCtx);

    // define the message to be encrypted
    Ipp8u ptext[] = {"quick brown fox jum over lazy dog"};

    // define an initial vector
    Ipp8u iv[bfBlkSize] = {0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};

    // allocate enough memory for the ciphertext
    // note that
    // the size of the ciphertext is always multiple of the cipher block size
    Ipp8u ctext[(sizeof(ptext)+bfBlkSize-1) &~(bfBlkSize-1)];

    // encrypt (CBC mode) ptext message
    // pay attention to the 'length' parameter
```

```

// it defines the number of bytes to be encrypted
ippsBlowfishEncryptCBC(pctx, ctext, sizeof(pctx),
                        pCtx,
                        iv,
                        IppsCPPaddingPKCS7);
// allocate memory for the decrypted message
Ipp8u rtext[sizeof(pctx)];
// decrypt (CBC mode) ctext message
// pay attention to the 'length' parameter
// it defines the number of bytes to be decrypted
ippsBlowfishDecryptCBC(ctext, rtext, sizeof(pctx),
                       pCtx,
                       iv,
                       IppsCPPaddingPKCS7);

delete (Ipp8u*)pCtx;
}

```

Twofish Functions

Twofish is a 16-round Feistel block cipher. The block size is 128 bits and the key can be any size up to 256 bits. The algorithm is one of the five Advanced Encryption Standard (AES) finalists. The cipher design for both the round function and key schedule enables an efficient implementation in software. Even though Twofish is free and not patented, it is nevertheless efficient and highly secure block cipher.

This section describes the functions for various operational modes under the Twofish cipher systems. The functions in this section are implemented to comply to the Twofish cipher schemes documented and submitted to NIST for candidacy of AES by B. Schneier.

Table “Intel IPP Twofish Algorithm Functions” lists Intel IPP Twofish algorithm functions:

Intel IPP Twofish Algorithm Functions

Function Base Name	Operation
TwofishGetSize	Gets the size of the <code>IppsTwofishSpec</code> context.
TwofishInit	Initializes user-supplied memory as <code>IppsTwofishSpec</code> context for future use.
TwofishPack, TwofishUnpack	Packs/unpacks the <code>IppsTwofishSpec</code> context into/from a user-defined buffer.
TwofishEncryptECB	Encrypts input plaintext according to Twofish scheme in the ECB mode.
TwofishDecryptECB	Decrypts byte data stream according to Twofish scheme in the ECB mode.
TwofishEncryptCBC	Encrypts byte data stream according to Twofish scheme in the CBC mode.
TwofishDecryptCBC	Decrypts byte data stream according to Twofish scheme in the CBC mode.
TwofishEncryptCFB	Encrypts byte data stream according to Twofish scheme in the CFB mode.
TwofishDecryptCFB	Decrypts byte data stream according to Twofish scheme in the CFB mode.

Function Base Name	Operation
TwofishEncryptOFB	Encrypts byte data stream according to Twofish scheme in the OFB mode.
TwofishDecryptOFB	Decrypts byte data stream according to Twofish scheme in the OFB mode.
TwofishEncryptCTR	Encrypts a variable length data stream in the CTR mode.
TwofishDecryptCTR	Decrypts a variable length data stream in the CTR mode.

Throughout this section, the functions for Twofish baseline cipher scheme employ the context `IppsTwofishSpec`. This structure serves as an operational vehicle to carry not only both a set of subkeys and a set of S-Boxes, but also the key management information.

Once the respective initialization function has generated a set of subkeys and S-Boxes, the functions for ECB, CBC, CFB, and CTR modes are ready to the execution of either encrypting or decrypting the streaming data with the selected padding scheme.

The application code for conducting typical encryption under the CBC mode using the Twofish scheme should follow the sequence of operations as outlined below:

1. Get the buffer size required to configure the context `IppsTwofishSpec` by calling the function [TwofishGetSize](#).
2. Call operating system memory allocation service function to allocate a buffer whose size is not less than the one specified by the function [TwofishGetSize](#).
3. Initialize the context `IppsTwofishSpec * pCtx` by calling the function [TwofishInit](#) with the allocated buffer and the respective Twofish cipher key of the specified size.
4. Specify the initialization vector and the padding scheme, then call the function [TwofishEncryptCBC](#) to encrypt the input data stream using the Twofish encryption function with CBC mode.
5. Call the operating system memory free service function to release the buffer allocated for the context `IppsTwofishSpec`, if needed.

The `IppsTwofishSpec` context is position-dependent. The [TwofishPack](#)/[TwofishUnpack](#) functions transform the position-dependent context to a position-independent form and vice versa.

TwofishGetSize

Gets the size of the `IppsTwofishSpec` context.

Syntax

```
IppStatus ippsTwofishGetSize(int* pSize);
```

Parameters

pSize Pointer to the `IppsTwofishSpec` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsTwofishSpec` context size in bytes and stores it in **pSize*.

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`.

TwofishInit

Initializes user-supplied memory as the IppsTwofishSpec context for future use.

Syntax

```
IppStatus ippsTwofishInit(const Ipp8u *pKey, int keylen, IppsTwofishSpec* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the Twofish key.
<i>keylen</i>	Key byte stream length in bytes.
<i>pCtx</i>	Pointer to the IppsTwofishSpec context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the IppsTwofishSpec context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>keylen</i> is less than 1 or greater than 32.

TwofishPack, TwofishUnpack

Packs/unpacks the IppsTwofishSpec context into/from a user-defined buffer.

Syntax

```
IppStatus ippsTwofishPack (const IppsTwofishSpec* pCtx, Ipp8u* pBuffer);  
IppStatus ippsTwofishUnpack (const Ipp8u* pBuffer, IppsTwofishSpec* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the IppsTwofishSpec context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `TwofishPack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `TwofishUnpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal IppsTwofishSpec context. The `TwofishPack` and `TwofishUnpack` functions enable replacing the position-dependent IppsTwofishSpec context in the memory.

Call the [TwofishGetSize](#) function prior to `TwofishPack`/`TwofishUnpack` to determine the size of the buffer.

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 <code>NULL</code> .

TwofishEncryptECB

Encrypts input plaintext in the ECB mode.

Syntax

```
IppStatus ippstTwofishEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const  
IppstTwofishSpec* pCtx, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>srclen</code>	Length of the input data stream in bytes.
<code>pCtx</code>	Pointer to the <code>IppstTwofishSpec</code> context.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the cipher scheme specified in [\[NIST SP 800-38A\]](#).

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptECB

Decrypts byte data stream according to Twofish scheme in the ECB mode.

Syntax

```
IppStatus ippstTwofishDecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const  
IppstTwofishSpec* pCtx, IppsCPPadding padding);
```


Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream of variable length.
<i>srclen</i>	Length of the ciphertext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsTwofishSpec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ECB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srclen</i> is not divisible by cipher block size.

TwofishEncryptCBC

Encrypts byte data stream according to Twofish scheme in the CBC mode.

Syntax

```
IppStatus ippstTwofishEncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const
IppsTwofishSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srclen</i>	Length of the plaintext data stream length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsTwofishSpec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CBC mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptCBC

Decrypts byte data stream according to Twofish scheme in the CBC mode.

Syntax

```
IppStatus ippStsTwofishDecryptCBC(const Ipp8u* pSrc, Ipp8u *pDst, int srclen, const IppsTwofishSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of the variable length.
<code>srclen</code>	Length of the ciphertext data stream length in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDESSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for CBC mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CBC mode as specified in [[NIST SP 800-38A](#)].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by cipher block size.

TwofishEncryptCFB

Encrypts byte data stream according to Twofish scheme in the CFB mode.

Syntax

```
IppStatus ippsTwofishEncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int cfbBlkSize,
const IppsTwofishSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <i>IppsTwofishSpec</i> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<i>IppsCPPaddingNONE</i> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<i>ippStsLengthErr</i>	Indicates an error condition if the input data stream length is less than or equal to zero.
<i>ippStsUnderRunErr</i>	Indicates an error condition if <i>srcLen</i> is not divisible by <i>cfbBlkSize</i> parameter value.
<i>ippStsCFBSizeErr</i>	Indicates an error condition if the value for <i>cfbBlkSize</i> is illegal.
<i>ippStsContextMatchErr</i>	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptCFB

Decrypts byte data stream according to Twofish scheme in the CFB mode.

Syntax

```
IppStatus ippsTwofishDecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int cfbBlkSize,
const IppsTwofishSpec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream.
<code>pDst</code>	Pointer to the resulting plaintext data stream of variable length.
<code>srcLen</code>	Length of the ciphertext data stream in bytes.
<code>cfbBlkSize</code>	Size of the CFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsTwofishSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the CFB mode operation.
<code>padding</code>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value for <code>cfbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srcLen</code> is not divisible by cipher block size.

TwofishEncryptOFB

Encrypts a variable length data stream according to Twofish scheme in the OFB mode.

Syntax

```
IppStatus ippstTwofishEncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int ofbBlkSize, const IppsTwofishSpec* pCtx, Ipp8u* pIV);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>srcLen</code>	Length of the plaintext data stream in bytes.
<code>ofbBlkSize</code>	Size of the OFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsTwofishSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>srclen</code> is not divisible by the <code>ofbBlkSize</code> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <code>ofbBlkSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptOFB

Decrypts a variable length data stream according to Twofish scheme in the OFB mode.

Syntax

```
ippStatus ippStsTwofishDecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize,
const IppsTwofishSpec* pCtx, Ipp8u* pIV);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream of variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream.
<code>srclen</code>	Length of the ciphertext data stream in bytes.
<code>ofbBlkSize</code>	Size of the OFB block in bytes.
<code>pCtx</code>	Pointer to the <code>IppsTwofishSpec</code> context.
<code>pIV</code>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.

<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>srcLen</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TwofishEncryptCTR

Encrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippStsTwofishEncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const  
IppsTwofishSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsTwofishSpec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length according to the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <i>ctrNumBitSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

TwofishDecryptCTR

Decrypts a variable length data stream in the CTR mode.

Syntax

```
IppStatus ippstTwofishDecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const  
IppsTwofishSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream.
<i>pDst</i>	Pointer to the resulting plaintext data stream of a variable length.
<i>srcLen</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsTwofishSpec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the CTR mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the output data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Example of Using Twofish Functions

Twofish Encryption and Decryption

```
// use of the FCB mode

void TF_sample(void){
    // size of the Twofish algorithm block is equal to 16
    const int tfBlkSize = 16;

    // get the size of the context needed for the encryption/decryption operation
    int ctxSize;
    ippTwofishGetSize(&ctxSize);

    // and allocate one
    IppsTwofishSpec* pCtx = (IppsTwofishSpec*)( new Ipp8u [ctxSize] );

    // define the key
    // note that the key length may vary from 16 to 32 bytes
    Ipp8u key[] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                   0x08,0x09,0x10,0x11,0x12,0x13,0x14,0x15,
                   0x16,0x17};

    // and prepare the context for Twofish usage
    ippTwofishInit(key,sizeof(key), pCtx);

    // define the message to be encrypted
    Ipp8u ptext[] = {"quick brown fox jump over lazy dog"};

    // fcb value (bytes)
    const int cfbBlkSize = 2;

    // define an initial vector
    Ipp8u iv[tfBlkSize] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
                           0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
```



```

// allocate enough memory for the ciphertext
// note that
// the size of the ciphertext is always multiple of the cfbBlkSize size
Ipp8u ctext[(sizeof(ptext)+cfbBlkSize-1) &~(cfbBlkSize-1)];

// encrypt (CFB mode) ptext message
// pay attention to the 'length' parameter
// it defines the number of bytes to be encrypted
ippsTwofishEncryptCFB(ptext, ctext, sizeof(ptext), cfbBlkSize,
                      pCtx,
                      iv,
                      IppsCPPaddingPKCS7);

// allocate memory for the decrypted message
Ipp8u rtext[sizeof(ptext)];

// decrypt (CFB mode) ctext message

// pay attention to the 'length' parameter
// it defines the number of bytes to be decrypted
ippsTwofishDecryptCFB(ctext, rtext, sizeof(ptext), cfbBlkSize,
                      pCtx,
                      iv,
                      IppsCPPaddingPKCS7);

delete (Ipp8u*)pCtx;
}

```

RC5* Functions

RC5, introduced by R.Rivest, is a fast symmetric block cipher that has a variable block size, variable key length, variable number of rounds and heavily uses data-independent rotations. Intel IPP for Cryptography implements functions for RC5 algorithm with 64- and 128-bit block sizes.

This section describes functions for various operational modes of RC5 cipher systems. The functions are implemented to comply with the RC5 cipher schemes documented in [\[RC5\]](#).

Table “Intel IPP RC5 Functions” lists Intel IPP RC5 functions.

Intel IPP RC5 Functions

Function Base Name	Operation
Functions for 64-bit Block Size	
ARCFive64GetSize	Gets the size of the <code>IppsARCFive64Spec</code> context.
ARCFive64Init	Initializes user-supplied memory as the <code>IppsARCFive64Spec</code> context for future use.
ARCFive64Pack , ARCFive64Unpack	Packs/unpacks the <code>ARCFive64Spec</code> context into/from a user-defined buffer.
ARCFive64EncryptECB	Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the ECB mode.
ARCFive64DecryptECB	Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the ECB mode.
ARCFive64EncryptCBC	Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CBC mode.
ARCFive64DecryptCBC	Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CBC mode.
ARCFive64EncryptCFB	Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CFB mode.
ARCFive64DecryptCFB	Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CFB mode.
ARCFive64EncryptOFB	Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the OFB mode.
ARCFive64DecryptOFB	Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the OFB mode.
ARCFive64EncryptCTR	Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CTR mode.
ARCFive64DecryptCTR	Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CTR mode.
Functions for 128-bit Block Size	
ARCFive128GetSize	Gets the size of the <code>IppsARCFive128Spec</code> context.
ARCFive128Init	Initializes user-supplied memory as the <code>IppsARCFive128Spec</code> context for future use.
ARCFive128Pack / ARCFive128Unpack	Packs/unpacks the <code>ARCFive128Spec</code> context into/from a user-defined buffer.
ARCFive128EncryptECB	Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the ECB mode.
ARCFive128DecryptECB	Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the ECB mode.
ARCFive128EncryptCBC	Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CBC mode.
ARCFive128DecryptCBC	Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CBC mode.
ARCFive128EncryptCFB	Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CFB mode.
ARCFive128DecryptCFB	Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CFB mode.
ARCFive128EncryptOFB	Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the OFB mode.
ARCFive128DecryptOFB	Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the OFB mode.
ARCFive128EncryptCTR	Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CTR mode.

Function Base Name	Operation
ARCFive128DecryptCTR	Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CTR mode.

Throughout this section, the functions for the RC5 cipher with 64-bit block size employ the context `IppsARCFive64Spec` and the functions for the RC5 cipher with 128-bit block size employ the context `IppsARCFive128Spec`. These contexts serve as operational vehicles to carry variables needed to accomplish RC5 encryption/decryption, namely the number of rounds and expanded key table `S`.

Once the respective initialization function generates these variables, the functions for ECB, CBC, CFB, and other modes are ready for either encrypting or decrypting the streaming data with the specified padding scheme.

The application code for conducting a typical encryption in the CBC mode using the RC5 cipher with 64-bit block size should follow the sequence of operations outlined below:

1. Get the size required to configure the context `IppsARCFive64Spec` by calling the function [ARCFive64GetSize](#).
2. Allocate a buffer whose size is not less than the function `ARCFive64GetSize` returns by calling the memory allocation service function of the operating system.
3. Initialize the context `IppsARCFive64Spec *pCtx` by calling the function [ARCFive64Init](#) with the allocated buffer and the respective RC5 cipher key of the specified size.
4. Specify the initialization vector and the padding scheme and then call the function [ARCFive64EncryptCBC](#) to encrypt the input data stream using the RC5 cipher in the CBC mode.
5. Release the memory allocated to the buffer by calling the respective operating system function.



NOTE. Similar procedure can be applied for ECB, CFB, OFB, and CTR modes of operation as well as the RC5 cipher with 128-bit block size.

The `ARCFive64Spec` and `ARCFive128Spec` contexts are position-dependent. The [ARCFive64Pack/ARCFive64Unpack](#) and [ARCFive128Pack/ARCFive128Unpack](#) functions transform the respective position-dependent context to a position-independent form and vice versa.

RC5* Algorithm Functions for 64-bit Block Size

ARCFive64GetSize

Gets the size of the `IppsARCFive64Spec` context.

Syntax

```
IppStatus ippsARCFive64GetSize (int rounds, int* pSize);
```

Parameters

rounds The number of rounds for the cipher.

pSize Pointer to the size value of the `IppsARCFive64Spec` context.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsARCFive64Spec` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

ARCFive64Init

Initializes user-supplied memory as the `IppsARCFive64Spec` context for future use.

Syntax

```
IppStatus ippARCFive64Init (const Ipp8u *pKey, int keylen, int rounds, IppsARCFive64Spec* pCtx);
```

Parameters

<code>pKey</code>	Pointer to the RC5 key.
<code>keylen</code>	Key byte stream length in bytes.
<code>rounds</code>	The number of rounds for the cipher.
<code>pCtx</code>	Pointer to the <code>IppsARCFive64Spec</code> context.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed to by `pCtx` as the `IppsARCFive64Spec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keylen</code> is less than 1 or greater than 256.

ARCFive64Pack, ARCFive64Unpack

Packs/unpacks the `IppsARCFive64Spec` context into/from a user-defined buffer.

Syntax

```
IppStatus ippARCFive64Pack (const IppsARCFive64Spec* pCtx, Ipp8u* pBuffer);  
IppStatus ippARCFive64Unpack (const Ipp8u* pBuffer, IppsARCFive64Spec* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsARCFive64Spec</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `ARCFive64Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the `*pBuffer` buffer. The `ARCFive64Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsARCFive64Spec` context. The `ARCFive64Pack` and `ARCFive64Unpack` functions enable replacing the position-dependent `IppsARCFive64Spec` context in the memory.

Call the [ARCFive64GetSize](#) function prior to `ARCFive64Pack/ARCFive64Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

ARCFive64EncryptECB

Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the ECB mode.

Syntax

```
IppStatus ipp5ARCFive64EncryptECB (const Ipp8u* pSrc, Ipp8u* pDst, int length, const IppsARCFive64Spec* pCtx, IppsCPPPadding padding);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>length</code>	Length of the input data stream in bytes.
<code>pCtx</code>	Pointer to the <code>IppsARCFive64Spec</code> context.
<code>padding</code>	<code>IppsCPPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the ECB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <code>length</code> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64DecryptECB

Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the ECB mode.

Syntax

```
IppStatus ippsARCFive64DecryptECB (const Ipp8u* pSrc, Ipp8u* pDst, int length, const IppsARCFive64Spec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive64Spec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the ECB mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64EncryptCBC

Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CBC mode.

Syntax

```
IppStatus ippsARCFive64EncryptCBC (const Ipp8u* pSrc, Ipp8u* pDst, int length, const IppsARCFive64Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>length</i>	Length of the plaintext data stream in bytes.

<i>pCtx</i>	Pointer to the <code>IppsARCFive64Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the CBC mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64DecryptCBC

Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CBC mode.

Syntax

```
IppStatus ippARCFive64DecryptCBC (const Ipp8u* pSrc, Ipp8u* pDst, int length, const IppsARCFive64Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive64Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the CBC mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .

<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64EncryptCFB

Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CFB mode.

Syntax

```
IppStatus ipp5ARCFive64EncryptCFB (const Ipp8u* pSrc, Ipp8u* pDst, int length, int cfbBlkSize, const IppsARCFive64Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>length</i>	Length of the plaintext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive64Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by the <i>cfbBlkSize</i> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value of <i>cfbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64DecryptCFB

Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CFB mode.

Syntax

```
IppStatus ippsARCFive64DecryptCFB (const Ipp8u* pSrc, Ipp8u* pDst, int length, int
cfbBlkSize, const IppsARCFive64Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive64Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by the <i>cfbBlkSize</i> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value of <i>cfbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64EncryptOFB

Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the OFB mode.

Syntax

```
IppStatus ippsARCFive64EncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int length, int
ofbBlkSize, const IppsARCFive64Spec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>length</i>	Length of the plaintext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive64Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64DecryptOFB

Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the OFB mode.

Syntax

```
IppStatus ipp5ARCFive64DecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int length, int ofbBlkSize, const IppsARCFive64Spec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive64Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64EncryptCTR

Encrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CTR mode.

Syntax

```
IppStatus ippSARCFive64EncryptCTR (const Ipp8u* pSrc, Ipp8u* pDst, int length, const
IppsARCFive64Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>length</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive64Spec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.

<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive64DecryptCTR

Decrypts a variable length data stream using the RC5 cipher with 64-bit block size in the CTR mode.

Syntax

```
IppStatus ipp5ARCFive64DecryptCTR (const Ipp8u* pSrc, Ipp8u* pDst, int length, const IppsARCFive64Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream of variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream.
<code>length</code>	Length of the ciphertext data stream in bytes.
<code>pCtx</code>	Pointer to the <code>IppsARCFive64Spec</code> context.
<code>pCtrValue</code>	Pointer to the counter data block.
<code>ctrNumBitSize</code>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

RC5* Algorithm Functions for 128-bit Block Size

ARCFive128GetSize

Gets the size of the IppsARCFive128Spec context.

Syntax

```
IppStatus ippsARCFive128GetSize (int rounds, int* pSize);
```

Parameters

<i>rounds</i>	The number of rounds for the cipher.
<i>pSize</i>	Pointer to the size of the IppsARCFive128Spec context.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsARCFive128Spec` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

ARCFive128Init

Initializes user-supplied memory as the IppsARCFive128Spec context for future use.

Syntax

```
IppStatus ippsARCFive128Init (const Ipp8u *pKey, int keylen, int rounds, IppsARCFive128Spec* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the RC5 key.
<i>keylen</i>	Key byte stream length in bytes.
<i>rounds</i>	The number of rounds for the cipher.
<i>pCtx</i>	Pointer to the IppsARCFive128Spec context.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsARCFive128Spec` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keylen</code> is less than 1 or greater than 256.

ARCFive128Pack, ARCFive128Unpack

Packs/unpacks the `IppsARCFive128Spec` context into/from a user-defined buffer.

Syntax

```
IppStatus ipp5ARCFive128Pack (const IppsARCFive128Spec* pCtx, Ipp8u* pBuffer);  
IppStatus ipp5ARCFive128Unpack (const Ipp8u* pBuffer, IppsARCFive128Spec* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsARCFive128Spec</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `ARCFive128Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the `*pBuffer` buffer. The `ARCFive128Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsARCFive128Spec` context. The `ARCFive128Pack` and `ARCFive128Unpack` functions enable replacing the position-dependent `IppsARCFive128Spec` context in the memory.

Call the [ARCFive128GetSize](#) function prior to `ARCFive128Pack/ARCFive128Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

ARCFive128EncryptECB

Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the ECB mode.

Syntax

```
IppStatus ipp5ARCFive128EncryptECB (const Ipp8u* pSrc, Ipp8u* pDst, int length, const  
IppsARCFive128Spec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>length</i>	Length of the input data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the ECB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128DecryptECB

Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the ECB mode.

Syntax

```
IppStatus ipp5ARCFive128DecryptECB (const Ipp8u* pSrc, Ipp8u* pDst, int length, const IppsARCFive128Spec* pCtx, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the ECB mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128EncryptCBC

Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CBC mode.

Syntax

```
IppStatus ipp5ARCFive128EncryptCBC (const Ipp8u* pSrc, Ipp8u* pDst, int length, const IppsARCFive128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>length</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the CBC mode as specified in the [\[NIST SP 800-38A\]](#).

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128DecryptCBC

Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CBC mode.

Syntax

```
IppStatus ipp5ARCFive128DecryptCBC (const Ipp8u* pSrc, Ipp8u* pDst, int length, const
IppsARCFive128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CBC mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the CBC mode as specified in the [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to 0.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by data block size.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128EncryptCFB

Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CFB mode.

Syntax

```
IppStatus ipp5ARCFive128EncryptCFB (const Ipp8u* pSrc, Ipp8u* pDst, int length, int
cfbBlkSize, const IppsARCFive128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.

<i>length</i>	Length of the plaintext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by the <i>cfbBlkSize</i> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value of <i>cfbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128DecryptCFB

Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CFB mode.

Syntax

```
IppStatus ipp5ARCFive128DecryptCFB (const Ipp8u* pSrc, Ipp8u* pDst, int length, int cfbBlkSize, const IppsARCFive128Spec* pCtx, const Ipp8u* pIV, IppsCPPadding padding);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>cfbBlkSize</i>	Size of the CFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the CFB mode operation.
<i>padding</i>	<code>IppsCPPaddingNONE</code> padding scheme.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the CFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by the <i>cfbBlkSize</i> parameter value.
<code>ippStsCFBSizeErr</code>	Indicates an error condition if the value of <i>cfbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128EncryptOFB

Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the OFB mode.

Syntax

```
IppStatus ippARCFive128EncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int length, int
ofbBlkSize, const IppsARCFive128Spec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of variable length.
<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>length</i>	Length of the plaintext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128DecryptOFB

Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the OFB mode.

Syntax

```
IppStatus ippsARCFive128DecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int length, int ofbBlkSize, const IppsARCFive128Spec* pCtx, Ipp8u* pIV);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>ofbBlkSize</i>	Size of the OFB block in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>pIV</i>	Pointer to the initialization vector for the OFB mode operation.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the OFB mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsUnderRunErr</code>	Indicates an error condition if <i>length</i> is not divisible by the <i>ofbBlkSize</i> parameter value.
<code>ippStsOFBSizeErr</code>	Indicates an error condition if the value of <i>ofbBlkSize</i> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128EncryptCTR

Encrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CTR mode.

Syntax

```
IppStatus ippsARCFive128EncryptCTR (const Ipp8u* pSrc, Ipp8u* pDst, int length, const IppsARCFive128Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input plaintext data stream of a variable length.
-------------	--

<i>pDst</i>	Pointer to the resulting ciphertext data stream.
<i>length</i>	Length of the plaintext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length in the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFive128DecryptCTR

Decrypts a variable length data stream using the RC5 cipher with 128-bit block size in the CTR mode.

Syntax

```
IppStatus ippARCFive128DecryptCTR (const Ipp8u* pSrc, Ipp8u* pDst, int length, const
IppsARCFive128Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Parameters

<i>pSrc</i>	Pointer to the input ciphertext data stream of variable length.
<i>pDst</i>	Pointer to the resulting plaintext data stream.
<i>length</i>	Length of the ciphertext data stream in bytes.
<i>pCtx</i>	Pointer to the <code>IppsARCFive128Spec</code> context.
<i>pCtrValue</i>	Pointer to the counter data block.
<i>ctrNumBitSize</i>	Number of bits in the specific part of the counter to be incremented.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length in the CTR mode as specified in [NIST SP 800-38A].

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than or equal to zero.
<code>ippStsCTRSizeErr</code>	Indicates an error condition if the value of the <code>ctrNumBitSize</code> is illegal.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFour Functions

As the RC4* stream cipher, widely used for file encryption and secure communications, is the property of RSA Security Inc., a cipher discussed in this section and resulting in the same encryption/decryption as RC4* is called ARCFour.

The ARCFour stream cipher ([AC]) uses a variable length key of up to 256 octets (bytes). ARCFour operates in the Output Feedback mode (OFB), defined in [NIST SP 800-38A], which creates the keystream independently of both the plaintext and the ciphertext.

Table “Intel IPP ARCFour Algorithm Functions” lists Intel IPP ARCFour algorithm functions:

Intel IPP ARCFour Algorithm Functions

Function Base Name	Operation
ARCFourGetSize	Gets the size of the <code>IppsARCFourState</code> context.
ARCFourCheckKey	Checks weakness of a user defined key.
ARCFourInit	Initializes user-supplied memory as the <code>IppsARCFourState</code> context for future use.
ARCFourPack , ARCFourUnpack	Packs/unpacks the <code>ARCFourSpec</code> context into/from a user-defined buffer.
ARCFourEncrypt	Encrypts a variable length data stream according to ARCFour.
ARCFourDecrypt	Decrypts a variable length data stream according to ARCFour.
ARCFourReset	Resets the <code>IppsARCFourState</code> context to the initial state.

The ARCFour algorithm functions, described in this section, use the context `IppsARCFourState` as an operational vehicle to carry variables needed to execute the algorithm: S-Boxes and a current pair of indices.

The typical application code for conducting an encryption or decryption using ARCFour should follow the sequence of operations listed below:

1. Get the buffer size required to configure the context `IppsARCFourState` by calling the function [ARCFourGetSize](#).
2. Call the operating system memory allocation service function to allocate a buffer whose size is not less than the one specified by the function [ARCFourGetSize](#).
3. Initialize the pointer `pCtx` to the `IppsARCFourState` context by calling the function [ARCFourInit](#) with the allocated buffer and the respective ARCFour cipher key of the specified size.
4. Call the [ARCFourEncrypt](#) or [ARCFourDecrypt](#) function to encrypt or decrypt the input data stream, respectively.
5. Call the operating system memory free service function to release the buffer allocated for the `IppsARCFourState` context, if needed.

The `ARCFourSpec` context is position-dependent. The `ARCFourPack/ARCFourUnpack` functions transform the position-dependent context to a position-independent form and vice versa.

ARCFourGetSize

Gets the size of the `IppsARCFourState` context.

Syntax

```
IppStatus ippsARCFourGetSize(int* pSize);
```

Parameters

pSize Pointer to the size value of the `IppsARCFourState` context.

Description

This function is declared in the `ippcp.h` file. The function gets the size of the `IppsARCFourState` context in bytes and stores it in **pSize*.

Return Values

`ippStsNoErr` Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr` Indicates an error condition if the specified pointer is `NULL`.

ARCFourCheckKey

Checks weakness of a user-defined key.

Syntax

```
IppStatus ippsARCFourCheckKey(const Ipp8u* pKey, int keyLen, IppsBool* pIsWeak);
```

Parameters

pKey Pointer to the user-defined key.
keyLen Length of the user-defined key in octets.
pIsWeak Pointer to the result of checking.

Description

This function is declared in the `ippcp.h` file. The function checks weakness of user-defined key. The function allows to make sure that the supplied key provides sufficient security.

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`.
`ippStsLengthErr` Indicates an error condition if *keyLen* < 1 or *keyLen* > 256.

ARCFourInit

Initializes user-supplied memory as the IppsARCFourState context for future use.

Syntax

```
IppStatus ippsARCFourInit(const Ipp8u* pKey, int keyLen, IppsARCFourState* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the user-defined key.
<i>keyLen</i>	Length of the user-defined key in octets.
<i>pCtx</i>	Pointer to the IppsARCFourState context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as IppsARCFourState context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>keyLen</i> < 1 or <i>keyLen</i> > 256.

ARCFourPack, ARCFourUnpack

Packs/unpacks the IppsARCFourSpec context into/from a user-defined buffer.

Syntax

```
IppStatus ippsARCFourPack (const IppsARCFourSpec* pCtx, Ipp8u* pBuffer);  
IppStatus ippsARCFourUnpack (const Ipp8u* pBuffer, IppsARCFourSpec* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the IppsARCFourSpec context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `ARCFourPack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `ARCFourUnpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal IppsARCFourSpec context. The `ARCFourPack` and `ARCFourUnpack` functions enable replacing the position-dependent IppsARCFourSpec context in the memory.

Call the [ARCFourGetSize](#) function prior to `ARCFourPack`/`ARCFourUnpack` to determine the size of the buffer.

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 <code>NULL</code> .

ARCFourEncrypt

Encrypts a variable length data stream according to ARCFour.

Syntax

```
IppStatus ippSARCFourEncrypt(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, IppsARCFourState* pCtx);
```

Parameters

<code>pSrc</code>	Pointer to the input plaintext data stream of variable length.
<code>pDst</code>	Pointer to the resulting ciphertext data stream.
<code>srclen</code>	Length of the plaintext data stream in octets.
<code>pCtx</code>	Pointer to the <code>ARCFourState</code> context.

Description

This function is declared in the `ippcp.h` file. The function encrypts the input data stream of a variable length using the ARCFour algorithm.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if length of the input data stream is less than one octet.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFourDecrypt

Decrypts a variable length data stream according to ARCFour.

Syntax

```
IppStatus ippSARCFourDecrypt(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, IppsARCFourState* pCtx);
```

Parameters

<code>pSrc</code>	Pointer to the input ciphertext data stream of variable length.
<code>pDst</code>	Pointer to the resulting plaintext data stream.

<code>srclen</code>	Length of the ciphertext data stream in octets.
<code>pCtx</code>	Pointer to the <code>ARCFourState</code> context.

Description

This function is declared in the `ippcp.h` file. The function decrypts the input data stream of a variable length according to the ARCFour algorithm.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if length of the input data stream is less than one octet.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ARCFourReset

Resets the `IppsARCFourState` context to the initial state.

Syntax

```
IppStatus ippARCFourReset(IppsARCFourState* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsARCFourState</code> context being reset.
-------------------	---

Description

This function is declared in the `ippcp.h` file. The function resets the `IppsARCFourState` context to the state it had immediately after the [ARCFourInit](#) function call. Contrary to `ARCFourInit`, `ARCFourReset` requires no secret key to initialize the S-Box.

Return Values

<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

One-Way Hash Primitives

Hash functions are used in cryptography with digital signatures and for ensuring data integrity.

When used with digital signatures, a publicly available hash function hashes the message and signs the resulting hash value. The party who receives the message can then hash the message and check if the block size is authentic for the given hash value.

Hash functions are also referred to as “message digests” and “one-way encryption functions”. Both terms are appropriate since hash algorithms do not have a key like symmetric and asymmetric algorithms and you can recover neither the length nor the contents of the plaintext message from the ciphertext.

To ensure data integrity, hash functions are used to compute the hash value that corresponds to a particular input. Then, if necessary, you can check if the input data has remained unmodified; you can re-compute the hash value again using the available input and compare it to the original hash value.

The [Hash Functions](#) section of this chapter describes functions that implement the following hash algorithms for streaming messages: MD5 [[RFC 1321](#)] , SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 [[FIPS PUB 180-2](#)]. These algorithms are widely used in enterprise applications nowadays.

Subsequent sections of this chapter describe [Generalized Hash Functions for Non-Streaming Messages](#), which apply hash algorithms to entire (non-streaming) messages, and [Mask Generation Functions](#), whose algorithms are often based on hash computations.

Each of the above algorithms is implemented as a set of primitive functions.

The full list of Intel® Integrated Performance Primitives (Intel® IPP) Hash Primitive Functions is given in [Table “One-way Hash Primitive Functions”](#).

One-way Hash Primitive Functions

Function Base Name	Operation
Hash Functions	
MD5GetSize	Gets the size of the <code>IppsMD5State</code> context.
MD5Init	Initializes user-supplied memory as <code>IppsMD5State</code> context for future use.
MD5Pack , MD5Unpack	Packs/unpacks the <code>IppsMD5State</code> context into/from a user-defined buffer.
MD5Duplicate	Copies one <code>IppsMD5State</code> context to another.
MD5Update	Digests the current input message stream of the specified length.
MD5Final	Completes computation of the MD5 digest value.
MD5GetTag	Computes the current MD5 digest value of the processed part of the message.
SHA1GetSize	Gets the size of the <code>IppsSHA1State</code> context.
SHA1Init	Initializes user-supplied memory as <code>IppsSHA1State</code> context for future use.
SHA1Pack , SHA1Unpack	Packs/unpacks the <code>IppsSHA1State</code> context into/from a user-defined buffer.

Function Base Name	Operation
SHA1Duplicate	Copies one <code>IppsSHA1State</code> context to another.
SHA1Update	Digests the current input message stream of the specified length.
SHA1Final	Completes computation of the SHA-1 digest value.
SHA1GetTag	Computes the current SHA-1 digest value of the processed part of the message.
SHA224GetSize	Gets the size of the <code>IppsSHA224State</code> context.
SHA224Init	Initializes user-supplied memory as <code>IppsSHA224State</code> context for future use.
SHA224Pack, SHA224Unpack	Packs/unpacks the <code>IppsSHA224State</code> context into/from a user-defined buffer.
SHA224Duplicate	Copies one <code>IppsSHA224State</code> context to another.
SHA224Update	Digests the current input message stream of the specified length.
SHA224Final	Completes computation of the SHA-224 digest value.
SHA224GetTag	Computes the current SHA-224 digest value of the processed part of the message.
SHA256GetSize	Gets the size of the <code>IppsSHA256State</code> context.
SHA256Init	Initializes user-supplied memory as <code>IppsSHA256State</code> context for future use.
SHA256Pack, SHA256Unpack	Packs/unpacks the <code>IppsSHA256State</code> context into/from a user-defined buffer.
SHA256Duplicate	Copies one <code>IppsSHA256State</code> context to another.
SHA256Update	Digests the current input message stream of the specified length.
SHA256Final	Completes computation of the SHA-256 digest value.
SHA256GetTag	Computes the current SHA-256 digest value of the processed part of the message.
SHA384GetSize	Gets the size of the <code>IppsSHA384State</code> context.
SHA384Init	Initializes user-supplied memory as <code>IppsSHA384State</code> context for future use.
SHA384Pack, SHA384Unpack	Packs/unpacks the <code>IppsSHA384State</code> context into/from a user-defined buffer.
SHA384Duplicate	Copies one <code>IppsSHA384State</code> context to another.
SHA384Update	Digests the current input message stream of the specified length.
SHA384Final	Completes computation of the SHA-384 digest value.
SHA384GetTag	Computes the current SHA-384 digest value of the processed part of the message.
SHA512GetSize	Gets the size of the <code>IppsSHA512State</code> context.
SHA512Init	Initializes user-supplied memory as <code>IppsSHA512State</code> context for future use.
SHA512Pack, SHA512Unpack	Packs/unpacks the <code>IppsSHA512State</code> context into/from a user-defined buffer.
SHA512Duplicate	Copies one <code>IppsSHA512State</code> context to another.
SHA512Update	Digests the current input message stream of the specified length.
SHA512Final	Completes computation of the SHA-512 digest value.
SHA512GetTag	Computes the current SHA-512 digest value of the processed part of the message.
Generalized Hash Functions for Non-Streaming Messages	
MD5MessageDigest	Computes MD5 digest value of the input message.

Function Base Name	Operation
SHA1MessageDigest	Computes SHA-1 digest value of the input message.
SHA224MessageDigest	Computes SHA-224 digest value of the input message.
SHA256MessageDigest	Computes SHA-256 digest value of the input message.
SHA384MessageDigest	Computes SHA-384 digest value of the input message.
SHA512MessageDigest	Computes SHA-512 digest value of the input message.
Mask Generation Functions	
MGF_MD5	Generates a pseudorandom mask of the specified length using MD5 hash function.
MGF_SHA1	Generates a pseudorandom mask of the specified length using SHA-1 hash function.
MGF_SHA224	Generates a pseudorandom mask of the specified length using SHA-224 hash function.
MGF_SHA256	Generates a pseudorandom mask of the specified length using SHA-256 hash function.
MGF_SHA384	Generates a pseudorandom mask of the specified length using SHA-384 hash function.
MGF_SHA512	Generates a pseudorandom mask of the specified length using SHA-512 hash function.

Hash Functions

Functions featured in this section apply hash algorithms to digesting streaming messages. A primitive implementing a hash algorithm uses the state context (for example, `ippsSHA1State`) as an operational vehicle to carry all necessary variables to manage the computation of the chaining digest value. For example, the primitive implementing the SHA-1 hash algorithm must use the `ippsSHA1State` context.

The function `Init` initializes (`MD5Init`, `SHA1Init`, `SHA224Init`, `SHA256Init`, `SHA384Init`, and `SHA512Init`) the context and sets up specified initialization vectors. Once initialized, the function `Update` (`MD5Update`, `SHA1Update`, `SHA224Update`, `SHA256Update`, `SHA384Update`, and `SHA512Update`) digests the input message stream with the selected hash algorithm till it exhausts all message blocks. The function `Final` (`MD5Final`, `SHA1Final`, `SHA224Final`, `SHA256Final`, `SHA384Final`, and `SHA512Final`) is designed to pad the partial message block into a final message block with the specified padding scheme, and then uses the hash algorithm to transform the final block into a message digest value.

The following example illustrates how the application code can apply the implemented SHA-1 hash standard to digest the input message stream.

1. Call the function [SHA1GetSize](#) to get the size required to configure the `ippsSHA1State` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the [SHA1Init](#) function to set up the initial context state with the SHA-1 specified initialization vectors.
3. Keep calling the function [SHA1Update](#) to digest incoming message stream in the queue till its completion. To determine the current value of the digest, call [SHA1GetTag](#) between the two calls to `SHA1Update`.
4. Call the function [SHA1Final](#) for padding the partial block into a final SHA-1 message block and transforming it into a 160-bit message digest value.
5. Call the operating system memory free service function to release the `ippsSHA1State` context.

The contexts used by the hash primitives (for example, `ippsSHA1State`) are position-dependent. The following functions transform the respective position-dependent context to a position-independent form and vice versa:

- [SHA1Pack/SHA1Unpack](#)

- [SHA224Pack/SHA224Unpack](#)
- [SHA256Pack/SHA256Unpack](#)
- [SHA384Pack/SHA384Unpack](#)
- [SHA512Pack/SHA512Unpack](#)
- [MD5Pack/MD5Unpack](#)

MD5GetSize

Gets the size of the `IppsMD5State` context in bytes.

Syntax

```
IppStatus ippsMD5GetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsMD5State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsMD5State` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

MD5Init

Initializes user-supplied memory as `IppsMD5State` context for future use.

Syntax

```
IppStatus ippsMD5Init(IppsMD5State* pCtx);
```

Parameters

pCtx Pointer to the `IppsMD5State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsMD5State` context.

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 <code>NULL</code> .

MD5Pack, MD5Unpack

Packs/unpacks the IppsMD5State context into/from a user-defined buffer.

Syntax

```
IppStatus ippsMD5Pack (const IppsMD5State* pCtx, Ipp8u* pBuffer);  
IppStatus ippsMD5Unpack (const Ipp8u* pBuffer, IppsMD5State* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the IppsMD5State context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `MD5Pack` function transforms the **pCtx* context to a position-independent form and stores it in the the **pBuffer* buffer. The `MD5Unpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal `IppsMD5State` context. The `MD5Pack` and `MD5Unpack` functions enable replacing the position-dependent `IppsMD5State` context in the memory.

Call the `MD5GetSize` function prior to `MD5Pack/MD5Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

MD5Duplicate

Copies one IppsMD5State context to another.

Syntax

```
IppStatus ippsMD5Duplicate(const IppsMD5State* pSrcCtx, IppsMD5State* pDstCtx);
```

Parameters

<i>pSrcCtx</i>	Pointer to the source IppsMD5State context.
<i>pDstCtx</i>	Pointer to the IppsMD5State context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsMD5State` context to another.

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 <code>NULL</code> .

<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
------------------------------------	---

MD5Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippMD5Update(const Ipp8u *pSrcMesg, int mesglen, IppsMD5State *pCtx);
```

Parameters

<code>pSrcMesg</code>	Pointer to the buffer containing a part of or the whole message.
<code>mesglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsMD5State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

MD5Final

Completes computation of the MD5 digest value.

Syntax

```
IppStatus ippMD5Final(Ipp8u *pMD, IppsMD5State *pCtx);
```

Parameters

<code>pMD</code>	Pointer to the resultant digest value.
<code>pCtx</code>	Pointer to the <code>IppsMD5State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

MD5GetTag

Computes the current MD5 digest value of the processed part of the message.

Syntax

```
IppStatus ippMD5GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsMD5State* pState);
```

Parameters

<code>pDstTag</code>	Pointer to the authentication tag.
<code>tagLen</code>	Length of the tag (in bytes).
<code>pState</code>	Pointer to the <code>IppsMD5State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 180-2] and [RFC 1321]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA1GetSize

Gets the size of the `IppsSHA1State` context in bytes.

Syntax

```
IppStatus ippSHA1GetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsSHA1State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA1State` context size in bytes and stores it in **pSize*.

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`.

SHA1Init

Initializes user-supplied memory as `IppsSHA1State` context for future use.

Syntax

```
IppStatus ippSHA1Init(IppsSHA1State* pCtx);
```

Parameters

pCtx Pointer to the `IppsSHA1State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsSHA1State` context.

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`.

SHA1Pack, SHA1Unpack

Packs/unpacks the `IppsSHA1State` context into/from a user-defined buffer.

Syntax

```
IppStatus ippSHA1Pack (const IppsSHA1State* pCtx, Ipp8u* pBuffer);  
IppStatus ippSHA1Unpack (const Ipp8u* pBuffer, IppsSHA1State* pCtx);
```

Parameters

pCtx Pointer to the `IppsSHA1State` context.
pBuffer Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `SHA1Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the the `*pBuffer` buffer. The `SHA1Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsSHA1State` context. The `SHA1Pack` and `SHA1Unpack` functions enable replacing the position-dependent `IppsSHA1State` context in the memory.

Call the `SHA1GetSize` function prior to `SHA1Pack`/`SHA1Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

SHA1Duplicate

Copies one IppsSHA1State context to another.

Syntax

```
IppStatus ippSHA1Duplicate(const IppsSHA1State* pSrcCtx, IppsSHA1State* pDstCtx);
```

Parameters

<code>pSrcCtx</code>	Pointer to the source <code>IppsSHA1State</code> context.
<code>pDstCtx</code>	Pointer to the <code>IppsSHA1State</code> context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA1State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA1Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippSHA1Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA1State *pCtx);
```

Parameters

<code>pSrcMesg</code>	Pointer to the buffer containing a part of or the whole message.
-----------------------	--

<code>msglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsSHA1State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

SHA1 Final

Completes computation of the SHA-1 digest value.

Syntax

```
IppStatus ippSHA1Final(Ipp8u *pMD, IppsSHA1State *pCtx);
```

Parameters

<code>pMD</code>	Pointer to the resultant digest value.
<code>pCtx</code>	Pointer to the <code>IppsSHA1State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA1GetTag

Computes the current SHA-1 digest value of the processed part of the message.

Syntax

```
IppStatus ippsSHA1GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA1State* pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppsSHA1State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 180-2] and [RFC 1321]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA224GetSize

Gets the size of the `IppsSHA224State` context in bytes.

Syntax

```
IppStatus ippsSHA224GetSize(int *pSize);
```

Parameters

<i>pSize</i>	Pointer to the <code>IppsSHA224State</code> context size value.
--------------	---

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA224State` context size in bytes and stores it in `*pSize`.

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 <code>NULL</code> .

SHA224Init

Initializes user-supplied memory as IppsSHA224State context for future use.

Syntax

```
IppStatus ippsha224init(IppsSHA224State* pCtx);
```

Parameters

pCtx Pointer to the IppsSHA224State context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as IppsSHA224State context.

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 <code>NULL</code> .

SHA224Pack, SHA224Unpack

Packs/unpacks the IppsSHA224State context into/from a user-defined buffer.

Syntax

```
IppStatus ippsha224pack(const IppsSHA224State* pCtx, Ipp8u* pBuffer);  
IppStatus ippsha224unpack(const Ipp8u* pBuffer, IppsSHA224State* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the IppsSHA224State context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `SHA224Pack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `SHA224Unpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal IppsSHA224State context. The `SHA224Pack` and `SHA224Unpack` functions enable replacing the position-dependent IppsSHA224State context in the memory.

Call the [SHA224GetSize](#) function prior to `SHA224Pack`/`SHA224Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

SHA224Duplicate

Copies one IppsSHA224State context to another.

Syntax

```
IppStatus ippsSHA224Duplicate(const IppsSHA224State* pSrcCtx, IppsSHA224State* pDstCtx);
```

Parameters

<i>pSrcCtx</i>	Pointer to the source SHA224State context.
<i>pDstCtx</i>	Pointer to the IppsSHA224State context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA224State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA224Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsSHA224Update(const Ipp8u *pSrcMesg, int msglen, IppsSHA224State *pCtx);
```

Parameters

<i>pSrcMesg</i>	Pointer to the buffer containing a part of or the whole message.
<i>msglen</i>	Length of the actual part of the message in bytes.
<i>pCtx</i>	Pointer to the IppsSHA224State context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

SHA224Final

Completes computation of the SHA-224 digest value.

Syntax

```
IppStatus ippSHA224Final(Ipp8u *pMD, IppsSHA224State *pCtx);
```

Parameters

<code>pMD</code>	Pointer to the resultant digest value.
<code>pCtx</code>	Pointer to the <code>IppsSHA224State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA224GetTag

Computes the current SHA-224 digest value of the processed part of the message.

Syntax

```
IppStatus ippSHA224GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA224State* pState);
```

Parameters

<code>pDstTag</code>	Pointer to the authentication tag.
<code>tagLen</code>	Length of the tag (in bytes).
<code>pState</code>	Pointer to the <code>IppsSHA224State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 180-2] and [RFC 1321]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA256GetSize

Gets the size of the `IppsSHA256State` context in bytes.

Syntax

```
IppStatus ippSHA256GetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsSHA256State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA256State` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

SHA256Init

Initializes user-supplied memory as `IppsSHA256State` context for future use.

Syntax

```
IppStatus ippSHA256Init(IppsSHA256State *pCtx);
```

Parameters

pCtx Pointer to the `IppsSHA256State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as `IppsSHA256State` context.

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 <code>NULL</code> .

SHA256Pack, SHA256Unpack

Packs/unpacks the `IppsSHA256State` context into/from a user-defined buffer.

Syntax

```
IppStatus ippsha256pack (const IppsSHA256State* pCtx, Ipp8u* pBuffer);  
IppStatus ippsha256unpack (const Ipp8u* pBuffer, IppsSHA256State* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsSHA256State</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `SHA256Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the `*pBuffer` buffer. The `SHA256Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsSHA256State` context. The `SHA256Pack` and `SHA256Unpack` functions enable replacing the position-dependent `IppsSHA256State` context in the memory.

Call the [SHA256GetSize](#) function prior to `SHA256Pack`/`SHA256Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

SHA256Duplicate

Copies one `IppsSHA256State` context to another.

Syntax

```
IppStatus ippsha256duplicate (const IppsSHA256State* pSrcCtx, IppsSHA256* pDstCtx);
```

Parameters

<code>pSrcCtx</code>	Pointer to the source <code>IppsSHA256State</code> context.
----------------------	---

pDstCtx Pointer to the `IppsSHA256State` context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA256State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA256Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippSHA256Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA256State *pCtx);
```

Parameters

<i>pSrcMesg</i>	Pointer to the buffer containing a part of or the whole message.
<i>mesglen</i>	Length of the actual part of the message in bytes.
<i>pCtx</i>	Pointer to the <code>IppsSHA256State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

SHA256Final

Completes computation of the SHA-256 digest value.

Syntax

```
IppStatus ippsha256final(Ipp8u *pMD, IppsSHA256State *pCtx);
```

Parameters

<i>pMD</i>	Pointer to the resultant digest value.
<i>pCtx</i>	Pointer to the <code>IppsSHA256State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA256GetTag

Computes the current SHA-256 digest value of the processed part of the message.

Syntax

```
IppStatus ippsha256gettag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA256State* pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppsSHA256State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 180-2] and [RFC 1321]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .

<code>ippStsLengthErr</code>	Indicates an error condition if <i>tagLen</i> < 1 or <i>tagLen</i> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA384GetSize

Gets the size of the `IppsSHA384State` context in bytes.

Syntax

```
IppStatus ippSHA384GetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsSHA384State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA384State` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

SHA384Init

Initializes user-supplied memory as `IppsSHA384State` context for future use.

Syntax

```
IppStatus ippSHA384Init(IppsSHA384State* pCtx);
```

Parameters

pCtx Pointer to the `IppsSHA384State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsSHA384State` context.

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 <code>NULL</code> .

SHA384Pack, SHA384Unpack

Packs/unpacks the IppsSHA384State context into/from a user-defined buffer.

Syntax

```
IppStatus ippsSHA384Pack (const IppsSHA384State* pCtx, Ipp8u* pBuffer);  
IppStatus ippsSHA384Unpack (const Ipp8u* pBuffer, IppsSHA384State* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the IppsSHA384State context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `SHA384Pack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `SHA384Unpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal `IppsSHA384State` context. The `SHA384Pack` and `SHA384Unpack` functions enable replacing the position-dependent `IppsSHA384State` context in the memory.

Call the [SHA384GetSize](#) function prior to `SHA384Pack`/`SHA384Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

SHA384Duplicate

Copies one IppsSHA384State context to another.

Syntax

```
IppStatus ippsSHA384Duplicate(const IppsSHA384State* pSrcCtx, IppsSHA384State* pDstCtx);
```

Parameters

<i>pSrcCtx</i>	Pointer to the source IppsSHA384State context.
<i>pDstCtx</i>	Pointer to the IppsSHA384State context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA384State` context to another.

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 <code>NULL</code> .

`ippStsContextMatchErr` Indicates an error condition if the context parameter does not match the operation.

SHA384Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippSHA384Update(const Ipp8u *pSrcMesg, int msglen, IppsSHA384State *pCtx);
```

Parameters

<code>pSrcMesg</code>	Pointer to the buffer containing a part of or the whole message.
<code>msglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsSHA384State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

SHA384Final

Completes computing of the SHA-384 digest value.

Syntax

```
IppStatus ippSHA384Final( Ipp8u *pMD, IppsSHA384State *pCtx );
```

Parameters

<code>pMD</code>	Pointer to the resultant digest value.
<code>pCtx</code>	Pointer to the <code>IppsSHA384State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA384GetTag

Computes the current SHA-384 digest value of the processed part of the message.

Syntax

```
IppStatus ippSHA384GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA384State* pState);
```

Parameters

<code>pDstTag</code>	Pointer to the authentication tag.
<code>tagLen</code>	Length of the tag (in bytes).
<code>pState</code>	Pointer to the <code>IppsSHA384State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 180-2] and [RFC 1321]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA512GetSize

Gets the size of the `IppsSHA512State` context in bytes.

Syntax

```
IppStatus ippSHA512GetSize(int *pSize);
```


Parameters

pSize Pointer to the `IppsSHA512State` context size value.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsSHA512State` context size in bytes and stores it in **pSize*.

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`.

SHA512Init

Initializes user-supplied memory as `IppsSHA512State` context for future use.

Syntax

```
IppStatus ippsha512init(IppsSHA512State* pState);
```

Parameters

pCtx Pointer to the `IppsSHA512State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as `IppsSHA512State` context.

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`.

SHA512Pack, SHA512Unpack

Packs/unpacks the `IppsSHA512State` context into/from a user-defined buffer.

Syntax

```
IppStatus ippsha512pack (const IppsSHA512State* pCtx, Ipp8u* pBuffer);  
IppStatus ippsha512unpack (const Ipp8u* pBuffer, IppsSHA512State* pCtx);
```

Parameters

pCtx Pointer to the `IppsSHA512State` context.
pBuffer Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `SHA512Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the the `*pBuffer` buffer. The `SHA512Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsSHA512State` context. The `SHA512Pack` and `SHA512Unpack` functions enable replacing the position-dependent `IppsSHA512State` context in the memory.

Call the [SHA512GetSize](#) function prior to `SHA512Pack`/`SHA512Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

SHA512Duplicate

Copies one IppsSHA512State context to another.

Syntax

```
IppStatus ippsha512Duplicate(const IppsSHA512State* pSrcCtx, IppsSHA512* pDstCtx);
```

Parameters

<code>pSrcCtx</code>	Pointer to the source <code>IppsSHA512State</code> context.
<code>pDstCtx</code>	Pointer to the <code>IppsSHA512State</code> context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsSHA512State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA512Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsha512Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA512State *pCtx);
```

Parameters

<code>pSrcMesg</code>	Pointer to the buffer containing a part of or the whole message.
-----------------------	--

<code>mesglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsSHA512State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

SHA512Final

Completes computation of the SHA-512 digest value.

Syntax

```
IppStatus ippSHA512Final(Ipp8u *pMD, IppsSHA512State *pCtx);
```

Parameters

<code>pMD</code>	Pointer to the resultant digest value.
<code>pCtx</code>	Pointer to the <code>IppsSHA512State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

SHA512GetTag

Computes the current SHA-512 digest value of the processed part of the message.

Syntax

```
IppStatus ippsSHA512GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA512State* pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppsSHA512State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 180-2] and [RFC 1321]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Generalized Hash Functions for Non-Streaming Messages

Intel IPP hash functions `MD5MessageDigest`, `SHA1MessageDigest`, `SHA224MessageDigest`, `SHA256MessageDigest`, `SHA384MessageDigest`, and `SHA512MessageDigest` are used to calculate a digest of an entire (non-streaming) input message by applying a selected hash algorithm.

Having the six hash algorithms currently available in the Intel IPP, you may still prefer to use a different implementation of a hash algorithm, based on some other function. In this case you can also use the IPPCP library. To do this, use the definition of a hash function introduced in IPPCP and given in the subsection below. This definition is also applied to a hash function when it is passed as a parameter that some Public Key Cryptography operations optionally use.

General Definition of a Hash Function

Syntax

```
typedef IppStatus(_STDCALL *IppHASH)(const Ipp8u* pMsg, int msgLen, Ipp8u* pMD);
```

Parameters

<i>pMsg</i>	Pointer to the input octet string.
<i>msgLen</i>	Length of the input string in octets.
<i>pMDparamname></i>	Pointer to the output message digest.

Description

This declaration is included in the `ippcp.h` file. The function calculates the digest of a non-streaming message using the implemented hash algorithm.



NOTE. Definition of a hash function used in Intel IPP limits length (in octets) of an input message for any specific hash function by the range of the `int` data type, with the upper bound of $2^{32}-1$.

MD5MessageDigest

Computes MD5 digest value of the input message.

Syntax

```
IppStatus ippMD5MessageDigest(const Ipp8u *pSrcMsg, int msgLen, Ipp8u *pMD);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in octets.
<i>pMD</i>	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute digest value of the entire (non-streaming) input message.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

Example

The code example below shows MD5 digest of a message.

```
void MD5_sample(void){
    // define message
    Ipp8u msg[] = "abcdefghijklmnopqrstuvwxyz";

    // once the whole message is placed into memory,
    // one can use the integrated primitive
    Ipp8u digest[16];
    ippsMD5MessageDigest(msg, strlen((char*)msg), digest);
}
```

SHA1MessageDigest

Computes SHA-1 digest value of the input message.

Syntax

```
IppStatus ippsSHA1MessageDigest(const Ipp8u *pSrcMesg, int mesglen, Ipp8u *pMD);
```

Parameters

<i>pSrcMesg</i>	Pointer to the input message.
<i>mesglen</i>	Message length in octets.
<i>pMD</i>	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

Example

The code example below shows SHA1 digest of a message.

```
//      Compute two SHA1 digests of a message:
//      1-st will correspond of 1/2 message
//      2-nd will correspond of whole message

void SHA1_sample(void){
    // get size of the SHA1 context
    int ctxSize;
    ippsSHA1GetSize(&ctxSize);

    // allocate the SHA1 context
    IppsSHA1State* pCtx = (IppsSHA1State*)( new Ipp8u [ctxSize] );
    // and initialize the context
    ippsSHA1Init(pCtx);
    // define a message
    Ipp8u msg[] = "abcdcbcdcedefdefgefghfghighijhi jkijkljklmklmnlmnomnopnopq";
    int n;

    // update digest using a piece of message
    for(n=0; n<(sizeof(msg)-1)/2; n++)
        ippsSHA1Update(msg+n, 1, pCtx);

// clone the SHA1 context
    IppsSHA1State* pCtx2 = (IppsSHA1State*)( new Ipp8u [ctxSize] );
    ippsSHA1Init(pCtx2);
    ippsSHA1Duplicate(pCtx, pCtx2);

// finalize and extract digest of a half message
    Ipp8u digest[20];
    ippsSHA1Final(digest, pCtx);

    // update digest using the SHA1 clone context
    ippsSHA1Update(msg+n, sizeof(msg)-1-n, pCtx2);

// finalize and extract digest of a whole message
    Ipp8u digest2[20];
    ippsSHA1Final(digest2, pCtx2);

    delete [] (Ipp8u*)pCtx;
```

```
delete [] (Ipp8u*)pCtx2;  
}
```

SHA224MessageDigest

Computes SHA-224 digest value of the input message.

Syntax

```
IppStatus ippsSHA224MessageDigest(const Ipp8u *pSrcMsg, int msglen, Ipp8u *pMD);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msglen</i>	Message length in octets.
<i>pMD</i>	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

SHA256MessageDigest

Computes SHA-256 digest value of the input message.

Syntax

```
IppStatus ippsSHA256MessageDigest(const Ipp8u *pSrcMsg, int msglen, Ipp8u *pMD);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msglen</i>	Message length in octets.
<i>pMD</i>	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

SHA384MessageDigest

Computes SHA-384 digest value of the input message.

Syntax

```
IppStatus ippSHA384MessageDigest(const Ipp8u *pSrcMsg, int msglen, Ipp8u *pMD);
```

Parameters

<code>pSrcMsg</code>	Pointer to the input message.
<code>msglen</code>	Message length in octets.
<code>pMD</code>	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

SHA512MessageDigest

Computes SHA-512 digest value of the input message.

Syntax

```
IppStatus ippSHA512MessageDigest(const Ipp8u *pSrcMsg, int msglen, Ipp8u *pMD);
```

Parameters

<code>pSrcMsg</code>	Pointer to the input message.
<code>msglen</code>	Message length in octets.
<code>pMD</code>	Pointer to the resultant digest.

Description

This function is declared in the `ippcp.h` file. The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

Mask Generation Functions

Public Key Cryptography frequently uses mask generation functions (MGFs) to achieve a particular security goal. For example, MGFs are used both in RSA-OAEP encryption and RSA-SSA signature schemes.

MGF function takes an octet string of a variable length and generates an octet string of a desired length. MGFs are deterministic, which means that the input octet string completely determines the output one. The output of an MGF should be pseudorandom, that is, infeasible to predict. The provable security of such cryptography schemes as RSA-OAEP or RSA-SSA, relies on the random nature of the MGF output. That is why one-way hash functions is one of the well-known ways to implement an MGF. The exact definition of an MGF based on a one-way hash function may be found in [PKCS 1.2.1].

This section describes MGFs based on widely-used MD5, SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 hash algorithms as well as the possibility to use a different implementation of MGF.



NOTE. Intel IPP implementation of MGFs limits length (in octets) of an input message for any specific MGF by the range of the `int` data type, with the upper bound of $2^{32}-1$.

User's Implementation of a Mask Generation Function

In case you prefer or have to use a different implementation of an MGF you can still use IPPCP. To do this, use the definition of MGF introduced in the IPPCP library and described in this section. The declaration provided below also defines an MGF when it is used as a parameter in some Public Key Cryptography operations.

Syntax

```
typedef IppStatus(_STDCALL *IppMGF)(const Ipp8u* pSeed, int seedLen, Ipp8u* pMask, int maskLen);
```

Parameters

<code>pSeed</code>	Pointer to the input octet string.
<code>seedLen</code>	Length of the input string.
<code>pMask</code>	Pointer to the output pseudorandom mask.
<code>maskLen</code>	Desired length of the output.

Description

This declaration is included in the `ippcp.h` file. The function generates an octet string of length *maskLen* according to the implemented algorithm, providing pseudorandom output.

MGF_MD5

Generates a pseudorandom mask of the specified length using MD5 hash function.

Syntax

```
IppStatus ippmGF_MD5(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int maskLen);
```

Parameters

<i>pSeed</i>	Pointer to the input octet string.
<i>seedLen</i>	Length of the input string.
<i>pMask</i>	Pointer to the output pseudorandom mask.
<i>maskLen</i>	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using MD5 hash algorithm.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if <i>pMask</i> pointer is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA1

Generates a pseudorandom mask of the specified length using SHA-1 hash function.

Syntax

```
IppStatus ippmGF_SHA1(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int maskLen);
```

Parameters

<i>pSeed</i>	Pointer to the input octet string.
<i>seedLen</i>	Length of the input string.
<i>pMask</i>	Pointer to the output pseudorandom mask.
<i>maskLen</i>	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-1 hash algorithm.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if <code>pMask</code> pointer is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA224

Generates a pseudorandom mask of the specified length using SHA-224 hash function.

Syntax

```
IppStatus ippmMGF_SHA224(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int maskLen);
```

Parameters

<code>pSeed</code>	Pointer to the input octet string.
<code>seedLen</code>	Length of the input string.
<code>pMask</code>	Pointer to the output pseudorandom mask.
<code>maskLen</code>	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-224 hash algorithm.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if <code>pMask</code> pointer is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA256

Generates a pseudorandom mask of the specified length using SHA-256 hash function.

Syntax

```
IppStatus ippmMGF_SHA256(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int maskLen);
```

Parameters

<i>pSeed</i>	Pointer to the input octet string.
<i>seedLen</i>	Length of the input string.
<i>pMask</i>	Pointer to the output pseudorandom mask.
<i>maskLen</i>	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-256 hash algorithm.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if <i>pMask</i> pointer is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA384

Generates a pseudorandom mask of the specified length using SHA-384 hash function.

Syntax

```
IppStatus ippmGF_SHA384(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int maskLen);
```

Parameters

<i>pSeed</i>	Pointer to the input octet string.
<i>seedLen</i>	Length of the input string.
<i>pMask</i>	Pointer to the output pseudorandom mask.
<i>maskLen</i>	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-384 hash algorithm.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if <i>pMask</i> pointers is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if any of the specified lengths is negative or zero.

MGF_SHA512

Generates a pseudorandom mask of the specified length using SHA-512 hash function.

Syntax

```
IppStatus ippsMGF_SHA512(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int maskLen);
```

Parameters

<i>pSeed</i>	Pointer to the input octet string.
<i>seedLen</i>	Length of the input string.
<i>pMask</i>	Pointer to the output pseudorandom mask.
<i>maskLen</i>	Desired length of the output.

Description

This function is declared in the `ippcp.h` file. The function generates a pseudorandom mask of the specified length using SHA-512 hash algorithm.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsNullPtrErr</code>	Indicates an error condition if <i>pMask</i> pointer is <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if any of the specified lengths is negative or zero.

Data Authentication Primitive Functions

This chapter describes the Intel® IPP functions for generating message authentication code (MAC), that is, [Message Authentication Functions](#), and the [Data Authentication Functions](#) (DAA) schemes.

Message Authentication Functions

Hash function-based MAC (HMAC) is widely used in the applications requiring message authentication and data integrity check. HMAC was initially put forward in [\[RFC 2401\]](#) and adopted by ANSI X9.71 and [\[FIPS PUB 198\]](#). See [Keyed Hash Functions](#) for a description of the Intel® Integrated Performance Primitives (Intel® IPP) HMAC primitives.

A MAC algorithm based on a symmetric key block cipher, in other words, a cipher-based MAC (CMAC), is standardized in [\[NIST SP 800-38B\]](#). CMAC may be appropriate for information systems where an approved block cipher is available rather than an approved hash function. See [CMAC Functions](#) for a description of the Intel IPP CMAC primitives.

AES-XCBC-MAC-96A, a specialization of MAC based on the AES block cipher in conjunction with the Cipher-Block-Chaining (CBC) mode of operation, is standardized in [\[RFC 3566\]](#) and has proved its security not only for messages of pre-selected fixed lengths but also for messages of varying lengths, such as used in typical IP datagrams. See [AES-XCBC Functions](#) for a description of the Intel IPP AES-XCBC-MAC-96A primitives.

Keyed Hash Functions

The Intel IPP HMAC primitive functions, described in this section, use various HMAC schemes based on one-way hash functions described in the [One-Way Hash Primitives](#) chapter.

Table “Intel IPP HMAC Functions” lists the Intel IPP HMAC functions.

Intel IPP HMAC Functions

Function Base Name	Operation
HMACSHA1GetSize	Gets the size of the <code>IppsHMACSHA1State</code> context.
HMACSHA1Init	Initializes user-supplied memory as <code>IppsHMACSHA1State</code> context for future use.
HMACSHA1Pack , HMACSHA1Unpack	Packs/unpacks the <code>IppsHMACSHA1State</code> context into/from a user-defined buffer.
HMACSHA1Duplicate	Copies one <code>IppsHMACSHA1State</code> context to another.

Function Base Name	Operation
HMACSHA1Update	Digests the current input message stream of the specified length using SHA1-based MAC.
HMACSHA1Final	Completes computation of the SHA1-based MAC value.
HMACSHA1GetTag	Computes the current HMAC value of the processed part of the message.
HMACSHA1MessageDigest	Computes the SHA1-based MAC value of an entire message.
HMACSHA224GetSize	Gets the size of the <code>IppsHMACSHA224State</code> context.
HMACSHA224Init	Initializes user-supplied memory as <code>IppsHMACSHA224State</code> context for future use.
HMACSHA224Pack , HMACSHA224Unpack	Packs/unpacks the <code>IppsHMACSHA224State</code> context into/from a user-defined buffer.
HMACSHA224Duplicate	Copies one <code>IppsHMACSHA224State</code> context to another.
HMACSHA224Update	Digests the current input message stream of the specified length using SHA224-based MAC.
HMACSHA224Final	Completes computation of the SHA224-based MAC value.
HMACSHA224GetTag	Computes the current SHA224-based MAC value of the processed part of the message.
HMACSHA224MessageDigest	Computes the SHA224-based MAC value of an entire message.
HMACSHA256GetSize	Gets the size of the <code>IppsHMACSHA256State</code> context.
HMACSHA256Init	Initializes user-supplied memory as <code>IppsHMACSHA256State</code> context for future use.
HMACSHA256Pack , HMACSHA256Unpack	Packs/unpacks the <code>IppsHMACSHA256State</code> context into/from a user-defined buffer.
HMACSHA256Duplicate	Copies one <code>IppsHMACSHA256State</code> context to another.
HMACSHA256Update	Digests the current input message stream of the specified length using SHA256-based MAC.
HMACSHA256Final	Completes computation of the SHA254-based MAC value.
HMACSHA256GetTag	Computes the current SHA256-based MAC value of the processed part of the message.
HMACSHA256MessageDigest	Computes the SHA256-based MAC value of an entire message.
HMACSHA384GetSize	Gets the size of the <code>IppsHMACSHA384State</code> context.

Function Base Name	Operation
<code>HMACSHA384Init</code>	Initializes user-supplied memory as <code>IppsHMACSHA384State</code> context for future use.
<code>HMACSHA384Pack</code> , <code>HMACSHA384Unpack</code>	Packs/unpacks the <code>IppsHMACSHA384State</code> context into/from a user-defined buffer.
<code>HMACSHA384Duplicate</code>	Copies one <code>IppsHMACSHA384State</code> context to another.
<code>HMACSHA384Update</code>	Digests the current input message stream of the specified length using SHA384-based MAC.
<code>HMACSHA384Final</code>	Completes computation of the SHA384-based MAC value.
<code>HMACSHA384GetTag</code>	Computes the current SHA384-based MAC value of the processed part of the message.
<code>HMACSHA384MessageDigest</code>	Computes the SHA384-based MAC value of an entire message.
<code>HMACSHA512GetSize</code>	Gets the size of the <code>IppsHMACSHA512State</code> context.
<code>HMACSHA512Init</code>	Initializes user-supplied memory as <code>IppsHMACSHA512State</code> context for future use.
<code>HMACSHA512Pack</code> , <code>HMACSHA512Unpack</code>	Packs/unpacks the <code>IppsHMACSHA512State</code> context into/from a user-defined buffer.
<code>HMACSHA512Duplicate</code>	Copies one <code>IppsHMACSHA512State</code> context to another.
<code>HMACSHA512Update</code>	Digests the current input message stream of the specified length using SHA512-based MAC.
<code>HMACSHA512Final</code>	Completes computation of the SHA512-based MAC value.
<code>HMACSHA512GetTag</code>	Computes the current SHA512-based MAC value of the processed part of the message.
<code>HMACSHA512MessageDigest</code>	Computes the SHA512-based MAC value of an entire message.
<code>HMACMD5GetSize</code>	Gets the size of the <code>IppsHMACMD5State</code> context.
<code>HMACMD5Init</code>	Initializes user-supplied memory as <code>IppsHMACMD5State</code> context for future use.
<code>HMACMD5Pack</code> , <code>HMACMD5Unpack</code>	Packs/unpacks the <code>IppsHMACMD5State</code> context into/from a user-defined buffer.
<code>HMACMD5Duplicate</code>	Copies one <code>IppsHMACMD5State</code> context to another.
<code>HMACMD5Update</code>	Digests the current input message stream of the specified length using MD5-based MAC.
<code>HMACMD5Final</code>	Completes computation of the MD5-based MAC value.

Function Base Name	Operation
HMACMD5GetTag	Computes the current MD5-based MAC value of the processed part of the message.
HMACMD5MessageDigest	Computes the MD5-based MAC value of an entire message.

Each HMAC scheme is implemented as a set of the primitive functions tabled above.

For example, the primitive implementing HMAC that is based on SHA-1 hash algorithm uses the `ippSHMACSHA1State` context as an operational vehicle to carry all necessary variables to manage computation of chaining digest value.

The function `HMACSHA1Init` initializes the context and sets up the specified initialization vectors. After the initialization, the function `HMACSHA1Update` digests the input message stream with the selected hash algorithm till it exhausts all message blocks.

The function `HMACSHA1Final` is designed to pad the partial message block into a final message block with the specified padding scheme, and then use the hash algorithm to transform the final block into a message digest value.

The following example illustrates how the application code can apply the implemented HMAC-SHA1 hash standard to digest the input message stream:

1. Call the function [HMACSHA1GetSize](#) to get the size required to configure the `IppsHMACSHA1State` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the function [HMACSHA1Init](#) to set up key material and the initial context state with the SHA-1 specified initialization vectors.
3. Keep calling the function [HMACSHA1Update](#) to digest incoming message stream in the queue till its completion. To determine the current value of the message digest, call [HMACSHA1GetTag](#) between the two calls to `HMACSHA1Update`.
4. Call the function [HMACSHA1Final](#) for padding the partial block into a final SHA-1 message block and transforming it into a resulting HMAC value.
5. Call the operating system memory free service function to release the `IppsHMACSHA1State` context.

The contexts used by the HMAC primitives (for example, `ippSHMACSHA1State`) are position-dependent. The following functions transform the respective position-dependent context to a position-independent form and vice versa:

- [HMACSHA1Pack/HMACSHA1Unpack](#)
- [HMACSHA224Pack/HMACSHA224Unpack](#)
- [HMACSHA256Pack/HMACSHA256Unpack](#)
- [HMACSHA384Pack/HMACSHA384Unpack](#)
- [HMACSHA512Pack/HMACSHA512Unpack](#)
- [HMACMD5Pack/HMACMD5Unpack](#)

HMACSHA1GetSize

Gets the size of the IppsHMACSHA1State context.

Syntax

```
IppStatus ippsHMACSHA1GetSize(int *pSize);
```

Parameters

pSize Pointer to the IppsHMACSHA1State context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the IppsHMACSHA1State context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

HMACSHA1Init

Initializes user-supplied memory as IppsHMACSHA1State context for future use.

Syntax

```
IppStatus ippsHMACSHA1Init(const Ipp8u *pKey, int keyLen, IppsHMACSHA1State * pCtx);
```

Parameters

<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length in bytes.
<i>pCtx</i>	Pointer to the IppsHMACSHA1State context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the IppsHMACSHA1State context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key *pKey*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>keyLen</i> is less than one.

HMACSHA1Pack, HMACSHA1Unpack

Packs/unpacks the IppsHMACSHA1State context into/from a user-defined buffer.

Syntax

```
IppStatus ippsHMACSHA1Pack (const IppsHMACSHA1State* pCtx, Ipp8u* pBuffer);  
IppStatus ippsHMACSHA1Unpack (const Ipp8u* pBuffer, IppsHMACSHA1State* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the IppsHMACSHA1State context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `HMACSHA1Pack` function transforms the **pCtx* context to a position-independent form and stores it in the **pBuffer* buffer. The `HMACSHA1Unpack` function performs the inverse operation, that is, transforms the contents of the **pBuffer* buffer into a normal `IppsHMACSHA1State` context. The `HMACSHA1Pack` and `HMACSHA1Unpack` functions enable replacing the position-dependent `IppsHMACSHA1State` context in the memory.

Call the [HMACSHA1GetSize](#) function prior to `HMACSHA1Pack/HMACSHA1Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

HMACSHA1Duplicate

Copies one IppsHMACSHA1State context to another.

Syntax

```
IppStatus ippsHMACSHA1Duplicate(const IppsHMACSHA1State* pSrcCtx, IppsHMACSHA1State* pDstCtx);
```

Parameters

<i>pSrcCtx</i>	Pointer to the source IppsHMACSHA1State context.
<i>pDstCtx</i>	Pointer to the source IppsHMACSHA1State context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA1State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA1Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippHMACSHA1Update(const Ipp8u *pSrcMsg, int msglen, IppsHMACSHA1State *pCtx);
```

Parameters

<code>pSrcMsg</code>	Pointer to the buffer containing a part of the whole message.
<code>msglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsHMACSHA1State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

HMACSHA1Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippHMACSHA1Final(Ipp8u *pMAC, int macLen, IppsHMACSHA1State *pCtx);
```

Parameters

<code>pMAC</code>	Pointer to the resultant HMAC value.
<code>macLen</code>	Specified HMAC length.

*pCtx*Pointer to the `IppsHMACSHA1State` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>macLen</i> is less than one or greater than the length of the hash value.

HMACSHA1GetTag

Computes the current HMAC value of the processed part of the message.

Syntax

```
IppStatus ippHMACSHA1GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsHMACSHA1State* pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppsHMACSHA1State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 198]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>tagLen</i> < 1 or <i>tagLen</i> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA1MessageDigest

Computes the HMAC value of an entire message.

Syntax

```
IppStatus ippSHA1MessageDigest(const Ipp8u *pSrcMesg, int msgLen, const Ipp8u *pKey,
int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<i>pSrcMesg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length in bytes.
<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>msgLen</i> is less than zero or <i>macLen</i> is less than one or greater than the length of the hash value.

HMACSHA224GetSize

Gets the size of the IppsHMACSHA224State context.

Syntax

```
IppStatus ippSHA224GetSize(int *pSize);
```

Parameters

<i>pSize</i>	Pointer to the <code>IppsHMACSHA224State</code> context size value.
--------------	---

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA224State` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

HMACSHA224Init

*Initializes user-supplied memory as
IppsHMACSHA224State context for future use.*

Syntax

```
IppStatus ippshMACSHA224Init(const Ipp8u *pKey, int keyLen, IppsHMACSHA224State * pCtx);
```

Parameters

<code>pKey</code>	Pointer to the user-supplied key.
<code>keyLen</code>	Key length in bytes.
<code>pCtx</code>	Pointer to the <code>IppsHMACSHA224State</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as the `IppsHMACSHA224State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key `pKey`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keyLen</code> is less than one.

HMACSHA224Pack, HMACSHA224Unpack

*Packs/unpacks the IppsHMACSHA224State context
into/from a user-defined buffer.*

Syntax

```
IppStatus ippshMACSHA224Pack (const IppsHMACSHA224State* pCtx, Ipp8u* pBuffer);  
IppStatus ippshMACSHA224Unpack (const Ipp8u* pBuffer, IppsHMACSHA224State* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsHMACSHA224State</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `HMACSHA224Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the the `*pBuffer` buffer. The `HMACSHA224Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsHMACSHA224State` context. The `HMACSHA224Pack` and `HMACSHA224Unpack` functions enable replacing the position-dependent `IppsHMACSHA224State` context in the memory.

Call the [HMACSHA224GetSize](#) function prior to `HMACSHA224Pack/HMACSHA224Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

HMACSHA224Duplicate

Copies one `IppsHMACSHA224State` context to another.

Syntax

```
IppStatus ippHMACSHA224Duplicate(const IppsHMACSHA224State* pSrcCtx, IppsHMACSHA224State* pDstCtx);
```

Parameters

<code>pSrcCtx</code>	Pointer to the source <code>IppsHMACSHA224State</code> context.
<code>pDstCtx</code>	Pointer to the source <code>IppsHMACSHA224State</code> context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA224State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA224Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippHMACSHA224Update(const Ipp8u *pSrcMesg, int mesglen, IppsHMACSHA224State *pCtx);
```

Parameters

<i>pSrcMesg</i>	Pointer to the buffer containing a part of the whole message.
<i>mesglen</i>	Length of the actual part of the message in bytes.
<i>pCtx</i>	Pointer to the <code>IppsHMACSHA224State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

HMACSHA224Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippHMACSHA224Final(Ipp8u *pMAC, int macLen, IppsHMACSHA224State *pCtx);
```

Parameters

<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.
<i>pCtx</i>	Pointer to the <code>IppsHMACSHA224State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

`ippStsLengthErr` Indicates an error condition if `macLen` is less than one or greater than the length of the hash value.

HMACSHA224GetTag

Computes the current HMAC value of the processed part of the message.

Syntax

```
IppStatus ippsha224GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const Ippsha224State* pState);
```

Parameters

<code>pDstTag</code>	Pointer to the authentication tag.
<code>tagLen</code>	Length of the tag (in bytes).
<code>pState</code>	Pointer to the <code>Ippsha224State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 198]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA224MessageDigest

Computes the HMAC value of an entire message.

Syntax

```
IppStatus ippsha224MessageDigest(const Ipp8u *pSrcMesg, int mesgLen, const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<code>pSrcMesg</code>	Pointer to the input message.
<code>mesgLen</code>	Message length in bytes.
<code>pKey</code>	Pointer to the user-supplied key.
<code>keyLen</code>	Key length in bytes.
<code>pMAC</code>	Pointer to the resultant HMAC value.

macLen Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>msgLen</i> is less than zero or <i>macLen</i> is less than one or greater than the length of the hash value.

HMACSHA256GetSize

Gets the size of the `IppsHMACSHA256State` context.

Syntax

```
IppStatus ippHMACSHA256GetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsHMACSHA256State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA256State` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

HMACSHA256Init

Initializes user-supplied memory as `IppsHMACSHA256State` context for future use.

Syntax

```
IppStatus ippHMACSHA256Init(const Ipp8u *pKey, int keyLen, IppsHMACSHA256State * pCtx);
```

Parameters

<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsHMACSHA256State</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as the `IppsHMACSHA256State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key `pKey`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keyLen</code> is less than one.

HMACSHA256Pack, HMACSHA256Unpack

Packs/unpacks the `IppsHMACSHA256State` context into/from a user-defined buffer.

Syntax

```
IppStatus ippsha256Pack (const IppsHMACSHA256State* pCtx, Ipp8u* pBuffer);
IppStatus ippsha256Unpack const (const Ipp8u* pBuffer, IppsHMACSHA256State* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsHMACSHA256State</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `HMACSHA256Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the `*pBuffer` buffer. The `HMACSHA256Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsHMACSHA256State` context. The `HMACSHA256Pack` and `HMACSHA256Unpack` functions enable replacing the position-dependent `IppsHMACSHA256State` context in the memory.

Call the [HMACSHA256GetSize](#) function prior to `HMACSHA256Pack/HMACSHA256Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

HMACSHA256Duplicate

Copies one `IppsHMACSHA256State` context to another.

Syntax

```
IppStatus ippsha256Duplicate(const IppsHMACSHA256State* pSrcCtx, IppsHMACSHA256State* pDstCtx);
```

Parameters

<code>pSrcCtx</code>	Pointer to the source <code>IppsHMACSHA256State</code> context.
<code>pDstCtx</code>	Pointer to the source <code>IppsHMACSHA256State</code> context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA256State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA256Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippHMACSHA256Update(const Ipp8u *pSrcMsg, int msglen, IppsHMACSHA256State *pCtx);
```

Parameters

<code>pSrcMsg</code>	Pointer to the buffer containing a part of the whole message.
<code>msglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsHMACSHA256State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

HMACSHA256Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACSHA256Final(Ipp8u *pMAC, int macLen, IppsHMACSHA256State *pCtx);
```

Parameters

<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.
<i>pCtx</i>	Pointer to the <code>IppsHMACSHA256State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>macLen</i> is less than one or greater than the length of the hash value.

HMACSHA256GetTag

Computes the current HMAC value of the processed part of the message.

Syntax

```
IppStatus ippsHMACSHA256GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsHMACSHA256State* pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppsHMACSHA256State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 198]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA256MessageDigest

Computes the HMAC value of an entire message.

Syntax

```
ippStatus ippsha256HMACMessageDigest(const Ipp8u *pSrcMesg, int msgLen, const Ipp8u
*pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<code>pSrcMesg</code>	Pointer to the input message.
<code>msgLen</code>	Message length in bytes.
<code>pKey</code>	Pointer to the user-supplied key.
<code>keyLen</code>	Key length in bytes.
<code>pMAC</code>	Pointer to the resultant HMAC value.
<code>macLen</code>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key `pKey` of the specified key length `keyLen` and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code `pMAC` of the specified length `macLen`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>msgLen</code> is less than zero or <code>macLen</code> is less than one or greater than the length of the hash value.

Example

The code example below computes the SHA256 HMACs of a message.

```
// Compute two keyed-hash based message
// authentication codes using the HMAC scheme
// 1-st will correspond of 1/2 message
// 2-nd will correspond of whole message

void HMACSHA256_sample(void){
    // define key
```



```

Ipp8u key[] = "the key for HMAC scheme";

// get size of HMACSHA256 context
int ctxSize;
ippsHMACSHA256GetSize(&ctxSize);

// allocate HMACSHA256 context
IppsHMACSHA256State* pCtx = (IppsHMACSHA256State*)( new Ipp8u [ctxSize] );
// and ini context
ippsHMACSHA256Init(key, strlen((char*)key), pCtx);
// define message
Ipp8u msg[] = "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq";

int n;
// update MAC using piece of message
for(n=0; n<(sizeof(msg)-1)/2; n++)
    ippsHMACSHA256Update(msg+n, 1, pCtx);

// clone HMACSHA256 context
IppsHMACSHA256State* pCtx2 = (IppsHMACSHA256State*)( new Ipp8u [ctxSize] );
ippsHMACSHA256Init(key, strlen((char*)key), pCtx2);
ippsHMACSHA256Duplicate(pCtx, pCtx2);

// finalize and extract digest of a half message
const int macLen = 16;
Ipp8u mac[macLen];
ippsHMACSHA256Final(mac, macLen, pCtx);

// update MAC using HMACSHA256 clone context
ippsHMACSHA256Update(msg+n, sizeof(msg)-1-n, pCtx2);

// finalize and extract digest of a whole message
Ipp8u mac2[macLen];
ippsHMACSHA256Final(mac2, macLen, pCtx2);

delete [] (Ipp8u*)pCtx;
delete [] (Ipp8u*)pCtx2;
}

```

HMACSHA384GetSize

Gets the size of the IppsHMACSHA384State context.

Syntax

```
IppStatus ippsHMACSHA384GetSize(int *pSize);
```

Parameters

pSize Pointer to the IppsHMACSHA384State context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA384State` context size in bytes and stores it in *pSize*.

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`.

HMACSHA384Init

*Initializes user-supplied memory as
IppsHMACSHA384State context for future use.*

Syntax

```
IppStatus ippshMACSHA384Init(const Ipp8u *pKey, int keyLen, IppsHMACSHA384State * pCtx);
```

Parameters

<code>pKey</code>	Pointer to the user-supplied key.
<code>keyLen</code>	Key length in bytes.
<code>pCtx</code>	Pointer to the <code>IppsHMACSHA384State</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as the `IppsHMACSHA384State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key `pKey`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keyLen</code> is less than one.

HMACSHA384Pack, HMACSHA384Unpack

*Packs/unpacks the IppsHMACSHA384State context
into/from a user-defined buffer.*

Syntax

```
IppStatus ippshMACSHA384Pack (const IppsHMACSHA384State* pCtx, Ipp8u* pBuffer);  
IppStatus ippshMACSHA384Unpack (const Ipp8u* pBuffer, IppsHMACSHA384State* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsHMACSHA384State</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `HMACSHA384Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the the `*pBuffer` buffer. The `HMACSHA384Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsHMACSHA384State` context. The `HMACSHA384Pack` and `HMACSHA384Unpack` functions enable replacing the position-dependent `IppsHMACSHA384State` context in the memory.

Call the [HMACSHA384GetSize](#) function prior to `HMACSHA384Pack/HMACSHA384Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

HMACSHA384Duplicate

Copies one `IppsHMACSHA384State` context to another.

Syntax

```
IppStatus ippHMACSHA384Duplicate(const IppsHMACSHA384State* pSrcCtx, IppsHMACSHA384State* pDstCtx);
```

Parameters

<code>pSrcCtx</code>	Pointer to the source <code>IppsHMACSHA384State</code> context.
<code>pDstCtx</code>	Pointer to the source <code>IppsHMACSHA384State</code> context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA384State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA384Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippHMACSHA384Update(const Ipp8u *pSrcMesg, int mesglen, IppsHMACSHA384State *pCtx);
```

Parameters

<i>pSrcMesg</i>	Pointer to the buffer containing a part of the whole message.
<i>mesglen</i>	Length of the actual part of the message in bytes.
<i>pCtx</i>	Pointer to the <code>IppsHMACSHA384State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

HMACSHA384Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippHMACSHA384Final(Ipp8u *pMAC, int macLen, IppsHMACSHA384State *pCtx);
```

Parameters

<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.
<i>pCtx</i>	Pointer to the <code>IppsHMACSHA384State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

`ippStsLengthErr` Indicates an error condition if `macLen` is less than one or greater than the length of the hash value.

HMACSHA384GetTag

Computes the current HMAC value of the processed part of the message.

Syntax

```
IppStatus ippshMACSHA384GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppshMACSHA384State* pState);
```

Parameters

<code>pDstTag</code>	Pointer to the authentication tag.
<code>tagLen</code>	Length of the tag (in bytes).
<code>pState</code>	Pointer to the <code>IppshMACSHA384State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 198]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA384MessageDigest

Computes the HMAC value of an entire message.

Syntax

```
IppStatus ippshMACSHA384MessageDigest(const Ipp8u *pSrcMesg, int mesgLen, const Ipp8u *pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<code>pSrcMesg</code>	Pointer to the input message.
<code>mesgLen</code>	Message length in bytes.
<code>pKey</code>	Pointer to the user-supplied key.
<code>keyLen</code>	Key length in bytes.
<code>pMAC</code>	Pointer to the resultant HMAC value.

macLen Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>msgLen</i> is less than zero or <i>macLen</i> is less than one or greater than the length of the hash value.

HMACSHA512GetSize

Gets the size of the `IppsHMACSHA512State` context.

Syntax

```
IppStatus ippHMACSHA512GetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsHMACSHA512State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACSHA512State` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

HMACSHA512Init

Initializes user-supplied memory as `IppsHMACSHA512State` context for future use.

Syntax

```
IppStatus ippHMACSHA512Init(const Ipp8u *pKey, int keyLen, IppsHMACSHA512State * pCtx);
```

Parameters

<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsHMACSHA512State</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as the `IppsHMACSHA512State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key `pKey`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keyLen</code> is less than one.

HMACSHA512Pack, HMACSHA512Unpack

Packs/unpacks the `IppsHMACSHA512State` context into/from a user-defined buffer.

Syntax

```
IppStatus ippsha512Pack (const IppsHMACSHA512State* pCtx, Ipp8u* pBuffer);
IppStatus ippsha512Unpack (const Ipp8u* pBuffer, IppsHMACSHA512State* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsHMACSHA512State</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `HMACSHA512Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the `*pBuffer` buffer. The `HMACSHA512Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsHMACSHA512State` context. The `HMACSHA512Pack` and `HMACSHA512Unpack` functions enable replacing the position-dependent `IppsHMACSHA512State` context in the memory.

Call the [HMACSHA512GetSize](#) function prior to `HMACSHA512Pack/HMACSHA512Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

HMACSHA512Duplicate

Copies one `IppsHMACSHA512State` context to another.

Syntax

```
IppStatus ippsha512Duplicate(const IppsHMACSHA512State* pSrcCtx, IppsHMACSHA512State* pDstCtx);
```

Parameters

<code>pSrcCtx</code>	Pointer to the source <code>IppsHMACSHA512State</code> context.
<code>pDstCtx</code>	Pointer to the source <code>IppsHMACSHA512State</code> context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACSHA512State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA512Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippHMACSHA512Update(const Ipp8u *pSrcMsg, int msglen, IppsHMACSHA512State *pCtx);
```

Parameters

<code>pSrcMsg</code>	Pointer to the buffer containing a part of the whole message.
<code>msglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsHMACSHA512State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

HMACSHA512Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACSHA512Final(Ipp8u *pMAC, int macLen, IppsHMACSHA512State * pCtx);
```

Parameters

<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.
<i>pCtx</i>	Pointer to the <code>IppsHMACSHA512State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>macLen</i> is less than one or greater than the length of the hash value.

HMACSHA512GetTag

Computes the current HMAC value of the processed part of the message.

Syntax

```
IppStatus ippsHMACSHA512GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsHMACSHA512State* pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppsHMACSHA512State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 198]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen</code> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACSHA512MessageDigest

Computes the HMAC value of an entire message.

Syntax

```
IppStatus ippsha512MessageDigest(const Ipp8u *pSrcMesg, int msgLen, const Ipp8u
*pKey, int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<code>pSrcMesg</code>	Pointer to the input message.
<code>msgLen</code>	Message length in bytes.
<code>pKey</code>	Pointer to the user-supplied key.
<code>keyLen</code>	Key length in bytes.
<code>pMAC</code>	Pointer to the resultant HMAC value.
<code>macLen</code>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key `pKey` of the specified key length `keyLen` and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code `pMAC` of the specified length `macLen`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>msgLen</code> is less than zero or <code>macLen</code> is less than one or greater than the length of the hash value.

HMACMD5GetSize

Gets the size of the IppsHMACMD5State context.

Syntax

```
IppStatus ippsha512GetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsHMACMD5State` context size value.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsHMACMD5State` context size in bytes and stores it in *pSize*.

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`.

HMACMD5Init

Initializes user-supplied memory as `IppsHMACMD5State` context for future use.

Syntax

```
IppStatus ippshMACMD5Init(const Ipp8u *pKey, int keyLen, IppsHMACMD5State *pCtx);
```

Parameters

pKey Pointer to the user-supplied key.
keyLen Key length in bytes.
pCtx Pointer to the `IppsHMACMD5State` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsHMACMD5State` context. The function also sets up the initial chaining digest value according to the Hash algorithm specified by the function base name and computes necessary key material from the supplied key *pKey*.

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`.
`ippStsLengthErr` Indicates an error condition if *keyLen* is less than one.

HMACMD5Pack, HMACMD5Unpack

Packs/unpacks the `IppsHMACMD5State` context into/from a user-defined buffer.

Syntax

```
IppStatus ippshMACMD5Pack (const IppsHMACMD5State* pCtx, Ipp8u* pBuffer);  

IppStatus ippshMACMD5Unpack (const Ipp8u* pBuffer, IppsHMACMD5State* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsHMACMD5State</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `HMACMD5Pack` function transforms the `*pCtx` context to a position-independent form and stores it in the the `*pBuffer` buffer. The `HMACMD5Unpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsHMACMD5State` context. The `HMACMD5Pack` and `HMACMD5Unpack` functions enable replacing the position-dependent `IppsHMACMD5State` context in the memory.

Call the [HMACMD5GetSize](#) function prior to `HMACMD5Pack/HMACMD5Unpack` to determine the size of the buffer.

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 <code>NULL</code> .

HMACMD5Duplicate

Copies one `IppsHMACMD5State` context to another.

Syntax

```
IppStatus ippshMACMD5Duplicate(const IppsHMACMD5State* pSrcCtx, IppsHMACMD5State* pDstCtx);
```

Parameters

<code>pSrcCtx</code>	Pointer to the source <code>IppsHMACMD5State</code> context.
<code>pDstCtx</code>	Pointer to the source <code>IppsHMACMD5State</code> context to be cloned.

Description

The function is declared in the `ippcp.h` file. The function copies one `IppsHMACMD5State` context to another.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACMD5Update

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsHMACMD5Update(const Ipp8u *pSrcMesg, int mesglen, IppsHMACMD5State *pCtx);
```

Parameters

<i>pSrcMesg</i>	Pointer to the buffer containing a part of the whole message.
<i>mesglen</i>	Length of the actual part of the message in bytes.
<i>pCtx</i>	Pointer to the <code>IppsHMACMD5State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length.

The function first integrates the previous partial block with the input message stream and then partitions them into multiple message blocks (as specified by the applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected hash algorithm to transform the block into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.

HMACMD5Final

Completes computation of the HMAC value.

Syntax

```
IppStatus ippsHMACMD5Final(Ipp8u *pMAC, int macLen, IppsHMACMD5State *pCtx);
```

Parameters

<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.
<i>pCtx</i>	Pointer to the <code>IppsHMACMD5State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into the specified *pMD* memory.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <i>macLen</i> is less than one or greater than the length of the hash value.

HMACMD5GetTag

Computes the current HMAC value of the processed part of the message.

Syntax

```
IppStatus ippSHMACMD5GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppSHMACMD5State* pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppSHMACMD5State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the message digest based on the current context as specified in [FIPS PUB 198]. A call to this function retains the possibility to update the digest.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>tagLen</i> < 1 or <i>tagLen</i> exceeds the maximal length of a particular digest.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

HMACMD5MessageDigest

Computes the HMAC value of the message in a single call.

Syntax

```
IppStatus ippshMACMD5MessageDigest(const Ipp8u *pSrcMesg, int msgLen, const Ipp8u *pKey,
int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<i>pSrcMesg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length in bytes.
<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applies the keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>msgLen</i> is less than zero or <i>macLen</i> is less than one or greater than the length of the hash value.

Example

The code example below computes MD5 HMAC of a message.

```
MD5 HMAC of a Messagevoid HMACMD5_sample(void){
    // define key
    Ipp8u key[] = "the key for HMAC scheme";

    // define message
    Ipp8u msg[] = "abcdefghijklmnopqrstuvwxyz";

    // as soon as whole message placed into memory
    // one can use integrated primitive
    int macLen = 12;
    Ipp8u mac[16];
    ippshMACMD5MessageDigest(msg, strlen((char*)msg),
                             key, strlen((char*)key),
                             mac, macLen);
}
```

CMAC Functions

The Intel IPP CMAC primitive functions use CMAC schemes based on block ciphers described in the [Symmetric Cryptography Primitive Functions](#) chapter.

Table “Intel IPP CMAC Functions” lists the Intel IPP CMAC functions. A typical usage of the CMAC primitives is similar to the one of the [Keyed Hash Functions](#).

Intel IPP CMAC Functions

Function Base Name	Operation
CMACRijndael128GetSize	Gets the size of the <code>IppsCMACRijndael128State</code> context.
CMACRijndael128Init , CMACSafeRijndael128Init	Initialize user-supplied memory as <code>IppsCMACRijndael128State</code> context for future use.
CMACRijndael128Update	Updates the MAC value depending on the current input message stream of the specified length.
CMACRijndael128Final	Completes computation of the MAC value.
CMACRijndael128MessageDigest	Computes the MAC value of the entire message.

CMACRijndael128GetSize

Gets the size of the `IppsCMACRijndael128State` context.

Syntax

```
IppStatus ippsCMACRijndael128GetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsCMACRijndael128State` context.

Description

This function is declared in the `ippcp.h` file. It gets the size of the `IppsCMACRijndael128State` context.

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`.

CMACRijndael128Init, CMACSafeRijndael128Init

*Initialize user-supplied memory as
IppsCMACRijndael128State context for future use.*

Syntax

```
IppStatus ippsCMACRijndael128Init(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,  
IppsCMACRijndael128State* pState);
```

```
IppStatus ippsCMACSafeRijndael128Init(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,  
IppsCMACRijndael128State* pState);
```

Parameters

<i>pKey</i>	Pointer to the Rijndael128 key.
<i>keyLen</i>	Key bytestream length (in bytes) defined by the IppsRijndaelKeyLength enumerator.
<i>pState</i>	Pointer to the <i>IppsCMACRijndael128State</i> being initialized.

Description

These functions are declared in the `ippcp.h` file. Each function initializes the memory at the address of *pState* as the *IppsCMACRijndael128State* context. In addition, each function uses the key to provide all necessary key material for both encryption and decryption operations. CMAC based on the Rijndael128 cipher scheme uses the AES algorithm. Depending upon whether you wish to employ fast or safe implementation of the AES algorithm, call *CMACRijndael128Init* or *CMACSafeRijndael128Init*, respectively. For more information, see [Rijndael Functions](#) in chapter 2.

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 <code>NULL</code> .
<i>ippStsLengthErr</i>	Indicates an error condition if <i>keyLen</i> is not equal to IppsRijndaelKey128, IppsRijndaelKey192, or IppsRijndaelKey256.

CMACRijndael128Update

*Updates the MAC value depending on the current input
message stream of the specified length.*

Syntax

```
IppStatus ippsCMACRijndael128Update(const Ipp8u *pSrc, int len, IppsCMACRijndael128State*  
pState);
```

Parameters

<i>pSrc</i>	Pointer to the buffer containing a part or the entire message.
<i>len</i>	Length of the actual part of the message in bytes.

pState Pointer to the `IppsCMACRijndael128State` context.

Description

This function is declared in the `ippcp.h` file. The function updates the MAC value depending on the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream and then partitions the obtained message into multiple message blocks with a possible additional partial block. For each message block, the function uses the Rijndael128 cipher to transform the input block into a new chaining MAC value.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

CMACRijndael128Final

Completes computation of the MAC value.

Syntax

```
IppStatus ippCMACRijndael128Final(Ipp8u *pMD, int mdLen, IppsCMACRijndael128State *pState);
```

Parameters

<i>pMD</i>	Pointer to the MAC value.
<i>mdLen</i>	Specified length of the MAC.
<i>pState</i>	Pointer to the <code>IppsCMACRijndael128State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stores the result into memory at the address of *pMD*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>mdLen</i> is less than 1 or greater than cipher's data block length.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

CMACRijndael128MessageDigest

Computes the MAC value of the entire message.

Syntax

```
IppStatus ippSCMACRijndael128MessageDigest(const Ipp8u *pMsg, int msgLen, const Ipp8u
*pKey, IppsRijndaelKeyLength keyLen, Ipp8u *pMD, int mdLen);
```

Parameters

<i>pMsg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length.
<i>pMD</i>	Pointer to the resulting MAC value.
<i>mdLen</i>	Specified MAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applies keyed cipher-based message authentication code scheme to transform the *entire* input message into the respective message authentication code *pMAC* of the specified length *mdLen*.

The function is actually a wrapper around the [CMACRijndael128Init](#), [CMACSafeRijndael128Init](#), [CMACRijndael128Update](#), and [CMACRijndael128Final](#) functions. If your application has no access to *all* the necessary data, you can compute the MAC using a typical sequence of invoking the primitives, like the one described at the beginning of the [Keyed Hash Functions](#) section. `CMACMessageDigest` is convenient when the application can access the entire message. In this case, you can use this function to compute the result in a single call.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>msgLen</i> is less than zero, <i>mdLen</i> is less than 1 or greater than cipher's data block length, or <i>keyLen</i> value is illegal.

AES-XCBC Functions

The AES-XCBC-MAC-96 [[RFC 3566](#)] extends the classic CBC-MAC algorithm to messages of varying lengths.

[Table "Intel IPP AES-XCBC Functions"](#) lists the Intel IPP AES-XCBC functions. A typical usage of the AES-XCBC primitives is similar to the one of [Keyed Hash Functions](#).

Intel IPP AES-XCBC Functions

Function Base Name	Operation
XCBCRijndael128GetSize	Gets the size of the <code>IppsXCBCRijndael128State</code> context.

Function Base Name	Operation
<code>XCBCRijndael128Init</code>	Initializes user-supplied memory as <code>IppsCMACRijndael128State</code> context for future use.
<code>XCBCRijndael128Update</code>	Updates the internal authentication tag depending on the current input message.
<code>XCBCRijndael128GetTag</code>	Computes the authentication tag.
<code>XCBCRijndael128Final</code>	Computes the authentication tag and terminates the authentication process.
<code>XCBCRijndael128MessageTag</code>	Computes the authentication tag of an entire message.

XCBCRijndael128GetSize

Gets the size of the `IppsXCBCRijndael128State` context.

Syntax

```
IppStatus ippsXCBCRijndael128GetSize(Ipp32u* pSize);
```

Parameters

pSize Pointer to the size size of the `IppsXCBCRijndael128State` context.

Description

This function is declared in the `ippcp.h` file. The function gets the size of the `IppsXCBCRijndael128State` context in bytes and stores it in **pSize*.

Return Values

`ippStsNoErr` Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr` Indicates an error condition if the specified pointer is `NULL`.

XCBCRijndael128Init

Initializes user-supplied memory as the `IppsXCBCRijndael128State` context for future use.

Syntax

```
IppStatus ippsXCBCRijndael128Init(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,
IppsXCBCRijndael128State* pState);
```

Parameters

pKey Pointer to the authentication key.

<i>keyLen</i>	Length of the key.
<i>pState</i>	Pointer to the <code>IppsXCBCRijndael128State</code> context.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pState* as the `IppsXCBCRijndael128State` context.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>keyLen</i> is not set to <code>IppsRijndaelKey128</code> , <code>IppsRijndaelKey192</code> or <code>IppsRijndaelKey256</code> .

XCBCRijndael128Update

Updates the internal authentication tag depending on the current input message.

Syntax

```
IppStatus ippXCBCRijndael128Update(const Ipp8u* pSrc, Ipp32u len, IppsXCBCRijndael128State* pState);
```

Parameters

<i>pSrc</i>	Pointer to the input data.
<i>len</i>	Length of the input data in bytes.
<i>pState</i>	Pointer to the <code>IppsXCBCRijndael128State</code> context.

Description

This function is declared in the `ippcp.h` file. The function updates the internal value of the authentication tag according to [RFC 3566].

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

XCBCRijndael128GetTag

Computes the authentication tag.

Syntax

```
IppStatus ippsXCBCRijndael128GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const
IppsXCBCRijndael128State* pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppsXCBCRijndael128State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the authentication tag based on the current context as specified in [RFC 3566]. A call to `ippsXCBCRijndael128GetTag` does not stop the process of updating the authentication tag.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen > 16</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

XCBCRijndael128Final

Computes the authentication tag and terminates the authentication process.

Syntax

```
IppStatus ippsXCBCRijndael128Final(Ipp8u* pDstTag, Ipp32u tagLen, IppsXCBCRijndael128State*
pState);
```

Parameters

<i>pDstTag</i>	Pointer to the authentication tag.
<i>tagLen</i>	Length of the tag (in bytes).
<i>pState</i>	Pointer to the <code>IppsXCBCRijndael128State</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the authentication tag based on the current context as specified in [RFC 3566]. Additionally `ippsXCBCRijndael128Final` reinitializes the internal state to enable computation of the authentication tag for another message.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>tagLen < 1</code> or <code>tagLen > 16</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

XCBCRijndael128MessageTag

Computes the authentication tag of an entire message.

Syntax

```
IppStatus ippXCBCRijndael128MessageTag(const Ipp8u* pMsg, Ipp32u msgLen, const Ipp8u*
pKey, IppsRijndaelKeyLength keyLen, Ipp8u* pDstTag, Ipp32u tagLen);
```

Parameters

<code>pMsg</code>	Pointer to the input message.
<code>msgLen</code>	Length of the message (in bytes).
<code>pKey</code>	Pointer to the authentication key.
<code>keyLen</code>	Length of the key.
<code>pDstTag</code>	Pointer to the authentication tag.
<code>tagLen</code>	Length of the tag (in bytes).

Description

This function is declared in the `ippcp.h` file. The function performs all steps of the authentication tag calculation, that is, the initialize, update, and final steps, in a single call. You can use this function when your application can access the entire message.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the length parameters do not meet any of the following conditions: <code>keyLen = IppsRijndaelKey128</code> or <code>keyLen = IppsRijndaelKey192</code> or <code>keyLen = IppsRijndaelKey256</code> $1 \leq tagLen \leq 16$.

Data Authentication Functions

DAA is widely used in applications to detect both intentional and accidental unauthorized data modifications. DAA specification can be found in [FIPS PUB 113].

The Intel IPP Data Authentication (DAA) primitive functions are designed for applications to use various DAA schemes based on the set of symmetric cryptography functions described in the [Symmetric Cryptography Primitive Functions](#) chapter.

The implementation of each DAA scheme is presented as a set of primitive functions.

The full list of Intel IPP DAA Functions is given in [Table “Intel IPP Data Authentication Functions”](#).

Intel IPP Data Authentication Functions

Function Base Name	Operation
DAADESGetSize	Gets the size of the <code>IppsDAADESState</code> context.
DAADESInit	Initializes user-supplied memory as <code>IppsDAADESState</code> context for future use.
DAADESUpdate	Digests the current input message stream of the specified length.
DAADESFinal	Completes computation of the DAC value.
DAADESMessageDigest	Computes the DAC value of the message.
DAATDESGetSize	Gets the size of <code>IppsDAATDESState</code> context.
DAATDESInit	Initializes user-supplied memory as <code>IppsDAATDESState</code> context for future use.
DAATDESUpdate	Digests the current input message stream of the specified length.
DAATDESFinal	Completes computation of the DAC value.
DAATDESMessageDigest	Computes the DAC value of the message.
DAARijndael128GetSize	Gets the size of the <code>IppsDAARijndael128State</code> context.
DAARijndael128Init , DAASafeRijndael128Init	Initialize user-supplied memory as <code>IppsDAARijndael128State</code> context for future use.
DAARijndael128Update	Digests the current input message stream of the specified length.
DAARijndael128Final	Completes computation of the DAC value.
DAARijndael128MessageDigest	Computes the DAC value of the message.
DAARijndael192GetSize	Gets the size of the <code>IppsDAARijndael192State</code> context.
DAARijndael192Init	Initializes user-supplied memory as <code>IppsDAARijndael192State</code> context for future use.
DAARijndael192Update	Digests the current input message stream of the specified length.

Function Base Name	Operation
<code>DAARijndael192Final</code>	Completes computation of the DAC value.
<code>DAARijndael192MessageDigest</code>	Computes the DAC value of the message.
<code>DAARijndael256GetSize</code>	Gets the size of the <code>IppsDAARijndael256State</code> context.
<code>DAARijndael256Init</code>	Initializes user-supplied memory as <code>IppsDAARijndael256State</code> context for future use.
<code>DAARijndael256Update</code>	Digests the current input message stream of the specified length.
<code>DAARijndael256Final</code>	Completes computation of the DAC value.
<code>DAARijndael256MessageDigest</code>	Computes the DAC value of the message.
<code>DAABlowfishGetSize</code>	Gets the size of the <code>IppsDAABlowfishState</code> context.
<code>DAABlowfishInit</code>	Initializes user-supplied memory as <code>IppsDAABlowfishState</code> context for future use.
<code>DAABlowfishUpdate</code>	Digests the current input message stream of the specified length.
<code>DAABlowfishFinal</code>	Completes computation of the DAC value.
<code>DAABlowfishMessageDigest</code>	Computes the DAC value of the message.
<code>DAATwofishGetSize</code>	Gets the size of the <code>IppsDAATwofishState</code> context.
<code>DAATwofishInit</code>	Initializes user-supplied memory as <code>IppsDAATwofishState</code> context for future use.
<code>DAATwofishUpdate</code>	Digests the current input message stream of the specified length.
<code>DAATwofishFinal</code>	Completes computation of the DAC value.
<code>DAATwofishMessageDigest</code>	Computes the DAC value of the message.

The primitive implementing a DAA scheme uses the context as the operational vehicle to carry all necessary variables to manage computation of the chaining digest value. For example, the primitive implementing the DAA scheme based on the Rijndael128 block cipher uses `ippsDAARijndael128` context.

The function `Init` (`DAARijndael128Init`, `DAADESInit`, and others) initializes the context and sets up the specified initialization vectors. Once initialized, the function `Update` (`DAARijndael128Update`, `DAADESUpdate`, and others) digests the input message stream with the selected hash algorithm till it exhausts all message blocks. The function `Final` (`DAARijndael128Final`, `DAADESFinal`, and others) is designed to pad the partial message block into a final message block with the specified padding scheme, and further uses the hash algorithm to transform the final block into a message digest value.

The following example illustrates how the application code can apply an implemented DAA based on the Rijndael128 block cipher to digest the input message stream:

- Call the function [DAARijndael128GetSize](#) to get the size required to configure the `ippsDAARijndael128State` context.
- Ensure that the required memory space is properly allocated. With the allocated memory, call the function [DAARijndael128Init](#) to set up the initial context state.
- Keep calling the function [DAARijndael128Update](#) to digest incoming message stream in the queue till its completion.
- Call the function [DAARijndael128Final](#) for padding the final partial block applying Rijndael128 block cipher and transforming ciphertext block into resultant DAA value.
- Call the operating system memory free service function to release the `IppsDAARijndaelState` context.

DAADESGetSize

Gets the size of the `IppsDAADESState` context.

Syntax

```
IppStatus ippsDAADESGetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsDAADESState` context.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsDAADESState` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

DAADESInit

Initializes user-supplied memory as the `IppsDAADESState` context for future use.

Syntax

```
IppStatus ippsDAADESInit(const Ipp8u *pKey, IppsDAADESState *pCtx);
```

Parameters

pKey Pointer to the DES key.
pCtx Pointer to the `IppsDAADESState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAADESState` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operation.

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 <code>NULL</code> .

DAADESUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippSDAADESUpdate(const Ipp8u *pSrcMesg, int msglen, IppsDAADESState* pCtx);
```

Parameters

<code>pSrcMesg</code>	Pointer to the buffer containing a part or the whole message.
<code>msglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDAADESState</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAADESFinal

Completes computation of the DAC value.

Syntax

```
IppStatus ippSDAADESFinal(Ipp8u *pDAC, int dacLen, IppsDAADESState* pCtx);
```

Parameters

<code>pDAC</code>	Pointer to the DAC value.
<code>dacLen</code>	Specified length of the DAC.

*pCtx*Pointer to the `IppsDAADESState` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>dacLen</i> is less than 1 or greater than cipher's data block length.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAADESMessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippDAADESMessageDigest(const Ipp8u *pSrcMsg, int msgLen, const Ipp8u *pKey,
Ipp8u *pMAC, int macLen);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>msgLen</i> is less than zero, <i>macLen</i> is less than 1 or greater than cipher's data block length, <i>keyLen</i> value is illegal.

DAATDESGetSize

Gets the size of the IppsDAATDESState context.

Syntax

```
IppStatus ippsDAATDESGetSize(int *pSize);
```

Parameters

pSize Pointer to the IppsDAATDESState context.

Description

The function is declared in the `ippcp.h` file. The function gets the IppsDAATDESState context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

DAATDESInit

Initializes user-supplied memory as the IppsDAATDESState context for future use.

Syntax

```
IppStatus ippsDAATDESInit(const Ipp8u* pKey, const Ipp8u* pKey2, const Ipp8u* pKey3,  
IppsDAATDESState* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the TDES key.
<i>pKey2</i>	Pointer to the TDES key.
<i>pKey3</i>	Pointer to the TDES key.
<i>pCtx</i>	Pointer to the IppsDAATDESState context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the IppsDAATDESState context. In addition, DAATDESInit uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .

DAATDESUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippDAATDESUpdate(const Ipp8u *pSrcMesg, int mesglen, IppsDAATDESState *pCtx);
```

Parameters

<i>pSrcMesg</i>	Pointer to the buffer containing a part or the whole message.
<i>mesglen</i>	Length of the actual part of the message in bytes.
<i>pCtx</i>	Pointer to the <code>IppsDAATDESState</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAATDESFinal

Completes computation of the DAC value.

Syntax

```
IppStatus ippDAATDESFinal(Ipp8u *pDAC, int dacLen, IppsDAATDESState* pCtx);
```

Parameters

<i>pDAC</i>	Pointer to the DAC value.
<i>dacLen</i>	Specified length of the DAC.
<i>pCtx</i>	Pointer to the <code>IppsDAATDESState</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>macLen</code> is less than 1 or greater than cipher's data block length.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAATDESMessageDigest

Computes the DAC value of the message.

Syntax

```
ippStatus ippDAATDESMessageDigest(const Ipp8u *pSrcMsg, int msgLen, const Ipp8u *pKey1,
const Ipp8u *pKey2, const Ipp8u *pKey3, Ipp8u *pMAC, int macLen);
```

Parameters

<code>pSrcMsg</code>	Pointer to the input message.
<code>msgLen</code>	Message length in bytes.
<code>pKey1</code>	Pointer to the user-supplied key.
<code>pKey2</code>	Pointer to the user-supplied key.
<code>pKey3</code>	Pointer to the user-supplied key.
<code>pMAC</code>	Pointer to the resultant HMAC value.
<code>macLen</code>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key `pKey1`, `pKey2`, and `pKey3` of the specified key length `keyLen` and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code `pMAC` of the specified length `macLen`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>msgLen</code> is less than zero, <code>macLen</code> is less than 1 or greater than cipher's data block length, <code>keyLen</code> value is illegal.

DAARijndael128GetSize

Gets the size of the IppsDAARijndael128State context.

Syntax

```
IppStatus ippsDAARijndael128GetSize(int *pSize);
```

Parameters

pSize Pointer to the IppsDAARijndael128State context.

Description

The function is declared in the `ippcp.h` file. The function gets the IppsDAARijndael128State context size in bytes and stores it in *pSize*.

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`.

DAARijndael128Init, DAASafeRijndael128Init

Initialize user-supplied memory as IppsDAARijndael128State context for future use.

Syntax

```
IppStatus ippsDAARijndael128Init(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,  
IppsDAARijndael128State* pCtx);
```

```
IppStatus ippsDAASafeRijndael128Init(const Ipp8u* pKey, IppsRijndaelKeyLength keyLen,  
IppsDAARijndael128State* pCtx);
```

Parameters

pKey Pointer to the Rijndael128 key.
keyLen Key byte stream length in bytes defined by the IppsRijndaelKeyLength enumerator.
pCtx Pointer to the IppsDAARijndael128State context being initialized.

Description

These functions are declared in the `ippcp.h` file. Each function initializes the memory pointed by *pCtx* as the IppsDAARijndael128State context. In addition, each function uses the key to provide all necessary key material for both encryption and decryption operations. DAA based on the Rijndael128 cipher scheme uses the AES algorithm. Depending upon whether you wish to employ fast or safe implementation of the AES algorithm, call `DAARijndael128Init` or `DAASafeRijndael128Init`, respectively. For more information, see [Rijndael Functions](#) in chapter 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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>keyLen</i> is not equal to <code>IppsRijndaelKey128</code> , <code>IppsRijndaelKey192</code> , or <code>IppsRijndaelKey256</code> .

DAARijndael128Update

Digest the current input message stream of the specified length.

Syntax

```
ippStatus ippDAARijndael128Update(const Ipp8u *pSrcMsg, int msglen,
IppsDAARijndael128State* pCtx);
```

Parameters

<i>pSrcMsg</i>	Pointer to the buffer containing a part or the whole message.
<i>msglen</i>	Length of the actual part of the message in bytes.
<i>pCtx</i>	Pointer to the <code>IppsDAARijndael128State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAARijndael128Final

Completes computation of the DAC value.

Syntax

```
IppStatus ippDAARijndael128Final(Ipp8u *pDAC, int dacLen, IppsDAARijndael128State *pCtx);
```

Parameters

<i>pDAC</i>	Pointer to the DAC value.
<i>dacLen</i>	Specified length of the DAC.
<i>pCtx</i>	Pointer to the <code>IppsDAARijndael128State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>dacLen</i> is less than 1 or greater than cipher's data block length.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAARijndael128MessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippDAARijndael128MessageDigest(const Ipp8u *pSrcMsg, int msgLen, const Ipp8u *pKey, IppsRijndaelKeyLength keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length.
<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .

`ippStsLengthErr` Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

DAARijndael192GetSize

Gets the size of the IppsDAARijndael192State context.

Syntax

```
IppStatus ippDAARijndael192GetSize(int *pSize);
```

Parameters

pSize Pointer to the IppsDAARijndael192State context.

Description

The function is declared in the `ippcp.h` file. The function gets the IppsDAARijndael192State context size in bytes and stores it in *pSize*.

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`.

DAARijndael192Init

Initializes user-supplied memory as IppsDAARijndael192State context for future use.

Syntax

```
IppStatus ippRijndael192Init(const Ipp8u* pKey, IppsDAARijndaelKeyLength keyLen,  
IppsRijndael192State* pCtx);
```

Parameters

pKey Pointer to the Rijndael192 key.
keyLen Key byte stream length in bytes defined by the IppsRijndaelKeyLength enumerator.
pCtx Pointer to the IppsDAARijndael192State context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the IppsDAARijndael192State context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keyLen</code> is not equal to <code>IppsRijndaelKey128</code> , <code>IppsRijndaelKey192</code> , or <code>IppsRijndaelKey256</code> .

DAARijndael192Update

Digest the current input message stream of the specified length.

Syntax

```
IppStatus ippDAARijndael192Update(const Ipp8u *pSrcMesg, int mesglen,
IppsDAARijndael192State* pCtx);
```

Parameters

<code>pSrcMesg</code>	Pointer to the buffer containing a part or the whole message.
<code>mesglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDAARijndael192State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAARijndael192Final

Completes computation of the DAC value.

Syntax

```
IppStatus ippDAARijndael192Final(Ipp8u *pDAC, int dacLen, IppsDAARijndael192State *pCtx);
```

Parameters

<i>pDAC</i>	Pointer to the DAC value.
<i>dacLen</i>	Specified length of the DAC.
<i>pCtx</i>	Pointer to the <code>IppsDAARijndael192State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>dacLen</i> is less than 1 or greater than cipher's data block length.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAARijndael192MessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippDAARijndael192MessageDigest(const Ipp8u *pSrcMsg, int msgLen, const Ipp8u *pKey, IppsRijndaelKeyLength keyLen, Ipp8u *pDAC, int macLen);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length.
<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .

`ippStsLengthErr`

Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

DAARijndael256GetSize

Gets the size of the IppsDAARijndael256State context.

Syntax

```
IppStatus ippDAARijndael256GetSize(int *pSize);
```

Parameters

pSize Pointer to the IppsDAARijndael256State context.

Description

The function is declared in the `ippcp.h` file. The function gets the IppsDAARijndael256State context size in bytes and stores it in *pSize*.

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`.

DAARijndael256Init

Initializes user-supplied memory as IppsDAARijndael256State context for future use.

Syntax

```
IppStatus ippDAARijndael256Init(const Ipp8u* pKey, IppsDAARijndaelKeyLength keyLen, IppsRijndael256State* pCtx);
```

Parameters

pKey Pointer to the Rijndael256 key.
keyLen Key byte stream length in bytes defined by the IppsRijndaelKeyLength enumerator.
pCtx Pointer to the IppsDAARijndael256State being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the IppsDAARijndael256State context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>keyLen</code> is not equal to <code>IppsRijndaelKey128</code> , <code>IppsRijndaelKey192</code> , or <code>IppsRijndaelKey256</code> .

DAARijndael256Update

Digest the current input message stream of the specified length.

Syntax

```
ippStatus ippDAARijndael256Update(const Ipp8u *pSrcMsg, int msglen,
IppsDAARijndael256State* pCtx);
```

Parameters

<code>pSrcMsg</code>	Pointer to the buffer containing a part or the whole message.
<code>msglen</code>	Length of the actual part of the message in bytes.
<code>pCtx</code>	Pointer to the <code>IppsDAARijndael256State</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAARijndael256Final

Completes computation of the DAC value.

Syntax

```
IppStatus ippDAARijndael256Final(Ipp8u *pDAC, int dacLen, IppsDAARijndael256State* pCtx);
```

Parameters

<i>pDAC</i>	Pointer to the DAC value.
<i>dacLen</i>	Specified length of the DAC.
<i>pCtx</i>	Pointer to the <code>IppsDAARijndael256State</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>dacLen</i> is less than 1 or greater than cipher's data block length.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAARijndael256MessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippDAARijndael256MessageDigest(const Ipp8u *pSrcMsg, int msgLen, const Ipp8u *pKey, IppsRijndaelKeyLength keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length.
<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .

`ippStsLengthErr` Indicates an error condition if *msgLen* is less than zero, *macLen* is less than 1 or greater than cipher's data block length, *keyLen* value is illegal.

DAABlowfishGetSize

Gets the size of the IppsDAABlowfishState context.

Syntax

```
IppStatus ippDAABlowfishGetSize(int *pSize);
```

Parameters

pSize Pointer to the IppsDAABlowFishState context.

Description

The function is declared in the `ippcp.h` file. The function gets the IppsDAABlowFishState context size in bytes and stores it in *pSize*.

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`.

DAABlowfishInit

Initializes user-supplied memory as IppsDAABlowfishState context for future use.

Syntax

```
IppStatus ippDAABlowfishInit(const Ipp8u* pKey, int keylen, IppsDAABlowfishState* pCtx);
```

Parameters

pKey Pointer to the DAABlowfish key.
keylen Key byte stream length in bytes.
pCtx Pointer to the IppsDAABlowfishState context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the IppsDAABlowfishState context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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`.

`ippStsLengthErr` Indicates an error condition if *keylen* is less than 1 or greater than 56.

DAABlowfishUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippDAABlowfishUpdate(const Ipp8u *pSrcMesg, int mesglen, IppsDAABlowfishState* pCtx);
```

Parameters

pSrcMesg Pointer to the buffer containing a part or the whole message.
mesglen Length of the actual part of the message in bytes.
pCtx Pointer to the `IppsDAABlowfishState` context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

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`.
`ippStsLengthErr` Indicates an error condition if the input data stream length is less than zero.
`ippStsContextMatchErr` Indicates an error condition if the context parameter does not match the operation.

DAABlowfishFinal

Completes computation of the DAC value.

Syntax

```
IppStatus ippDAABlowfishFinal(Ipp8u *pDAC, int dacLen, IppsDAABlowfishState* pCtx);
```

Parameters

pDAC Pointer to the DAC value.
dacLen Specified length of the DAC.
pCtx Pointer to the `IppsDAABlowfishState` context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>dacLen</i> is less than 1 or greater than cipher's data block length.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAABlowfishMessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippDAABlowfishMessageDigest(const Ipp8u *pSrcMsg, int msgLen, const Ipp8u *pKey,
int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length.
<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>msgLen</i> is less than zero, <i>macLen</i> is less than 1 or greater than cipher's data block length, <i>keyLen</i> value is illegal.

DAATwofishGetSize

Gets the size of the `IppsDAATwofishState` context.

Syntax

```
IppStatus ippsDAATwofishGetSize(int *pSize);
```

Parameters

pSize Pointer to the `IppsDAATwoFishState` context.

Description

The function is declared in the `ippcp.h` file. The function gets the `IppsDAATwoFishState` context size in bytes and stores it in *pSize*.

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 <code>NULL</code> .

DAATwofishInit

*Initializes user-supplied memory as
`IppsDAATwofishState` context for future use.*

Syntax

```
IppStatus ippsDAATwofishInit(const Ipp8u* pKey, int keylen, IppsDAATwofishState* pCtx);
```

Parameters

<i>pKey</i>	Pointer to the <code>DAATwofish</code> key.
<i>keylen</i>	Key byte stream length in bytes.
<i>pCtx</i>	Pointer to the <code>IppsDAATwofishState</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDAATwofishState` context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>keylen</i> is less than 1 or greater than 32.

DAATwofishUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippSDAATwofishUpdate(const Ipp8u *pSrcMesg, int msglen, IppsDAATwofishState* pCtx);
```

Parameters

<i>pSrcMesg</i>	Pointer to the buffer containing a part or the whole message.
<i>msglen</i>	Length of the actual part of the message in bytes.
<i>pCtx</i>	Pointer to the <code>IppsDAATwofishState</code> context.

Description

This function is declared in the `ippcp.h` file. The function digests the current input message stream of the specified length. The function first integrates the previous partial message block with the input message stream, and then partitions them into multiple message blocks (as specified by applied hash algorithm) with a possible additional partial block. For each message block, the function uses the selected block cipher to transform the block of plaintext into a new chaining digest value.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the input data stream length is less than zero.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAATwofishFinal

Completes computation of the DAC value.

Syntax

```
IppStatus ippSDAATwofishFinal(Ipp8u *pDAC, int dacLen, IppsDAATwofishState* pCtx);
```

Parameters

<i>pDAC</i>	Pointer to the DAC value.
<i>dacLen</i>	Specified length of the DAC.
<i>pCtx</i>	Pointer to the <code>IppsDAATwofishState</code> context.

Description

This function is declared in the `ippcp.h` file. The function completes calculation of the digest value and stored result into the specified *pDAC* memory.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>dacLen</i> is less than 1 or greater than cipher's data block length.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

DAATwofishMessageDigest

Computes the DAC value of the message.

Syntax

```
IppStatus ippSDAATwofishMessageDigest(const Ipp8u *pSrcMsg, int msgLen, const Ipp8u *pKey,
int keyLen, Ipp8u *pMAC, int macLen);
```

Parameters

<i>pSrcMsg</i>	Pointer to the input message.
<i>msgLen</i>	Message length in bytes.
<i>pKey</i>	Pointer to the user-supplied key.
<i>keyLen</i>	Key length.
<i>pMAC</i>	Pointer to the resultant HMAC value.
<i>macLen</i>	Specified HMAC length.

Description

This function is declared in the `ippcp.h` file. The function takes the input key *pKey* of the specified key length *keyLen* and applied keyed hash-based message authentication code scheme to transform the input message into the respective message authentication code *pMAC* of the specified length *macLen*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>msgLen</i> is less than zero, <i>macLen</i> is less than 1 or greater than cipher's data block length, <i>keyLen</i> value is illegal.

Public Key Cryptography Functions

Big Number Arithmetic

This section describes primitives for performing arithmetic operations with integer big numbers of variable length. The full list of functions for big number arithmetic is given in [Table “Intel IPP Big Number Arithmetic Functions”](#).

Intel IPP Big Number Arithmetic Functions

Function Base Name	Operation
Unsigned Big Number Arithmetic Functions	
Add_BNU	Adds two unsigned integer big numbers of the same length.
Sub_BNU	Subtracts one integer big number from another integer big number of the same length.
MulOne_BNU	Multiplies unsigned integer big number by 32-bit unsigned integer.
MACOne_BNU_I	Computes multiplication of unsigned integer big number by 32-bit integer and accumulates the result with another integer big number.
Mul_BNU4	Multiplies two unsigned integers of 4*32 bits.
Mul_BNU8	Multiplies two unsigned integers of 8*32 bits.
Div_64u32u	Divides unsigned 64-bit integer by unsigned 32-bit integer.
Sqr_32u64u	Computes the square of 32-bit words in the input array.
Sqr_BNU4	Computes the square of an unsigned integer big number of 4*32 bits.
Sqr_BNU8	Computes the square of an unsigned integer big number of 8*32 bits.
SetOctString_BNU	Converts octet string into unsigned integer big number.
GetOctString_BNU	Converts unsigned integer big number into octet string.
Signed Big Number Arithmetic Functions	
BigNumGetSize	Gets the size of the <code>IppsBigNumState</code> context.
BigNumInit	Initializes context and partitions allocated buffer.
Set_BN	Defines the sign and value of the context.
SetOctString_BN	Converts octet string into a positive Big Number.
GetSize_BN	Returns the maximum length of the integer big number the structure can store.
Get_BN	Extracts the sign and value of the integer big number from the input structure.
ExtGet_BN	Extracts the specified combination of the sign, data length, and value characteristics of the integer big number from the input structure.
Ref_BN	Extracts the main characteristics of the integer big number from the input structure.
GetOctString_BN	Converts a positive Big Number into octet String.
Cmp_BN	Compares two Big Numbers.

Function Base Name	Operation
CmpZero_BN	Checks the value of the input data field.
Add_BN	Adds two integer big numbers.
Sub_BN	Subtracts one integer big number from another.
Mul_BN	Multiplies two integer big numbers.
MAC_BN_I	Multiplies two integer big numbers and accumulates the result with the third integer big number.
Div_BN	Divides one integer big number by another.
Mod_BN	Computes modular reduction for input integer big number with respect to specified modulus.
Gcd_BN	Computes the greatest common divisor.
ModInv_BN	Computes multiplicative inverse of a positive integer big number with respect to specified modulus.

The magnitude of an integer big number is specified by an array of unsigned integer data type `Ipp32u rp[length]` and corresponds to the mathematical value

$$r = \sum_{0 \leq i < \text{length}} rp[i] \times 2^{32i}.$$

This section uses the following definition for the sign of an integer big number:

```
typedef enum {
    IppsBigNumNEG=0,
    IppsBigNumPOS=1
} IppsBigNumSGN;
```

The functions described in this section use the context `IppsBigNumState` to serve as an operational vehicle that carries not only the sign and value of the data, but also a sufficient working buffer reserved for various arithmetic operations. The length of the context `IppsBigNumState` is defined as the length of the data carried by the structure and the size of the context `IppsBigNumState` is therefore defined as the maximal length of the data that this operational vehicle can carry.



NOTE. In all unsigned big number arithmetic functions described below, integers pointed to by *a*, *b*, and *r* are all of (*n**32) bits.

Add_BNU

Adds two unsigned integer big numbers of the same length.

Syntax

```
IppStatus ippsAdd_BNU(const Ipp32u *a, const Ipp32u *b, Ipp32u *r, int n, Ipp32u * carry);
```

Parameters

a First unsigned integer big number of *n**32 bits.

<i>b</i>	Second unsigned integer big number of $n \cdot 32$ bits.
<i>n</i>	Size specified for the input parameters <i>a</i> and <i>b</i> and the result parameter <i>r</i> . The size is expressed in the number of 32-bit words.
<i>r</i>	On output, addition result.
<i>carry</i>	On output, addition carry. The possible value is 0 or 1.

Description

This function is declared in the `ippcp.h` file. The function adds two unsigned integer big numbers of the same length and returns the result of the operation with a possible carry.

The following pseudocode represents this function:

$$(carry | (*r)) \leftarrow (*a) + (*b).$$

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>n</i> is less than or equal to 0.

Sub_BNU

Subtracts one integer big number from another integer big number of the same length.

Syntax

```
IppStatus ippSub_BNU(const Ipp32u *a, const Ipp32u *b, Ipp32u *r, int n, Ipp32u * carry);
```

Parameters

<i>a</i>	First unsigned integer big number of $n \cdot 32$ bits.
<i>b</i>	Second unsigned integer big number of $n \cdot 32$ bits.
<i>n</i>	Size specified for the input parameters <i>a</i> and <i>b</i> and the result parameter <i>r</i> . The size is expressed in the number of 32-bit words.
<i>r</i>	Subtraction result.
<i>carry</i>	Subtraction borrow. Possible value is 0 or 1.

Description

This function is declared in the `ippcp.h` file. The function subtracts one integer big number from another big-number integer of the same length and returns the result of the operation with a possible borrow returned as the *carry* argument.

The following pseudocode represents this function:

$$(*r) \leftarrow (*a) - (*b);$$

$carry = ((*a) < (*b)).$

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if n is less than or equal to 0.

MulOne_BNU

Multiplies unsigned integer big number by 32-bit unsigned integer.

Syntax

```
IppStatus ippMulOne_BNU(const Ipp32u *a, Ipp32u *r, int n, Ipp32u w, Ipp32u *carry);
```

Parameters

a	Unsigned integer big number of $n * 32$ bits.
w	Unsigned long integer of 32 bits serving as multiplier for the operation.
n	Size specified for the input parameter a and the result parameter r . The size is expressed in the number of 32-bit words.
r	Multiplication result.
$carry$	Multiplication carry.

Description

This function is declared in the `ippcp.h` file. The function computes the multiplication of an unsigned integer big number a by a 32-bit unsigned integer w . The function returns the result to the length array r , and the carry of the result to $carry$.

The following pseudocode represents this function:

$(carry | (*r)) \leftarrow (*a)w.$

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if n is less than or equal to 0.

MACOne_BNU_I

Computes multiplication of unsigned integer big number by 32-bit integer and accumulates the result with another integer big number.

Syntax

```
IppStatus ippsMACOne_BNU_I(const Ipp32u *a, Ipp32u *r, int n, Ipp32u w, Ipp32u *carry);
```

Parameters

<i>a</i>	Multiplicand, an unsigned integer big number.
<i>w</i>	32-bit unsigned long integer multiplier.
<i>n</i>	Size specified for the input parameters <i>a</i> and the result parameter <i>r</i> . The size is expressed in the number of 32-bit words.
<i>r</i>	Unsigned integer big number accumulator.
<i>carry</i>	Operation carry.

Description

This function is declared in the `ippcp.h` file. The function computes the multiplication of an unsigned integer big number *a* by a 32-bit integer *w* and accumulates the result with another integer big number *r* of same length. The result of the operation is returned to *r*, and the carry of the result is returned as *carry*.

The following pseudocode represents this function:

$$(carry | (*r)) \leftarrow (*r) + (*a)_w.$$

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>n</i> is less than or equal to 0.

Mul_BNU4

*Multiplies two unsigned integers of 4*32 bits.*

Syntax

```
IppStatus ippsMul_BNU4(const Ipp32u *a, const Ipp32u *b, Ipp32u *r);
```

Parameters

<i>a</i>	Multiplicand, an integer big number of 4*32 bits.
<i>w</i>	Multiplier, an integer big number of 4*32 bits.
<i>r</i>	Multiplication result of 8*32 bits.

Description

This function is declared in the `ippcp.h` file. The function multiplies two unsigned integers of 4*32 bit and returns the result of 8*32 bits.

The following pseudocode represents this function:

$$(*r) \leftarrow (*a) (*b).$$

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 <code>NULL</code> .

Mul_BNU8

*Multiplies two unsigned integers of 8*32 bits.*

Syntax

```
IppStatus ippMul_BNU8(const Ipp32u *a, const Ipp32u *b, Ipp32u *r);
```

Parameters

<i>a</i>	Multiplicand, an integer big number of 8*32 bits.
<i>b</i>	Multiplier, an integer big number of 8*32 bits.
<i>r</i>	Multiplication result of 16*32 bits.

Description

This function is declared in the `ippcp.h` file. The function multiplies two unsigned integers of 8*32 bit and returns the result of 16*32 bits.

The following pseudocode represents this function:

$$(*r) \leftarrow (*a) (*b).$$

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 <code>NULL</code> .

Div_64u32u

Divides unsigned 64-bit integer by unsigned 32-bit integer.

Syntax

```
IppStatus ippDiv_64u32u(Ipp64u a, Ipp32u b, Ipp32u *q, Ipp32u *r);
```

Parameters

a	Dividend of 64 bits.
b	Divisor of 32 bits.
q	Quotient of 32 bits.
r	Remainder of 32 bits.

Description

This function is declared in the `ippcp.h` file. The function divides a 64-bit unsigned integer dividend by a 32-bit unsigned divisor and returns the quotient and remainder.

The following pseudocode represents this function:

$$a = b * q + r.$$

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 <code>NULL</code> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if the quotient is not 32 bits. In this case the result is undefined.
<code>ippStsDivByZeroErr</code>	Indicates an error condition if zero divisor is used.

Sqr_32u64u

Computes the square of 32-bit words in the input array.

Syntax

```
IppStatus ippSqr_32u64u(const Ipp32u *src, int n, Ipp64u *dst);
```

Parameters

src	Array of 32-bit words.
n	Input array size.
dst	Pointer to the result.

Description

This function is declared in the `ippcp.h` file. The function computes the square of each unsigned 32-bit long word in the input array. The result of the operation is stored in the array of 64-bit unsigned integers.

The following pseudocode represents this function:

$$dst[i] \leftarrow src[i] * src[i], \text{ where } i = 0, 1, 2, \dots, n-1.$$

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 <code>NULL</code> .

`ippStsLengthErr`Indicates an error condition if n is less than or equal to 0.

Sqr_BNU4

*Computes the square of an unsigned integer big number of 4*32 bits.*

Syntax

```
IppStatus ippSqr_BNU4(const Ipp32u *a, Ipp32u *r);
```

Parameters

a	Multiplicand of 4*32 bits.
r	Square operation result of 8*32 bits.

Description

This function is declared in the `ippcp.h` file. The function computes the square of an unsigned integer big number of 4*32 bits and stores the result in the memory.

The following pseudocode represents this function:

$$(*r) \leftarrow (*a) (*a).$$

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 <code>NULL</code> .

Sqr_BNU8

*Computes the square of an unsigned integer big number of 8*32 bits.*

Syntax

```
IppStatus ippSqr_BNU8(const Ipp32u *a, Ipp32u *r);
```

Parameters

a	Multiplicand of 8*32 bits.
r	Square operation result of 16*32 bits.

Description

This function is declared in the `ippcp.h` file. The function computes the square of an unsigned integer big number of 8*32 bits and stores the result in the memory.

The following pseudocode represents this function:

$$(*r) \leftarrow (*a) (*a).$$

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 <code>NULL</code> .

SetOctString_BNU

Converts octet string into unsigned integer big number.

Syntax

```
IppStatus ippSetOctString_BNU(const Ipp8u* pOctStr, int strLen, Ipp32u* pBNU, int* pBNUsize);
```

Parameters

<code>pOctStr</code>	Pointer to the input octet string <code>OctStr</code> .
<code>strLen</code>	Length of the octet string.
<code>pBNU</code>	Pointer to the unsigned integer big number BNU.
<code>pBNUsize</code>	Pointer to the size (in 32-bit items) of the unsigned integer big number.

Description

The function is declared in the `ippcp.h` file. This function converts octet string into unsigned integer big number.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if specified <code>strLen</code> is less than 1.
<code>ippStsSizeErr</code>	Indicates an error condition if specified <code>*pBNUsize</code> is not sufficient for keeping actual <code>strLen</code> .

GetOctString_BNU

Converts unsigned integer big number into octet string.

Syntax

```
IppStatus ippGetOctString_BNU(const Ipp32u* pBNU, int bnuSize, Ipp8u* pOctStr, int strLen);
```

Parameters

<code>pBNU</code>	Pointer to the source unsigned integer big number BNU.
<code>bnuSize</code>	Unsigned integer big number size (in 32-bit items)
<code>pOctStr</code>	Pointer to the source octet string <code>OctStr</code> .
<code>strLen</code>	Length of the octet string.

Description

The function is declared in the `ippcp.h` file. This function converts unsigned integer big number BNU into octet string `OctStr`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if specified <code>bnuSize</code> is less than 1.
<code>ippStsRangeErr</code>	Indicates an error condition if 256^{strLen} is less than the unsigned integer big number.

BigNumGetSize

Gets the size of the `IppsBigNumState` context in bytes.

Syntax

```
IppStatus ippBigNumGetSize(int length, int *size);
```

Parameters

<code>length</code>	Integer big number length in <code>Ipp32u</code> .
<code>size</code>	Size of the buffer in bytes required for initialization.

Description

This function is declared in the `ippcp.h` file. The function specifies the buffer size required to define a structuralized working buffer of the context `IppsBigNumState` for the storage and operations on an integer big number in bytes.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>length</code> is less than or equal to 0.

BigNumInit

Initializes context and partitions allocated buffer.

Syntax

```
IppStatus ippBigNumInit(int length, IppsBigNumState *b);
```

Parameters

<code>length</code>	Size of the big number for the context initialization.
---------------------	--

b Pointer to the supplied buffer used to store the initialized context
IppsBigNumState.

Description

This function is declared in the `ippcp.h` file. The function initializes the context `IppsBigNumState` using the specified buffer space and partitions the given buffer to store and execute arithmetic operations on an integer big number of the *length* size.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>length</i> is less than or equal to 0.

Set_BN

Defines the sign and value of the context.

Syntax

```
IppStatus ippSet_BN(IppsBigNumSGN sgn, int length, const Ipp32u *data, IppsBigNumState *x);
```

Parameters

<i>sgn</i>	Sign of <code>IppsBigNumState *x</code> .
<i>length</i>	Array length of the input data.
<i>data</i>	Data array.
<i>x</i>	On output, the context <code>IppsBigNumState</code> updated with the input data.

Description

This function is declared in the `ippcp.h` file. The function defines the sign and value for `IppsBigNumState *x` with the specified inputs `IppsBigNumSGN sgn` and `const Ipp32u *data`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>length</i> is less than or equal to 0.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <i>length</i> is more than the size of <code>IppsBigNumState *x</code> .
<code>ippStsBadArgErr</code>	Indicates an error condition if the big number is set to zero with the negative sign.

Example

The code example below shows how to create a big number.

```

IppsBigNumState* New_BN(int size, const Ipp32u* pData=0){
    // get the size of the Big Number context
    int ctxSize;
    ippsBigNumGetSize(size, &ctxSize);
    // allocate the Big Number context
    IppsBigNumState* pBN = (IppsBigNumState*) (new Ipp8u [ctxSize] );
    // and initialize one
    ippsBigNumInit(size, pBN);
    // if any data was supplied, then set up the Big Number value
    if(pData)
        ippsSet_BN(IppsBigNumPOS, size, pData, pBN);
    // return pointer to the Big Number context for future use
    return pBN;
}

```

SetOctString_BN

Converts octet string into a positive Big Number.

Syntax

```
IppStatus ippsSetOctString_BN(const Ipp8u* pOctStr, int strLen, IppsBigNumState* pBN);
```

Parameters

<i>pOctStr</i>	Pointer to the input octet string.
<i>strLen</i>	Octet string length in bytes.
<i>pBN</i>	Pointer to the context of the output Big Number.

Description

The function is declared in the `ippcp.h` file. This function converts octet string into a positive Big Number.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if specified <i>strLen</i> is less than 1.

`ippStsSizeErr` Indicates an error condition if insufficient space has been reserved for Big Number.

Example

The code example below shows how to create a big number from a string.

```
void Set_BN_sample(void){
    // desired value of Big Number is 0x123456789abcdef0fedcba9876543210
    Ipp8u desiredBNvalue[] = "\x12\x34\x56\x78\x9a\xbc\xde\x0"
                             "\xfe\xdc\xba\x98\x76\x54\x32\x10";

    // estimate required size of Big Number

    //int size = (sizeof(desiredBNvalue)+3)/4;
    int size = (sizeof(desiredBNvalue)-1+3)/4;
    // and create new (and empty) one
    IppsBigNumState* pBN = New_BN(size);
    // set up the value from the string
    ippSetOctString_BN(desiredBNvalue, sizeof(desiredBNvalue)-1, pBN);
    Type_BN("Big Number value is:\n", pBN);
}
```

GetSize_BN

Returns the maximum length of the integer big number the structure can store.

Syntax

```
IppStatus ippGetSize_BN(const IppsBigNumState *b, int *size);
```

Parameters

b Integer big number of the data type `IppsBigNumState`.
size Maximum length of the integer big number.

Description

This function is declared in the `ippcp.h` file. The function evaluates the working buffer assigned to the context `IppsBigNumState` and returns the size of the structure to indicate the maximum length of the integer big number that the structure can store.

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`.

Get_BN

Extracts the sign and value of the integer big number from the input structure.

Syntax

```
IppStatus ippsGet_BN(IppsBigNumSGN *sgn, int *length, Ipp32u *data, const IppsBigNumState *x);
```

Parameters

<i>sgn</i>	Sign of IppsBigNumState *x.
<i>length</i>	Array length of the input data.
<i>data</i>	Data array.
<i>x</i>	Integer big number of the context IppsBigNumState.

Description

This function is declared in the `ippcp.h` file. The function extracts the sign and value of the integer big number from the input structure.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

ExtGet_BN

Extracts the specified combination of the sign, data length, and value characteristics of the integer big number from the input structure.

Syntax

```
IppStatus ippsExtGet_BN(IppsBigNumSGN *pSgn, int *pLengthInBits, Ipp32u *pData, const IppsBigNumState *pBN);
```

Parameters

<i>pSgn</i>	Pointer to the sign of IppsBigNumState *pBN.
<i>pLengthInBits</i>	Pointer to the length of *pData in bits.
<i>pData</i>	Pointer to the data array.
<i>pBN</i>	Pointer to the integer big number context IppsBigNumState.

Description

This function is declared in the `ippcp.h` file. For the integer big number from the input structure, the function extracts the specified combination of the following characteristics: sign, data length, and value. The function is similar to the [Get_BN](#) function but more flexible, because any target pointer (*pSgn*, *pLengthInBits*, and/or *pData*) may be `NULL`, in which case the appropriate big number characteristic will not be extracted. For example,

`ippExtGet_BN(&sgn, 0,0, pBN);` extracts only the sign

`ippExtGet_BN(0, &dataLen, 0, pBN);` extracts only the data length

`ippExtGet_BN(&sgn, &dataLen, 0, pBN);` extracts the sign and data length

`ippExtGet_BN(0,0,0, pBN);` does nothing

`ippExtGet_BN(&sgn, &dataLen, pData, pBN);` does exactly what [Get_BN](#) does.

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 integer big number of the context is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

Ref_BN

Extracts the main characteristics of the integer big number from the input structure.

Syntax

```
IppStatus ippRef_BN(IppsBigNumSGN *sgn, int *bitSize, const Ipp32u **pData, const IppsBigNumState *x);
```

Parameters

<i>sgn</i>	Sign of <code>IppsBigNumState *x</code> .
<i>bitSize</i>	Length of the integer big number in bits.
<i>pData</i>	Pointer to the data array.
<i>x</i>	Integer big number of the context <code>IppsBigNumState</code> .

Description

This function is declared in the `ippcp.h` file. The function extracts from the input structure the main characteristics of the integer big number: sign, length, and pointer to the data array. You can extract either the entire set or any subset of these characteristics. To turn off extraction of a particular characteristic, set the appropriate function parameter to `NULL`.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

GetOctString_BN

Converts a positive Big Number into octet String.

Syntax

```
IppStatus ippGetOctString_BN(const Ipp8u* pOctStr, int strLen, const IppsBigNumState* pBN);
```

Parameters

<code>pOctStr</code>	Pointer to the input octet string.
<code>strLen</code>	Octet string length in bytes.
<code>pBN</code>	Pointer to the context of the input Big Number.

Description

The function is declared in the `ippcp.h` file. This function converts a positive Big Number into the octet string.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if specified <code>pOctStr</code> is insufficient in length.
<code>ippStsRangeErr</code>	Indicates an error condition if Big Number is negative.

Example

The code example below types a big number.

```
void Type_BN(const char* pMsg, const IppsBigNumState* pBN){
    // size of Big Number
    int size;
    ippsGetSize_BN(pBN, &size);
    // extract Big Number value and convert it to the string presentation
    Ipp8u* bnValue = new Ipp8u [size*4];
    ippsGetOctString_BN(bnValue, size*4, pBN);
    // type header
    if(pMsg)
        cout <<pMsg;
    // type value
    for(int n=0; n<size*4; n++)
        cout<<hex <<(int)bnValue[n];
    cout <<endl;
    delete [] bnValue;
}
```

Cmp_BN

Compares two Big Numbers.

Syntax

```
IppStatus ippsCmp_BN(const IppsBigNumState *pA, const IppsBigNumState *pB, Ipp32u *pResult);
```

Parameters

<i>pA</i>	Pointer to the context of the Big Number A.
<i>pB</i>	Pointer to the context of the Big Number B.
<i>pResult</i>	Pointer to the result of the comparison.

Description

The function is declared in `ippcp.h` file. This function compares Big Numbers A and B and sets up the result according to the following conditions:

- if $A=B$, then *pResult* = `IS_ZERO`
- if $A > B$, then *pResult* = `GREATER_THAN_ZERO`
- if $A < B$, then *pResult* = `LESS_THAN_ZERO`

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

CmpZero_BN

Checks the value of the input data field.

Syntax

```
IppStatus ippCmpZero_BN(const IppsBigNumState *b, Ipp32u *result);
```

Parameters

<code>b</code>	Integer big number of the data type <code>IppsBigNumState</code> .
<code>result</code>	Indicates whether the input integer big number is positive, negative, or zero.

Description

This function is declared in the `ippcp.h` file. The function scans the data field of the input `const IppsBigNumState *b` and returns

- `IS_ZERO` if the value held by `IppsBigNumState *b` is zero
- `GREATER_THAN_ZERO` if the input is more than zero
- `LESS_THAN_ZERO` if the input is less than zero.

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 <code>NULL</code> .

Add_BN

Adds two integer big numbers.

Syntax

```
IppStatus ippAdd_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *r);
```

Parameters

<code>a</code>	First integer big number of the data type <code>IppsBigNumState</code> .
<code>b</code>	Second integer big number of the data type <code>IppsBigNumState</code> .
<code>r</code>	Addition result.

Description

This function is declared in the `ippcp.h` file. The function adds two integer big numbers regardless of their signs and sizes and returns the result of the operation.

The following pseudocode represents this function:

$$(*r) \leftarrow (*a) + (*b).$$

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 <code>NULL</code> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if the size of <code>r</code> is smaller than the resulting data length.



NOTE. The function executes only under the condition that size of `IppsBigNumState * r` is not less than either the length of `IppsBigNumState * a` or that of `IppsBigNumState * b`.

Example

The code example below adds big numbers.

```
void Add_BN_sample(void){
    // define and set up Big Number A
    const Ipp32u bnuA[] = {0x01234567,0x9abcdeff,0x11223344};
    IppsBigNumState* bnA = New_BN(sizeof(bnuA)/sizeof(Ipp32u));
    // define and set up Big Number B
    const Ipp32u bnuB[] = {0x76543210,0xfedcabee,0x44332211};
    IppsBigNumState* bnB = New_BN(sizeof(bnuB)/sizeof(Ipp32u), bnB);
    // define Big Number R
    int sizeR = max(sizeof(bnuA), sizeof(bnuB));
    IppsBigNumState* bnR = New_BN(1+sizeR/sizeof(Ipp32u));
    // R = A+B
    ippAdd_BN(bnA, bnB, bnR);
    // type R
    Type_BN("R=A+B:\n", bnR);
    delete [] (Ipp8u*)bnA;
    delete [] (Ipp8u*)bnB;
    delete [] (Ipp8u*)bnR;
}
```

Sub_BN

Subtracts one integer big number from another.

Syntax

```
IppStatus ippsSub_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *r);
```

Parameters

<i>a</i>	First integer big number of the data type <code>IppsBigNumState</code> .
<i>b</i>	Second integer big number of the data type <code>IppsBigNumState</code> .
<i>r</i>	Subtraction result.

Description

This function is declared in the `ippcp.h` file. The function subtracts one integer big number from another regardless of their signs and sizes and returns the result of the operation.

The following pseudocode represents this function:

$$(*r) \leftarrow (*a) - (*b).$$

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 <code>NULL</code> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>IppsBigNumState *r</code> is smaller than the result data length.



NOTE. The function executes only under the condition that size of `IppsBigNumState *r` is not less than either the length of `IppsBigNumState *a` or that of `IppsBigNumState *b`.

Mul_BN

Multiplies two integer big numbers.

Syntax

```
IppStatus ippsMul_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *r);
```

Parameters

<i>a</i>	Multiplicand of <code>IppsBigNumState</code> .
<i>b</i>	Multiplier of <code>IppsBigNumState</code> .
<i>r</i>	Multiplication result.

Description

This function is declared in the `ippcp.h` file. The function multiplies an integer big number by another integer big number regardless of their signs and sizes and returns the result of the operation.

The following pseudocode represents this function:

$$r \leftarrow a * b.$$

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 <code>NULL</code> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>IppsBigNumState *r</code> is smaller than the result data length.



NOTE. The function executes only under the condition that the size `IppsBigNumState *r` is not less than the sum of the lengths of `IppsBigNumState *a` or that of `IppsBigNumState *b` minus one.

MAC_BN_I

Multiplies two integer big numbers and accumulates the result with the third integer big number.

Syntax

```
IppStatus ippMAC_BN_I(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *r);
```

Parameters

<code>a</code>	Multiplicand of <code>IppsBigNumState</code> .
<code>b</code>	Multiplier of <code>IppsBigNumState</code> .
<code>r</code>	Multiplication result.

Description

This function is declared in the `ippcp.h` file. The function multiplies one integer big number by another and accumulates the result with the third input integer big number regardless of their signs and sizes. The function subsequently returns the result of the operation.

The following pseudocode represents this function:

$$r \leftarrow r + a * b.$$

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 <code>NULL</code> .

`ippStsOutOfRangeErr` Indicates an error condition if `IppsBigNumState *r` is smaller than the result data length.



NOTE. The function executes only under the condition that the size `IppsBigNumState *r` is not less than the sum of the lengths of `IppsBigNumState *a` or that of `IppsBigNumState *b` minus one.

Div_BN

Divides one integer big number by another.

Syntax

```
IppStatus ippDiv_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *q,
IppsBigNumState *r);
```

Parameters

<code>a</code>	Dividend of <code>IppsBigNumState</code> .
<code>b</code>	Divisor of <code>IppsBigNumState</code> .
<code>q</code>	Quotient of <code>IppsBigNumState</code> .
<code>r</code>	Remainder of <code>IppsBigNumState</code> .

Description

This function is declared in the `ippcp.h` file. The function divides an integer big number dividend by another integer big number regardless of their signs and sizes and returns the quotient of the division and the respective remainder.

The following pseudocode represents this function:

$$q \leftarrow a/b$$

$$r \leftarrow a - b*q.$$

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 <code>NULL</code> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>IppsBigNumState *r</code> is smaller than the length of <code>IppsBigNumState *b</code> or when the size of <code>IppsBigNumState *q</code> is smaller than the quotient result data length.
<code>ippStsDivByZeroErr</code>	Indicates an error condition if the zero divisor is attempted.



NOTE. The size of `IppsBigNumState *q` should not be less than $(\text{length of } *a) - (\text{length of } *b) + 1$, and the size of `IppsBigNumState *r` should be no less than the length of `IppsBigNumState *b`.

Mod_BN

Computes modular reduction for input integer big number with respect to specified modulus.

Syntax

```
IppStatus ippsMod_BN(IppsBigNumState *a, IppsBigNumState *m, IppsBigNumState *r);
```

Parameters

<i>a</i>	Integer big number of IppsBigNumState.
<i>m</i>	Modulus integer of IppsBigNumState.
<i>r</i>	Modular reduction result.

Description

This function is declared in the `ippcp.h` file. The function computes the modular reduction for an input integer big number with respect to the modulus specified by a positive integer big number and returns the modular reduction result in the range of $[0, (m-1)]$.

The following pseudocode represents this function:

$r \leftarrow a \bmod m.$

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 <code>NULL</code> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if IppsBigNumState * <i>r</i> is smaller than the length of IppsBigNumState * <i>m</i> .
<code>ippStsBadModulusErr</code>	Indicates an error condition if the modulus IppsBigNumState * <i>m</i> is not a positive integer.



NOTE. The size of IppsBigNumState **r* should not be less than the length of IppsBigNumState **m*.

Gcd_BN

Computes greatest common divisor.

Syntax

```
IppStatus ippsGcd_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *g);
```

Parameters

<i>a</i>	First integer big number of IppsBigNumState.
<i>b</i>	Second integer big number of IppsBigNumState.

g Greatest common divisor to a and b .

Description

This function is declared in the `ippcp.h` file. The function computes the greatest common divisor (GCD) for two positive integer big numbers.

The following pseudocode represents this function:

$g \leftarrow gcd(a, b).$

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 <code>NULL</code> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>IppsBigNumState *g</code> is smaller than the length of <code>IppsBigNumState *a</code> or <code>IppsBigNumState *b</code> .



NOTE. The size of `IppsBigNumState *g` should not be less than either the length of `IppsBigNumState *a` and `IppsBigNumState *b`.

ModInv_BN

Computes multiplicative inverse of a positive integer big number with respect to specified modulus.

Syntax

```
ippStatus ippModInv_BN(IppsBigNumState *e, IppsBigNumState *m, IppsBigNumState *d);
```

Parameters

e	Integer big number of <code>IppsBigNumState</code> .
m	Modulus integer of <code>IppsBigNumState</code> .
d	Multiplicative inverse.

Description

This function is declared in the `ippcp.h` file. The function uses the extended Euclidean algorithm to compute the multiplicative inverse of a given positive integer big number e with respect to the modulus specified by another positive integer big number m , where $gcd(e, m) = 1$.

The following pseudocode represents this function:

compute d such that $d * e = 1 \bmod m$.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsBadArgErr</code>	Indicates an error condition if e is less than or equal to 0.

<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is <code>NULL</code> .
<code>ippStsBadModulusErr</code>	Indicates an error condition if the modulus e is more than m , or $\gcd(e, m)$ is more than 1, or m is less than or equal to 0.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>IppsBigNumState *d</code> is smaller than the length of <code>IppsBigNumState *m</code> .



NOTE. The size of `IppsBigNumState *d` should not be less than the length of `IppsBigNumState *m`.

Montgomery Reduction Scheme Functions

This section describes Montgomery reduction scheme functions. The full list of these functions is given in [Table 5-2](#).

Table 5-2 Intel IPP Montgomery Reduction Scheme Functions

Function Base Name	Operation
<code>MontGetSize</code>	Gets the size of the <code>IppsMontState</code> context.
<code>MontInit</code>	Initializes the context and partitions the specified buffer space.
<code>MontSet</code>	Sets the input integer big number to a value and computes the Montgomery reduction index.
<code>MontGet</code>	Extracts the big number modulus.
<code>MontForm</code>	Converts input positive integer big number into Montgomery form.
<code>MontMul</code>	Computes Montgomery modular multiplication for positive integer big numbers of Montgomery form.
<code>MontExp</code>	Computes Montgomery exponentiation.

Montgomery reduction is a technique for efficient implementation of modular multiplication without explicitly carrying out the classical modular reduction step.

This section describes functions for Montgomery modular reduction, Montgomery modular multiplication, and Montgomery modular exponentiation.

Let n be a positive integer, and let R and T be integers such that $R > n$, $\gcd(n, R) = 1$, and $0 < T < nR$. The Montgomery reduction of T modulo n with respect to R is defined as $TR^{-1} \bmod n$.

For better results, functions included in the cryptography package use $R = b^k$ where $b = 2^{32}$ and k is the Montgomery index integer computed by the ceiling function of the bit length of the integer n over 32.

All functions use employ the context `IppsMontState` to serve as an operational vehicle to carry the Montgomery reduction index k , the integer big number modulus n , the least significant word n_0 of the multiplicative inverse of the modulus n with respect to the Montgomery reduction factor R , and a sufficient working buffer reserved for various Montgomery modular operations.

Furthermore, two new terms are introduced in this section:

- length of the context `IppsMontState` is defined as the data length of the modulus n carried by the structure
- size of the context `IppsMontState` is therefore defined as the maximum data length of such an integer modulus n that could be carried by this operational vehicle.

The following example can briefly illustrate the procedure of using the primitives described in this section to compute a classical modular exponentiation $T = x^e \bmod n$. Consider computing $T = x^4 \bmod n$, for some integer x with $0 < x < n$.

First get the buffer size required to configure the context `IppsMontState` by calling `MontGetSize` and then allocate the working buffer using OS service function, with allocated buffer to call `MontInit` to initialize the context `IppsMontState`.

Set the modulus n by calling `MontSet` and then convert x into its respective Montgomery form by calling `MontForm`, that is, computing

$$\underline{x} = xR \bmod n.$$

Then compute the Montgomery reduction of

$$\underline{x}\underline{x}$$

using the function `MontMul` to generate

$$T = \underline{x}\underline{x}R^{-1} \bmod n.$$

The Montgomery reduction of $T * T \bmod n$ with respect to R is

$$T^2 R^{-1} \bmod n = (\underline{x}^2 R^{-1})^2 R^{-1} \bmod n = x^4 R \bmod n.$$

Further applying `MontMul` with this value and the value of 1 yields the desired result $T = x^4 \bmod n$.

The classical modular exponentiation should be computed by performing the following sequence of operations:

1. Get the buffer size required to configure the context `IppsMontState` by calling the function `MontGetSize`. For limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer size significantly while using sliding window method enhances the performance.
2. Allocate working buffer through an operating system memory allocation function and configure the structure `IppsMontState` by calling the function `MontInit` with the allocated buffer and the choice made on the modular exponential method at time invoking `MontGetSize`.
3. Call the function `MontSet` to set the integer big number module for `IppsMontState`.
4. Call the function `MontForm` to convert the integer x to be its Montgomery form.
5. Call the function `MontExp` to compute the Montgomery modular exponentiation.
6. Call the function `MontMul` to compute the Montgomery modular multiplication of the above result with the integer 1 as to convert the above result back to the desired classical modular exponential result.
7. Free the memory using an operating system memory free function, if needed.

MontGetSize

Gets the size of the `IppsMontState` context.

Syntax

```
IppStatus ippsMontGetSize(IppsExpMethod method, int length, int * size);
```

Parameters

<i>method</i>	Selected exponential method.
<i>length</i>	Data field length for the modulus.
<i>size</i>	Size of the buffer required for initialization.

Description

This function is declared in the `ippcp.h` file. The function specifies the buffer size required to define the structuralized working buffer of the context `IppsMontState` to store the modulus and perform operations using various Montgomery modulus schemes.

The function returns the required buffer size based on the selected exponential method. The binary method helps to significantly reduce the buffer size, while the sliding windows method results in enhanced performance.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>length</i> is less than or equal to 0.

MontInit

Initializes the context and partitions the specified buffer space.

Syntax

```
IppStatus ippsMontInit(IppsExpMethod method, int length, IppsMontState *m);
```

Parameters

<i>method</i>	Selected exponential method.
<i>buffer</i>	Buffer for initializing <i>m</i> .
<i>length</i>	Data field length for the modulus.
<i>m</i>	Pointer to the context <code>IppsMontState</code> .

Description

This function is declared in the `ippcp.h` file. The function initializes the context using the specified buffer space. The function then partitions the buffer using the selected modular exponential method in such a way as to carry up to `length * sizeof(Ipp32u)`-bit big number modulus and execute various Montgomery modulus operations.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>length</code> is less than or equal to 0.

MontSet

Sets the input integer big number to a value and computes the Montgomery reduction index.

Syntax

```
IppStatus ippMontSet(const Ipp32u *n, int length, IppsMontState *m);
```

Parameters

<code>n</code>	Input big number modulus.
<code>length</code>	The length of the modulus.
<code>m</code>	Pointer to the context <code>IppsMontState</code> capturing the modulus and the least significant word of the multiplicative inverse Ni .

Description

This function is declared in the `ippcp.h` file. The function sets the input positive integer big number n to be the modulus for the context `IppsMontState *m`, computes the Montgomery reduction index k with respect to the input big number modulus n and the least significant 32-bit word of the multiplicative inverse Ni with respect to the modulus R , that satisfies $R * R^{-1} - n * Ni = 1$.

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 <code>NULL</code> .
<code>ippStsBadModulusErr</code>	Indicates an error condition if the modulus is not a positive odd integer.
<code>ippStsLengthErr</code>	Indicates an error condition if <code>length</code> is less than or equal to 0.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>length</code> is larger than <code>IppsMontState*m</code> .

MontGet

Extracts the big number modulus.

Syntax

```
IppStatus ippMontGet(Ipp32u *n, int *length, const IppsMontState *m);
```

Parameters

<i>m</i>	context <code>IppsMontState</code> .
<i>n</i>	Modulus data field.
<i>length</i>	Modulus data length.

Description

This function is declared in the `ippcp.h` file. The function extracts the big number modulus from the input `IppsMontState *m`.

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 <code>NULL</code> .

MontForm

Converts input positive integer big number into Montgomery form.

Syntax

```
IppStatus ippMontForm(IppsBigNumState *a, IppsMontState *m, IppsBigNumState *r);
```

Parameters

<i>a</i>	Input integer big number within the range $[0, m - 1]$.
<i>m</i>	Input big number modulus of <code>IppsBigNumState</code> .
<i>r</i>	Resulting Montgomery form $r = a * R \bmod m$.

Description

This function is declared in the `ippcp.h` file. The function converts an input positive integer big number into the Montgomery form with respect to the big number modulus and stores the conversion result.

The following pseudocode represents this function:

$r \leftarrow a * R \bmod m$.

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 <code>NULL</code> .
<code>ippStsBadArgErr</code>	Indicates an error condition if <i>a</i> is a negative integer.
<code>ippStsScaleRangeErr</code>	Indicates an error condition if <i>a</i> is more than <i>m</i> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>IppsBigNumState *r</code> is larger than <code>IppsMontState *m</code> .



NOTE. The size of `IppsBigNumState *r` should not be less than the data length of the modulus `m`.

MontMul

Computes Montgomery modular multiplication for positive integer big numbers of Montgomery form.

Syntax

```
IppStatus ippsMontMul(IppsBigNumState *a, IppsBigNumState *b, IppsMontState *m,
IppsBigNumState *r);
```

Parameters

<code>a</code>	Multiplicand within the range $[0, m - 1]$.
<code>b</code>	Multiplier within the range $[0, m - 1]$.
<code>m</code>	Modulus.
<code>r</code>	Montgomery multiplication result.

Description

This function is declared in the `ippcp.h` file. The function computes the Montgomery modular multiplication for positive integer big numbers of Montgomery form with respect to the modulus `IppsMontState *m`. As a result, `IppsBigNumState *r` holds the product.

The following pseudocode represents this function:

$$r \leftarrow a * b * R^{-1} \bmod m.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsBadArgErr</code>	Indicates an error condition if <code>a</code> or <code>b</code> is a negative integer.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is <code>NULL</code> .
<code>ippStsScaleRangeErr</code>	Indicates an error condition if <code>a</code> or <code>b</code> is more than <code>m</code> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>IppsBigNumState *r</code> is larger than <code>IppsMontState *m</code> .



NOTE. The size of `IppsBigNumState *r` should not be less than the data length of the modulus `m`.

Example of Using Montgomery Reduction Scheme Functions

Montgomery Multiplication

```
void MontMul_sample(void){
    int size;

    // define and initialize Montgomery Engine over Modulus N
    Ipp32u bnuN = 19;
    ippsMontGetSize(IppsBinaryMethod, 1, &size);
    IppsMontState* pMont = (IppsMontState*)( new Ipp8u [size] );
    ippsMontInit(IppsBinaryMethod, 1, pMont);
    ippsMontSet(&bnuN, 1, pMont);

    // define and init Big Number multiplicand A
    Ipp32u bnuA = 12;
    IppsBigNumState* bnA = New_BN(1, &bnuA);
    // encode A into Montgomery form
    ippsMontForm(bnA, pMont, bnA);

    // define and init Big Number multiplicand B
    Ipp32u bnuB = 15;
    IppsBigNumState* bnB = New_BN(1, &bnuB);

    // compute R = A*B mod N
    IppsBigNumState* bnR = New_BN(1);
    ippsMontMul(bnA, bnB, pMont, bnR);

    Type_BN("R = A*B mod N:\n", bnR);
    delete [] (Ipp8u*)pMont;
    delete [] (Ipp8u*)bnA;
    delete [] (Ipp8u*)bnB;
    delete [] (Ipp8u*)bnR;
}
```

MontExp

Computes Montgomery exponentiation.

Syntax

```
IppStatus ippMontExp(IppsBigNumState *a, IppsBigNumState *e, IppsMontState *m,
IppsBigNumState *r);
```

Parameters

<i>a</i>	Big number Montgomery integer within the range of $[0, m - 1]$.
<i>e</i>	Big number exponent.
<i>m</i>	Modulus.
<i>r</i>	Montgomery exponentiation result.

Description

This function is declared in the `ippcp.h` file. The function computes Montgomery exponentiation with the exponent specified by the input positive integer big number to the given positive integer big number of the Montgomery form with respect to the modulus *m*.

The following pseudocode represents this function:

$$r \leftarrow a^e R^{-(e-1)} \bmod m.$$

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsBadArgErr</code>	Indicates an error condition if <i>a</i> or <i>e</i> is a negative integer.
<code>ippStsNullPtrErr</code>	Indicates an error condition if any of the specified pointers is <code>NULL</code> .
<code>ippStsScaleRangeErr</code>	Indicates an error condition if <i>a</i> or <i>e</i> is more than <i>m</i> .
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>IppsBigNumState*r</code> is larger than <code>IppsMontState*m</code> .



NOTE. The size of `IppsBigNumState *r` should not be less than the data length of the modulus *m*.

Pseudorandom Number Generation Functions

Many cryptographic systems rely on pseudorandom number generation functions in their design that make the unpredictable nature inherited from a pseudorandom number generator the security foundation to ensure safe communication over open channels and protection against potential adversaries.

The full list of Intel IPP Pseudorandom Number Generation Functions is given in [Table “Intel IPP Pseudorandom Number Generation Functions”](#).

Intel IPP Pseudorandom Number Generation Functions

Function Base Name	Operation
PRNGGetSize	Gets the size of the <code>IppsPRNGState</code> context.
PRNGInit	Initializes user-supplied memory as <code>IppsPRNGState</code> context for future use.
PRNGSetSeed	Sets the initial state with the given input seed for pseudorandom number generation.
PRNGSetAugment	Sets the initial state with the given input entropy for the pseudorandom number generation.
PRNGSetModulus	Sets the initial state with the given input modulus for the pseudorandom number generation.
PRNGSetH0	Sets the initial state with the given input IV for the SHA-1 algorithm.
PRNGen	Generates a pseudorandom unsigned Big Number of the specified bitlength.
PRNGen_BN	Generates a pseudorandom positive Big Number of the specified bitlength.

This section describes functions that comprise the pseudorandom bit sequence generator implemented by a US FIPS-approved method and based on a SHA-1 one-way hash function specified by [\[FIPS PUB 186-2\]](#), [appendix 3](#).

The application code for generating a sequence of pseudorandom bits should perform the following sequence of operations:

1. Call the function [PRNGGetSize](#) to get the size required to configure the `IppsPRNGState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the [PRNGInit](#) function to set up the default value of the parameters for pseudorandom generation process.
3. If the default values of the parameters are not satisfied, call the function [PRNGSetSeed](#) and/or [PRNGSetAugment](#) and/or [PRNGSetModulus](#) and/or [PRNGSetH0](#) to reset any of the control pseudorandom generator parameters.
4. Keep calling the function [PRNGen](#) or [PRNGen_BN](#) to generate pseudo random value of the desired format.
5. Free the memory allocated for the `IppsPRNGState` context by calling the operating system memory free service function.

User’s Implementation of a Pseudorandom Number Generator

Both functions `ippsPRNGen` and `ippsPRNGen_BN`, as well as their supplementary functions represent the implementation of the pseudorandom number generator in the IPPCP library. This given implementation is based on recommendations made in [\[FIPS PUB 186-2\]](#). If you prefer to use the implementation of the pseudorandom number generator which is different from the given, you can still use IPPCP library. To do this, use the following definition of the generator introduced by the IPPCP library:

Syntax

```
typedef IppStatus(_STD_CALL *IppBitSupplier)(Ipp32u* pData, int nBits, void* pEbsParams);
```

Parameters

<code>pData</code>	Pointer to the output data.
<code>nBits</code>	Number of generated data bits.

pEbsParams

Pointer to the user defined context.

Description

This declaration is included in the `ippcp.h` file. The function generates any data (probably pseudorandom numbers) of the specified *nBits* length.

Return Values

<code>ippStsNoErr</code>	Indicates no error. Any other value indicates an error or warning.
<code>ippStsErr</code>	Indicates an error condition.

PRNGGetSize

Gets the size of the `IppsPRNGState` context in bytes.

Syntax

```
IppStatus ippPRNGGetSize(int *pSize);
```

Parameters

*pSize*Pointer to the `IppsPRNGState` context size in bytes.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsPRNGState` context size in bytes and stores it in **pSize*.

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 <code>NULL</code> .

PRNGInit

Initializes user-supplied memory as `IppsPRNGState` context for future use.

Syntax

```
IppStatus ippPRNGInit(int seedBits, IppsPRNGState* pCtx);
```

Parameters

seedBits

Size in bits for the seed value.

*pCtx*Pointer to the `IppsPRNGState` context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as the `IppsPRNGState` context. In addition, the function sets up the default internal random generator parameters (seed, entropy augment, modulus, and initial hash value H0 of the SHA-1 algorithm). PRNG default parameters are as follows:

- seed = 0x0
- entropy augment = 0x0
- modulus = 0xFF
- H0 = 0xC3D2E1F01032547698BADCFEEFCDAB8967452301

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>seedBits</code> is less than 1 or greater than 512.

PRNGSetSeed

Sets up the seed value for the pseudorandom number generator.

Syntax

```
IppStatus ippSPRNGSetSeed(const IppsBigNumState* pSeed, IppsPRNGState* pCtx);
```

Parameters

<code>pSeed</code>	Pointer to the seed value being set up.
<code>pCtx</code>	Pointer to the <code>IppsPRNGState</code> context.

Description

This function is declared in the `ippcp.h` file. The function resets the seed value with the supplied value of `seedBits` bit length. The supplied big number should be created prior to the function call using the appropriate Big Number Arithmetic functions (see [Example 5-1](#)).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.



NOTE. This function restarts the pseudorandom number generation process, which results in losing already generated pseudorandom numbers.

PRNGSetAugment

Sets the initial state with the given input entropy for the pseudorandom number generation.

Syntax

```
IppsStatus ippsPRNGSetAugment(const IppsBigNumState* pAugment, IppsPRNGState* pCtx);
```

Parameters

<i>pAugment</i>	Pointer to the entropy augment value being set up.
<i>pCtx</i>	Pointer to the <code>IppsPRNGState</code> context.

Description

This function is declared in the `ippcp.h` file. The function resets entropy augment value with the supplied value of the *seedBits* bit length. The supplied big number should be created prior to the function call using the appropriate Big Number Arithmetic functions (see [Example](#)).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

PRNGSetModulus

Sets the initial state with the given input modulus for the pseudorandom number generation.

Syntax

```
IppsStatus ippsPRNGSetModulus(const IppsBigNumState* pModulus, IppsPRNGState* pCtx);
```

Parameters

<i>pModulus</i>	Pointer to the modulus value being set up.
<i>pCtx</i>	Pointer to the <code>IppsPRNGState</code> context.

Description

This function is declared in the `ippcp.h` file. The function resets the modulus value with the supplied value up to 160 bit length. The supplied big number should be created prior to the function call using the appropriate Big Number Arithmetic functions (see [Example](#)).

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 <code>NULL</code> .

<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
------------------------------------	---

PRNGSetH0

Sets the initial state with the given input IV for the SHA-1 algorithm.

Syntax

```
IppStatus ippSPRNGSetH0(const IppsBigNumState* pH0, IppsPRNGState* pCtx);
```

Parameters

<code>pH0</code>	Pointer to the initial hash value being set up.
<code>pCtx</code>	Pointer to the <code>IppsPRNGState</code> context.

Description

This function is declared in the `ippcp.h` file. The function resets the initial hash value with the supplied value up to 160 bit length. The supplied big number should be created prior to the function call using the appropriate Big Number Arithmetic functions (see [Example](#)).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

PRNGen

Generates a pseudorandom unsigned Big Number of the specified bitlength.

Syntax

```
IppStatus ippSPRNGen(Ipp32u* pRandBNU, int nBits, void* pCtx);
```

Parameters

<code>pRandBNU</code>	Pointer to the output pseudorandom unsigned integer big number.
<code>nBits</code>	Number of the generated pseudorandom bit.
<code>pCtx</code>	Pointer to the <code>IppsPRNGState</code> context.

Description

This function is declared in the `ippcp.h` file. The function generates pseudorandom unsigned integer big number of the specified `nBits` length.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <code>nBits</code> is less than 1.

PRNGen_BN

Generates a pseudorandom positive Big Number of the specified bitlength.

Syntax

```
IppStatus ippSPRNGen_BN(IppsBigNumState* pRandBN, int nBits, void* pCtx);
```

Parameters

<code>pRandBN</code>	Pointer to the output pseudorandom Big Number.
<code>nBits</code>	Number of the generated pseudorandom bit.
<code>pCtx</code>	Pointer to the <code>IppsPRNGState</code> context.

Description

This function is declared in the `ippcp.h` file. The function generates pseudorandom positive Big Number of the specified `nBits` length.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if <code>nBits</code> is less than 1.

Example of Using Pseudorandom Number Generation Functions

Find Pseudorandom Co-primes

```
void FindCoPrimes(void){
    int size;

    // define Pseudo Random Generator (default settings)
    ippsPRNGGetSize(&size);
    IppsPRNGState* pPrng = (IppsPRNGState*)(new Ipp8u [size] );
    ippsPRNGInit(160, pPrng);

    // define 256-bits Big Numbers X and Y
    const int bnBitSize = 256;
    IppsBigNumState* bnX = New_BN(bnBitSize/32);
    IppsBigNumState* bnY = New_BN(bnBitSize/32);

    // define temporary Big Numbers GCD and 1
    IppsBigNumState* bnGCD = New_BN(bnBitSize/32);
    Ipp32u one = 1;
    IppsBigNumState* bnOne = New_BN(1, &one);

    // generate pseudo random X and Y
    // while GCD(X,Y) != 1
    Ipp32u result;
    int counter;
    for(counter=0,result=1; result; counter++) {
        ippsPRNGen_BN(bnX, bnBitSize, pPrng);
        ippsPRNGen_BN(bnY, bnBitSize, pPrng);
        ippsGcd_BN(bnX, bnY, bnGCD);
        ippsCmp_BN(bnGCD, bnOne, &result);
    }

    cout <<"Coprimes:" <<endl;
```

```

Type_BN("X: ", bnX); cout <<endl;
Type_BN("Y: ", bnY); cout <<endl;
cout <<"were fond on " <<counter <<" attempt" <<endl;

delete [] (Ipp8u*)pPrng;
delete [] (Ipp8u*)bnX;
delete [] (Ipp8u*)bnY;
delete [] (Ipp8u*)bnGCD;
delete [] (Ipp8u*)bnOne;
}

```

Prime Number Generation Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) functions for prime number generation. The full list of prime number generation functions is given in [Table “Intel IPP Prime Number Generation Functions”](#).

Intel IPP Prime Number Generation Functions

Function Base Name	Operation
PrimeGetSize	Gets the size of the <code>IppsPrimeState</code> context.
PrimeInit	Initializes user-supplied memory as the <code>IppsPrimeState</code> context for future use.
PrimeGen	Generates a random probable prime number of the specified bitlength.
PrimeTest	Tests the given integer for being a probable prime.
PrimeSet	Sets the Big Number for primality testing.
PrimeSet_BN	Sets the Big Number for primality testing.
PrimeGet	Extracts the probable prime unsigned integer big number.
PrimeGet_BN	Extracts the probable prime positive Big Number.

This section describes Intel IPP functions for generating probable prime numbers of variable lengths and validating probable prime numbers through a probabilistic primality test scheme for cryptographic use. A probable prime number is thus defined as an integer that passes the Miller-Rabin probabilistic primality-based test.

The scheme adopted for the probable prime number generation is based on a well-known prime number theorem. Study shows that the number of primitives that are no greater than the given large integer x is closely approximated by the expression. Let $\pi(x)$ denote the number of primes that are not greater than x . In this case the statement is true

$$\lim_{x \rightarrow \infty} \frac{\pi(x)}{x / (\ln x)} = 1.$$

Further study indicates that if X represents the event where the tested k -bit integer n is composite and if Y_t denotes the event where the Miller-Rabin test with the security parameter t declares n to be a prime, the test error probability is upper bounded by

$$p_{k,t} \leq k^{2/3} 2^{t-1/2} t 4^{2-\sqrt{tk}} \text{ for } t = 2, k \geq 88, \text{ or } 3 \leq t \leq k/9, k \geq 21.$$

Subsequently, a practical strategy for generating a random k -bit probable prime is to repeatedly pick k -bit random odd integers until finding one integer that can pass a recognized probabilistic primality test scheme as a probable prime. The available set of probable prime number generation functions enables you to specify an appropriate value of the security parameter t used in the Miller-Rabin primality test to meet the cryptographic requirements for your application.

All Intel IPP for prime number generation use the context `IppsPrimeState` as an operational vehicle that carries the bitlength of the target probable prime number, the structure capturing the state of the pseudorandom number generation, the structuralized working buffer used for Montgomery modular computation in the Miller-Rabin primality test, and the buffer to store the generated probable prime number.

The following sequence of operations is required to generate a probable prime number of the specified bitlength:

1. Call the function `PrimeGetSize` to get the size required to configure the `IppsPrimeState` context.
2. Allocate memory through the operating system memory allocation function and configure the `IppsPrimeState` context by calling the function `PrimeInit`.
3. Generate probable prime number of the specified bitlength by calling the function `PrimeGen`. If the returned `IppStatus` is `ippStsInsufficientEntropy`, then change the parameters of the pseudorandom generator and call the function `PrimeGen` again.
4. Extract the generated probable prime number by calling the functions `PrimeGet` and `PrimeGet_BN`.
5. Free the memory allocated to the `IppsPrimeState` context by calling the operating system memory-free service function.

PrimeGetSize

Gets the size of the `IppsPrimeState` context in bytes.

Syntax

```
IppStatus ippPrimeGetSize(int nMaxBits, int* pSize);
```

Parameters

<code>nMaxBits</code>	Maximum length of the probable prime number in bits.
<code>pSize</code>	Pointer to the <code>IppsPrimeState</code> context size in bytes.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsPrimeState` context size in bytes and stores it in `pSize`.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>nMaxBits</code> is less than 1.

PrimeInit

Initializes user-supplied memory as IppsPrimeState context for future use.

Syntax

```
IppStatus ippsPrimeInit(int nMaxBits, IppsPrimeState* pCtx);
```

Parameters

<i>nMaxBits</i>	Maximum length of the probable prime number in bits.
<i>pCtx</i>	Pointer to the IppsPrimeState context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the IppsPrimeState context.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>nMaxBits</i> is less than 1.

PrimeGen

Generates a random probable prime number of the specified bitlength.

Syntax

```
IppStatus ippsPrimeGen(int nBits, int nTrials, IppsPrimeState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>nbits</i>	Target bitlength for the desired probable prime number.
<i>nTrials</i>	Security parameter specified for the Miller-Rabin probable primality.
<i>pCtx</i>	Pointer to the IppsPrimeState context.
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function employs the `rndFuncRandom` Generator specified by the user to generate a random probable prime number of the specified *nBits* length. The generated probable prime number is further validated by the Miller-Rabin primality test scheme with the specified security parameter *nTrials*.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <code>nBits</code> is less than 1.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsBadArgErr</code>	Indicates an error condition if <code>nTrials</code> is less than 1.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <code>nBits</code> > <code>nMaxBits</code> (see PrimeGetSize and PrimeInit)
<code>ippStsInsufficientEntropy</code>	Indicates a warning condition if prime generation fails due to poor choice of entropy.

PrimeTest

Tests the given integer for being a probable prime.

Syntax

```
IppStatus ippPrimeTest(int nTrials, Ipp32u *pResult, IppsPrimeState* pCtx, IppBitSupplier
rndFunc, void* pRndParam);
```

Parameters

<code>nTrials</code>	Security parameter specified for the Miller-Rabin probable primality.
<code>pResult</code>	Pointer to the result of the primality test.
<code>pCtx</code>	Pointer to the <code>IppsPrimeState</code> context.
<code>rndFunc</code>	Specified Random Generator.
<code>pRndParam</code>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function uses the Miller-Rabin probabilistic primality test scheme with the given security parameter to test if the given integer is a probable prime. The pseudorandom number used in the Miller-Rabin test is generated by the specified `rndFunc` Random Generator. The function sets up the `*pResult` as `IS_PRIME` or `IS_COMPOSITE` to show if the input probable prime passes the Miller-Rabin test.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsBadArgErr</code>	Indicates an error condition if <code>nTrials</code> is less than 1.

PrimeSet

Sets the Big Number for primality testing.

Syntax

```
IppStatus ippsPrimeSet(const Ipp32u* pBNU, int nBits, IppsPrimeState* pCtx);
```

Parameters

<i>pBNU</i>	Pointer to the unsigned integer big number.
<i>nBits</i>	Unsigned integer big number length in bits.
<i>pCtx</i>	Pointer to the <code>IppsPrimeState</code> context.

Description

This function is declared in the `ippcp.h` file. The function sets a probable prime number and its length for the probabilistic primality test.

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if <i>nBits</i> is less than 1.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if <i>nBits</i> is too large to fit <i>pCtx</i> .

PrimeSet_BN

Sets the Big Number for primality testing.

Syntax

```
IppStatus ippsPrimeSet_BN(const IppsBigNumState* pBN, IppsPrimeState* pCtx);
```

Parameters

<i>pBN</i>	Pointer to the Big Number context.
<i>pCtx</i>	Pointer to the <code>IppsPrimeState</code> context.

Description

This function is declared in the `ippcp.h` file. The function sets the Big Number for probabilistic primality test.

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 <code>NULL</code> .

<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if the Big Number is too large to fit <i>pCtx</i> .

PrimeGet

Extracts the probable prime unsigned integer big number.

Syntax

```
IppStatus ippPrimeGet(Ipp32u* pBNU, int *pSize, const IppsPrimeState *pCtx);
```

Parameters

<i>pBNU</i>	Pointer to the unsigned integer big number.
<i>pSize</i>	Pointer to the length of the unsigned integer big number.
<i>p</i>	Pointer to the <code>IppsPrimeState</code> context.

Description

This function is declared in the `ippcp.h` file. The function extracts the probable prime number from **pCtx* context and stores it into the specified unsigned integer big number.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

PrimeGet_BN

Extracts the probable prime positive Big Number.

Syntax

```
IppStatus IppsPrimeGet_BN(IppsBigNumState* pBN, const IppsPrimeState *pCtx);
```

Parameters

<i>pBN</i>	Pointer to the Big Number context.
<i>pCtx</i>	Pointer to the <code>IppsPrimeState</code> context.

Description

This function is declared in the `ippcp.h` file. The function extracts the probable prime positive big number from the **pCtx* context and stores it into the specified Big Number context.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if the Big Number is too small to store probable prime number.

Example of Using Prime Number Generation Functions

Check Primality

```
void CheckPrime(void){
    Ipp32u result;
    int size;

    // define 256-bit Prime Generator
    int maxBitSize = 256;
    ippsPrimeGetSize(maxBitSize, &size);
    IppsPrimeState* pPrimeGen = (IppsPrimeState*)( new Ipp8u [size] );
    ippsPrimeInit(maxBitSize, pPrimeGen);

    // define Pseudo Random Generator (default settings)
    ippsPRNGGetSize(&size);
    IppsPRNGState* pPrng = (IppsPRNGState*)(new Ipp8u [size] );
    ippsPRNGInit(160, pPrng);

    // define known prime value  $(2^{128} - 3)/76439$ 
    Ipp32u bnuPrime1[] = {
        0xBEAD208B, 0x5E668076, 0x2ABF62E3, 0xDB7C};
    IppsBigNumState* bnP1 = New_BN(4, bnuPrime1);
    // make sure P1 is really prime
    ippsPrimeSet_BN(bnP1, pPrimeGen);
    ippsPrimeTest(50, &result, pPrimeGen,
        ippsPRNGen, pPrng);
    IS_PRIME==result?
        cout <<"Primality of P1 is confirmed\n" :
        cout <<"Primality of P1 isn't confirmed\n";

    // define another known prime value  $2^{128} - 2^{97} - 1$ 
    Ipp32u bnuPrime2[] = {
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFFD};
```

```

IppsBigNumState* bnP2 = New_BN(4, bnuPrime2);
// make sure P2 is really prime
ippsPrimeSet_BN(bnP2, pPrimeGen);
ippsPrimeTest(50, &result, pPrimeGen,
              ippsPRNGen, pPrng);
IS_PRIME==result?
    cout <<"Primality of P2 is confirmed\n" :
    cout <<"Primality of P2 isn't confirmed\n";

// define composite Big Number C = P1*P2
IppsBigNumState* bnC = New_BN(8);
ippsMul_BN(bnP1, bnP2, bnC);
// make sure C is really composite
ippsPrimeSet_BN(bnC, pPrimeGen);
ippsPrimeTest(50, &result, pPrimeGen,
              ippsPRNGen, pPrng);
IS_PRIME==result?
    cout <<"Strange, but C=P1*P2 is prime\n" :
    cout <<"OK, C=P1*P2 is composite\n";

delete [] (Ipp8u*)pPrimeGen;
delete [] (Ipp8u*)pPrng;
delete [] (Ipp8u*)bnP1;
delete [] (Ipp8u*)bnP2;
delete [] (Ipp8u*)bnC;
}

```

RSA Algorithm Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) functions for RSA algorithm. The section describes a set of primitives to perform operations required for RSA cryptographic systems [PKCS 1.2.1]. This set of primitives offers a flexible user interface that enables the RSA crypto key size scalability with the limit of up to 4096 bits.

RSA algorithm functions include:

- [Functions for Building RSA System](#), the system being then used by functions listed below.
- [RSA Primitives](#), which perform RSA encryption and decryption.

- [RSA Encryption Schemes](#) and [RSA Signature Schemes](#), which combine RSA cryptographic primitives with other techniques, such as computing hash message digests or applying mask generation functions (MGFs), to achieve a particular security goal.

Functions for Building RSA System

The list of functions for building RSA cryptographic system is given in [Table “Intel IPP RSA Algorithm Functions”](#).

Intel IPP RSA Algorithm Functions

Function Base Name	Operation
RSAGetSize	Gets the size of the <code>IppRSASState</code> context.
RSAInit	Initializes user-supplied memory as the <code>IppRSASState</code> context for future use.
RSAPack , RSAUnpack	Packs/unpacks the <code>IppRSASState</code> context into/from a user-defined buffer.
RSASetKey	Sets the tag-designated key component into the established RSA context.
RSAGetKey	Extracts the tag-designated key component from the RSA context.
RSAGenerate	Generates key components for the desired RSA cryptographic system.
RSAValidate	Validates key components of the RSA cryptographic system.

You can use the primitives to build an RSA cryptographic system with the supplied randomized seed and stimulus. The function [RSAGenerate](#) generates the RSA system probable primes p and q , the system composite integer n , as well as the key pair: the RSA public key e and its respective private key d .

[RSA Primitives](#) and RSA-based schemes ([RSA-OAEP Scheme Functions](#) and [RSA-SSA Scheme Functions](#)) use `IppsRSASState` context, which is initialized using the [RSAInit](#) function, as an operational vehicle carrying the RSA system composite integer, a pair of RSA probable primes, RSA key pair, and working buffers.

The `IppsRSASState` context is position-dependent. The [RSAPack](#)/[RSAUnpack](#) functions transform the position-dependent context to a position-independent form and vice versa.

RSAGetSize

Gets the size of the `IppsRSASState` context.

Syntax

```
IppStatus ippsRSAGetSize(int nBitsN, int nBitsP, IppRSAKeyType flag, int* pSize);
```

Parameters

<i>nBitsN</i>	Length of the RSA system in bits (that is, the length of the composite RSA modulus n in bits).
<i>nBitsP</i>	Length in bits of the largest of two prime factors of the RSA modulus.
<i>flag</i>	The flag indicating RSA system for encryption or decryption operation.
<i>pSize</i>	Pointer to the <code>IppsRSASState</code> context size in bytes.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsRSASState` context size in bytes and stores it in `*pSize`. Use the `flag == IppRSAPublic` to establish RSA for encryption or `flag == IppRSAprivate` to establish RSA for the decryption operation. Refer to [RSA-OAEP Scheme Functions](#) and [RSA-SSA Scheme Functions](#) on which `flag` value to select for the schemes.

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 <code>NULL</code> .
<code>ippStsNotSupportedModeErr</code>	Indicates an error condition if <code>nBitsN < 32</code> or <code>nBitsN > 4096</code> .
<code>ippStsBadArgErr</code>	Indicates an error condition if $(nBitsP \geq nBitsN)$ or $(2 * nBitsP < nBitsN)$ or <code>flag</code> value is illegal.

RSASInit

Initializes user-supplied memory as the `IppsRSASState` context for future use.

Syntax

```
IppStatus ippRSASInit(int nBitsN, int nBitsP, IppRSAKeyType flag, IppsRSASState* pCtx);
```

Parameters

<code>nBitsN</code>	Length of the RSA system in bits (that is, the length of the composite RSA modulus <code>n</code> in bits)
<code>nBitsP</code>	Length in bits of the largest of two prime factors of the RSA modulus.
<code>flag</code>	The flag indicating RSA system for encryption or decryption operation.
<code>pCtx</code>	Pointer to the <code>IppsRSASState</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by `pCtx` as the `IppsRSASState` context. Use the `flag == IppRSAPublic` to initialize RSA context for encryption or `flag == IppRSAprivate` to initialize RSA context for the decryption operation. Refer to [RSA-OAEP Scheme Functions](#) and [RSA-SSA Scheme Functions](#) on which `flag` value to select for the schemes.

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 <code>NULL</code> .
<code>ippStsNotSupportedModeErr</code>	Indicates an error condition if <code>nBitsN < 32</code> or <code>nBitsN > 4096</code> .
<code>ippStsBadArgErr</code>	Indicates an error condition if <code>nBitsN > nBitsP</code> or illegal <code>flag</code> value.

RSAPack, RSAUnpack

Packs/unpacks the IppsRSAState context into/from a user-defined buffer.

Syntax

```
IppStatus ippsRSAPack (const IppsRSAState* pCtx, Ipp8u* pBuffer);
IppStatus ippsRSAUnpack (const Ipp8u* pBuffer, IppsRSAState* pCtx);
```

Parameters

<i>pCtx</i>	Pointer to the IppsRSAState context.
<i>pBuffer</i>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `RSAPack` function transforms the *pCtx* context to a position-independent form and stores it in the *pBuffer* buffer. The `RSAUnpack` function performs the inverse operation, that is, transforms the contents of the *pBuffer* buffer into a normal `IppsRSAState` context. The `RSAPack` and `RSAUnpack` functions enable replacing the position-dependent `IppsRSAState` context in the memory.

Call the `RSAGetSize` function prior to `RSAPack`/`RSAUnpack` to determine the size of the buffer.

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 <code>NULL</code> .

RSASetKey

Sets the tag-designated key component into the established RSA context.

Syntax

```
IppStatus ippsRSASetKey(const IppsBigNumState* pBN, IppsRSAKeyTag tag, IppsRSAState* pCtx);
```

Parameters

<i>pBN</i>	Pointer to the Big Number context presented by key component.
<i>tag</i>	The tag of the key component being set up.
<i>pCtx</i>	Pointer to the IppsRSAState context.

Description

This function is declared in the `ippcp.h` file. The function sets the key specified by *pBN* to the tag-designated component of the `IppsRSAState` context:

- *tag* == `IppRSAkeyN`, the function sets up the RSA composite integer *n*

- `tag == IppRSAkeyP`, the function sets up the RSA prime factor p
- `tag == IppRSAkeyQ`, the function sets up the RSA prime factor q
- `tag == IppRSAkeyE`, the function sets up the RSA public exponent e
- `tag == IppRSAkeyD`, the function sets up the RSA public exponent d
- `tag == IppRSAkeyDp`, the function sets up the RSA CRT exponent dP of the p -th factor
- `tag == IppRSAkeyDq`, the function sets up the RSA CRT exponent dQ of the q -th factor
- `tag == IppRSAkeyQinv`, the function sets up the RSA CRT coefficient $qInv$.

The values of p , q , dP , dQ , $qInv$, and e must meet the following equations:

$$e \cdot dP = 1 \bmod (p - 1)$$

$$e \cdot dQ = 1 \bmod (q - 1)$$

$$q \cdot qInv = 1 \bmod p$$

The sequence of function calls

```
ippsRSASetKey(pBN_E, IppRSAkeyE, pRSActx);
```

```
ippsRSASetKey(pBN_N, IppRSAkeyN, pRSActx);
```

defines the public key of the RSA cryptosystem.

The sequence of function calls

```
ippsRSASetKey(pBN_D, IppRSAkeyD, pRSActx);
```

```
ippsRSASetKey(pBN_N, IppRSAkeyN, pRSActx);
```

defines the RSA private key with its type 1 representation.

The sequence of function calls

```
ippsRSASetKey(pBN_P, IppRSAkeyP, pRSActx);
```

```
ippsRSASetKey(pBN_Q, IppRSAkeyQ, pRSActx);
```

```
ippsRSASetKey(pBN_dP, IppRSAkeyDp, pRSActx);
```

```
ippsRSASetKey(pBN_dQ, IppRSAkeyDq, pRSActx);
```

```
ippsRSASetKey(pBN_qInv, IppRSAkeyQinv, pRSActx);
```

defines the RSA private key with its type 2 representation.

Representation types for a private key of an RSA cryptosystem are defined in [\[PKCS 1.2.1\]](#).

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 <code>NULL</code> .
<code>ippStsLengthErr</code>	Indicates an error condition if the length of the Big Number specified by <code>pBN</code> is less than 1.
<code>ippStsBadArgErr</code>	Indicates an error condition if some of the arguments are invalid: Big Number specified by <code>pBN</code> is negative, key component specified by <code>tag</code> does not match the RSA context.
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

`ippStsOutOfRangeErr` Indicates an error condition if the length of the Big Number specified by `pBN` is too big to be stored in the `IppsRSAState` context.

RSAGetKey

Extracts the tag-designated key component from the RSA context.

Syntax

```
IppStatus ippRSAGetKey(IppsBigNumState* pBN, IppsRSAKeyTag tag, const IppsRSAState* pCtx);
```

Parameters

`pBN` Pointer to the output Big Number context.

`tag` The tag of the key component being extracted from.

`pCtx` Pointer to the `IppsRSAState` context.

Description

This function is declared in the `ippcp.h` file. The function extracts the tag-designated key component from the `*pCtx` RSA context and stores it in the specified `*pBN` Big Number context:

- `tag == IppRSAkeyN`, the function extracts the RSA composite integer n
- `tag == IppRSAkeyP`, the function extracts the RSA prime factor p
- `tag == IppRSAkeyQ`, the function extracts the RSA prime factor q
- `tag == IppRSAkeyE`, the function extracts the RSA public exponent e
- `tag == IppRSAkeyD`, the function extracts the RSA public exponent d
- `tag == IppRSAkeyDp`, the function extracts the RSA CRT exponent dP of the p -th factor
- `tag == IppRSAkeyDq`, the function extracts the RSA CRT exponent dQ of the q -th factor
- `tag == IppRSAkeyQinv`, the function extracts the RSA CRT coefficient $qInv$.

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 key component specified by `tag` does not match the RSA context.

`ippStsContextMatchErr` Indicates an error condition if the context parameter does not match the operation.

`ippStsOutOfRangeErr` Indicates an error condition if the length of the Big Number specified by `pBN` is too small to be stored as a key component.

RSAGenerate

Generates key components for the desired RSA cryptographic system.

Syntax

```
IppStatus ippRSAGenerate(IppsBigNumState* pE, int nBitsN, int nBitsP, int nTrials,
IppsRSAStruct* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>pE</i>	Pointer to the <code>IppsBigNumState</code> context of the newly generated RSA public exponent key.
<i>nBitsN</i>	Length of the RSA system in bits (that is, the length of the composite RSA modulus <i>n</i> in bits).
<i>nBitsP</i>	Length in bits of one of the two prime factors of the RSA modulus.
<i>nTrials</i>	Security parameter specified for the Miller-Rabin probable primality.
<i>pCtx</i>	Pointer to the <code>IppsRSAStruct</code> context.
<i>rndFunc</i>	Pseudorandom number generator.
<i>pRndParam</i>	Pointer to the context of the pseudorandom number generator.

Description

The function is declared in the `ippcp.h` file. This function generates the desired RSA cryptographic system based on the following data:

- Input **pE* context, specifying the initial value for searching the RSA public exponent.
- Input parameters *nBitsN* and *nBitsP*, specifying the bit lengths of the composite RSA modulus *n* and the largest RSA prime factor *p*, respectively.

This function sequentially performs the following computations:

1. Generates random probable prime numbers *p* and *q* using the specified pseudorandom number generator *rndFunc*.
2. Computes the RSA composite modulus $n = (p * q)$ and the RSA private exponent *d*.
3. Computes all other CRT-related RSA components.

To generate RSA keys using the `RSAGenerate` function, call it in the following sequence of steps:

1. Establish the pseudorandom number generator.
2. Define the size of the RSA cryptosystem, that is, lengths *nBitsN* and *nBitsP* of *n* and *p* (in bits), respectively.
3. Define the RSA cryptosystem using the `IppRSAprivate` flag. To do this, use the following function calls:

```
ippRSAGetSize(nBitsN, nBitsP, IppRSAprivate, &rsaCtxSize);
```

```
ippRSAInit(nBitsN, nBitsP, IppRSAprivate, pCtx);
```

where *pCtx* is the memory allocated for the RSA context.

4. Define the initial value of the public key E and make the call

```
ippsRSAGenerate(pE, nBitsN, nBitsP, nTrials, pCtx, rndFunc, pRndParam);
```

If `ippsRSAGenerate` returns `IppNoErr`, keys are generated.

If `ippsRSAGenerate` returns `ippStsInsufficientEntropy`, repeat this step from the beginning.



NOTE. `ippsRSAGenerate` may change the initial value of the public key E .

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsNotSupportedModeErr</code>	Indicates an error condition if RSA context was initialized with the <code>IppRSAPublic</code> flag.
<code>ippStsBadArgErr</code>	Indicates an error condition in cases not explicitly mentioned above.
<code>ippStsInsufficientEntropy</code>	Indicates a warning condition if prime generation fails due to poor choice of entropy.

See Also

- [RSAGetSize](#)
- [RSAInit](#)
- [RSAValidate](#)
- [Pseudorandom Number Generation Functions](#)

RSAValidate

Validates key components of the RSA cryptographic system.

Syntax

```
IppStatus ippsRSAValidate(const IppsBigNumState* pE, int nTrials, Ipp32u* pResult,
IppsRSAState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<code>pE</code>	Pointer to the <code>IppsBigNumState</code> context of the RSA public exponent (e).
<code>nTrials</code>	Parameter that determines the probability of primality for each P and Q factors of RSA modulo N .
<code>pResult</code>	Pointer to the result of validation.

<i>pCtx</i>	Pointer to the <code>IppsRSAState</code> context.
<i>rndFunc</i>	Pseudorandom number generator.
<i>pRndParam</i>	Pointer to the context of the pseudorandom number generator.

Description

This function is declared in the `ippcp.h` file. The function validates key components of the RSA cryptographic system and stores the result (`IS_VALID_KEY`, `IS_INVALID_KEY`, or `IS_INCOMPLETED_KEY`) of the validation procedure in **pResult*.

The meanings of the values of **pResult* are as follows:

<code>IS_VALID_KEY</code>	<p>The RSA key is valid.</p> <p>The private key is valid if it meets all the following conditions:</p> <ul style="list-style-type: none"> • p and q are prime. • $e \cdot dP = 1 \pmod{p-1}$ and $e \cdot dQ = 1 \pmod{q-1}$ • $q \cdot qInv = 1 \pmod{p}$
<code>IS_INVALID_KEY</code>	<p>The RSA key is not valid.</p> <p>The key is invalid if any of the above conditions is not met.</p>
<code>IS_INCOMPLETED_KEY</code>	<p>Some of the necessary key components of the quintuple (p, q, dP, dQ, or $qInv$) are not available.</p>

Prepare the contexts for a call to `RSASValidate` as follows:

- Initialize the *pCtx** context for a private-key RSA system.
To do this, supply `flag==IppRSAprivate` to the `RSASInit` function.
- Set up the private key in the form of a quintuple (p , q , dP , dQ , $qInv$).
There are two ways to do this:
 - Set each component explicitly through a sequence of calls to `RSASetKey` with `IppRSAkeyP`, `IppRSAkeyQ`, `IppRSAkeyDp`, `IppRSAkeyDq`, and `IppRSAkeyQinv` key tags.
 - Use a sequence of calls to `RSASetKey` with `IppRSAkeyP`, `IppRSAkeyQ`, and `IppRSAkeyD` key tags. This suffices because in this case all the other key components are computed implicitly.



NOTE. Because the `RSAGenerate` and `RSASValidate` functions use similar algorithms, `RSASValidate` is unnecessary if the RSA system is generated by `RSAGenerate`. Call `RSASValidate` when RSA keys are got from outside and you are not sure whether they are valid.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

<code>ippStsBadArgErr</code>	Indicates an error condition if $nBitsN > nBitsP$ or $nTrials < 1$.
<code>ippStsLengthErr</code>	Indicates an error condition if the length of the Big Number specified by pE is less than 1.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if the length of the Big Number specified by pE is too big to be stored in the <code>IppsRSASState</code> context.

See Also

- [Pseudorandom Number Generation Functions](#)

RSA Primitives

The list of RSA cryptographic primitives is given in [Table "Intel IPP RSA Primitives"](#).

Intel IPP RSA Primitives

Function Base Name	Operation
RSAEncrypt	Performs the RSA encryption operation.
RSADecrypt	Performs the RSA decryption operation.

The application code for conducting a typical RSA encryption must perform the following sequence of operations, starting with building of a crypto system:

1. Call the function [RSAGetSize](#) to get the size required to configure `IppsRSASState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the [RSAInit](#) function to initialize the context for the RSA encryption.
3. Keep calling the [RSASetKey](#) to set up RSA public key (n, e) .
4. Invoke the [RSAEncrypt](#) function with the established RSA public key to encode the plaintext into the respective ciphertext.
5. Free the memory allocated for the `IppsRSASState` context by calling the operating system memory free service function.

The typical application code for the RSA decryption must perform the following sequence of operations:

1. Call the function [RSAGetSize](#) to get the size required to configure `IppsRSASState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the [RSAInit](#) function to initialize the context for the RSA decryption.
3. Establish the RSA private key by means of either [RSAGenerate](#) function or by key setup function [RSASetKey](#). The [RSAGenerate](#) function computes the private key d with respect to the generated public key e , as well as several other components for applying the CRT. When using [RSASetKey](#), you have an option of presenting the private key either as a pair (n, d) or quantiple $(p, q, dP, dQ, qInv)$.
4. Invoke the [RSADecrypt](#) function with the established RSA public key to decode the ciphertext into the respective plaintext.
5. Free the memory allocated for the `IppsRSASState` context by calling the operating system memory free service function.

RSAEncrypt

Performs the RSA encryption operation.

Syntax

```
IppStatus ippsRSAEncrypt(const IppsBigNumState* pX, IppsBigNumState* pY, IppsRSAStruct* pCtx);
```

Parameters

<i>pX</i>	Ppointer to the <code>IppsBigNumState</code> context of the plaintext.
<i>pY</i>	Pointer to the <code>IppsBigNumState</code> context of the ciphertext.
<i>pCtx</i>	Pointer to the <code>IppsRSAStruct</code> context.

Description

This function is declared in the `ippcp.h` file. The function performs the RSA encryption operation.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if: length of the Big Number specified by <i>pX</i> is too big length of the Big Number specified by <i>pY</i> is too small.

RSADecrypt

Performs the RSA decryption operation.

Syntax

```
IppStatus ippsRSADecrypt(const IppsBigNumState* pX, IppsBigNumState* pY, IppsRSAStruct* pCtx);
```

Parameters

<i>pX</i>	Ppointer to the <code>IppsBigNumState</code> context of the ciphertext.
<i>pY</i>	Pointer to the <code>IppsBigNumState</code> context of the plaintext.
<i>pCtx</i>	Pointer to the <code>IppsRSAStruct</code> context.

Description

This function is declared in the `ippcp.h` file. The function performs the RSA encryption operation.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if: length of the Big Number specified by pX is too big or length of the Big Number specified by pY is too small.

Example of Using RSA Primitive Functions

The following example illustrates the use of RSA primitives. The example uses the `BigNumber` class and functions creating some cryptographic contexts, whose source code can be found in [Appendix Support Functions and Classes](#).

Use of RSA Primitives

```

static Ipp32u dataN[] = {
    0x091DBDCB, 0x46F8E5FD,
    0xCA2A8F59, 0xE2537298, 0xF6C1687F, 0x527A9A41, 0x7B61A51F, 0xE0AAB12D,
    0x4598394E, 0x8834B245, 0x06095374, 0xEE6A649D, 0xD93A2584, 0x3EE6B4B7,
    0xDFC73772, 0xAFB8E0A3, 0x5B8B807F, 0x19719D8A, 0x60E1EC46, 0x76ED520D,
    0xEB6FCD48, 0x61EA48CE, 0x035C02AB, 0xB8DFBAAF, 0x7454F51F, 0x40D6B6F0,
    0xD41043A4, 0x368D07EE, 0x9DA871F7, 0x2338AC2B, 0x0682CE9C, 0xBBF82F09
};

static Ipp32u dataP[] = {
    0x58FB6599, 0x7541BA2A, 0x459D1F39, 0x5B252176,
    0xAA040A2D, 0x7E28FAE7, 0x6E5D1E3B, 0x124EF023, 0x3D84F632, 0x93B81A9E,
    0xAEF4FDA4, 0x99EB9F44, 0xA1B56001, 0x08810B10, 0xB1B9B3C9, 0xEECF8E81
};

static Ipp32u dataQ[] = {
    0xAF461503, 0xA441E700, 0x4D0416A5, 0xCE335252,
    0x3204B5CF, 0xEA0DA3B4, 0x66B42E92, 0x9840B416, 0x028B9D86, 0x5A0F2035,
    0x8866B1D0, 0x3F6C42D0, 0xAAD1D935, 0x341233EA, 0x27F453F6, 0xC97FB1F0
};

static Ipp32u dataE[] = {0x11};

int RSA_sample(void)
{
    BigNumber P(dataP, sizeof(dataP)/sizeof(dataP[0]));
    BigNumber Q(dataQ, sizeof(dataQ)/sizeof(dataQ[0]));
    BigNumber N = P*Q;
    BigNumber E(dataE, sizeof(dataE)/sizeof(dataE[0]));
    IppsRSAState* pRSAPub = newRSA(N.BitSize(), P.BitSize(), IppRSAPublic);
    IppsRSAState* pRSAprv1 = newRSA(N.BitSize(), P.BitSize(), IppRSAprivate);
    IppsRSAState* pRSAprv2 = newRSA(N.BitSize(), P.BitSize(), IppRSAprivate);

    // compute private key
    BigNumber phi = (P-BigNumber(1))*(Q-BigNumber(1));
    BigNumber D = phi.InverseMul(E);

```

```

// set up public RSA (N,E)
ippsRSASetKey(N, IppRSAkeyN, pRSAPub);
ippsRSASetKey(E, IppRSAkeyE, pRSAPub);

// set up private (no CRT) RSA (N, D)
ippsRSASetKey(N, IppRSAkeyN, pRSAprv1);
ippsRSASetKey(D, IppRSAkeyD, pRSAprv1);

// set up private (CRT) RSA (P,Q,D)
ippsRSASetKey(P, IppRSAkeyP, pRSAprv2);
ippsRSASetKey(Q, IppRSAkeyQ, pRSAprv2);
ippsRSASetKey(D, IppRSAkeyD, pRSAprv2);

// validate RSA
IppsPRNGState* pRand = newPRNG();
Ipp32u result;
ippsRSAValidate(E, 50, &result, pRSAprv2, ippsPRNGen, pRand);
if(IS_VALID_KEY!=result)
{
    cout <<"validation fail" <<endl;
    return 0;
}
// validation pass

// plaintext
Ipp32u dataM[] = {
    0x4D353E2D,0xD2F1B76D,
    0x5281CE32,0x7BC27519,0x2F3AC14F,0x0448DB97,0xD095AEB4,0x82FB3E87,
    0x1BE392F9,0x43581159,0xD5024121,0xB48D2869,0x2BAAD29A,0xA1B7C136,
    0xF47728B4,0x4CDCFE4F,0x839A2DDB,0xFF8AE10E,0x25C9C2B3,0xF93EDCFB,
    0x4626F5AF,0xD7E0B2C0,0xB4251F84,0xC31B2E8B,0xA8F55267,0x5C68F1EE,
    0x26DCD87D,0xCA82310B,0x504B45E2,0x6350E329,0xACE9E300,0x00EB7A19
};

BigNumber M(dataM,sizeof(dataM)/sizeof(dataM[0]));

```

```
// encrypt plaintext
BigNumber C(0,N.DwordSize());
ippsRSAEncrypt(M, C, pRSAPub);

// decrypt ciphertext using pRSAprv1
BigNumber Z1(0,N.DwordSize());
ippsRSADecrypt(C, Z1, pRSAprv1);

// decrypt ciphertext using pRSAprv2
BigNumber Z2(0,N.DwordSize());
ippsRSADecrypt(C, Z2, pRSAprv2);

deleteRSA(pRSAPub);
deleteRSA(pRSAprv1);
deleteRSA(pRSAprv2);

return (M==Z1) && (M==Z2);
}
```

RSA Encryption Schemes

RSA-OAEP Scheme Functions

This subsection describes functions implementing RSA-OAEP encryption scheme, featured in [PKCS 1.2.1]. The full list of these functions is given in [Table “Intel IPP RSA-based Encryption Scheme Functions”](#).

Intel IPP RSA-based Encryption Scheme Functions

Function Base Name	Operation
RSAOAEPEncrypt	Carries out the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_MD5	MD5-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA1	SHA-1-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA224	SHA-224-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA256	SHA-256-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA384	SHA-384-based helper of the RSA-OAEP encryption scheme.
RSAOAEPEncrypt_SHA512	SHA-512-based helper of the RSA-OAEP encryption scheme.

Function Base Name	Operation
RSAOAEPDecrypt	Carries out the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_MD5	MD5-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA1	SHA-1-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA224	SHA-224-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA256	SHA-256-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA384	SHA-384-based helper of the RSA-OAEP decryption scheme.
RSAOAEPDecrypt_SHA512	SHA-512-based helper of the RSA-OAEP decryption scheme.

To invoke a function that carries out RSA-OAEP scheme, the `IppsRSASState` context must be initialized (by the `RSASInit` function) with a proper value of the `flag` parameter:

- For the RSA-OAEP encryption scheme, `flag == IppRSAPublic`
- For the RSA-OAEP decryption scheme, `flag == IppRSAprivate`.

RSAOAEPEncrypt

Carries out the RSA-OAEP encryption scheme.

Syntax

```
IppStatus ippsRSAOAEPEncrypt(const Ipp8u* pSrc, int srcLen, const Ipp8u* pLabel, int
labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSASState* pCtx, IppHash hashFunc, int hashLen,
IppMGF mgfFunc);
```

Parameters

<code>pSrc</code>	Pointer to the octet message to be encrypted.
<code>srcLen</code>	Length of the message to be encrypted.
<code>pLabel</code>	Pointer to the optional label to be associated with the message.
<code>labLen</code>	Length of the optional label.
<code>pSeed</code>	Pointer to the random octet string of length <code>hashLen</code> .
<code>pDst</code>	Pointer to the output octet ciphertext string.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.
<code>hashFunc</code>	Hash function, which meets the General Definition of a Hash Function provided in chapter 3 .
<code>hashLen</code>	Length of the hash function output, in octets.
<code>mgfFunc</code>	Mask generation function (MGF), which meets the definition provided in section User's Implementation of a Mask Generation Function in chapter 3 .

Description

This function is declared in the `ippcp.h` file. The function carries out the RSA-OAEP encryption scheme, defined in [PKCS 1.2.1].

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input length parameters do not meet any of the following conditions:

$srcLen > 0$

$srcLen > N - 2 * hashLen - 2$, where N is the length of the RSA modulus in octets,

$labLen \geq 0$,

$hashLen > 0$.

RSAOAEP Encrypt_MD5

Carries out the RSA-OAEP encryption scheme using MD5 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEP Encrypt_MD5(const Ipp8u* pSrc, int srcLen, const Ipp8u* pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

<code>pSrc</code>	Pointer to the octet message to be encrypted.
<code>srcLen</code>	Length of the message to be encrypted.
<code>pLabel</code>	Pointer to the optional label to be associated with the message.
<code>labLen</code>	Length of the optional label.
<code>pSeed</code>	Pointer to the random octet string of length <code>hashLen</code> , where <code>hashLen</code> is the length (in octets) of the hash message digest.
<code>pDst</code>	Pointer to the output octet ciphertext string.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses MD5 hash function and MD5-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general [RSAOAEP Encrypt](#) function.

Return Values

The function may return any of the values that the general `RSAOAEP Encrypt` function returns.

RSAOAEP_Encrypt_SHA1

Carries out the RSA-OAEP encryption scheme using SHA-1 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEP_Encrypt_SHA1(const Ipp8u* pSrc, int srcLen, const Ipp8u* pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet message to be encrypted.
<i>srcLen</i>	Length of the message to be encrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pSeed</i>	Pointer to the random octet string of length <i>hashLen</i> , where <i>hashLen</i> is the length (in octets) of the hash message digest.
<i>pDst</i>	Pointer to the output octet ciphertext string.
<i>pCtx</i>	Pointer to the properly initialized <i>IppsRSAState</i> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-1 hash function and SHA-1-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general [RSAOAEP_Encrypt](#) function.

Return Values

The function may return any of the values that the general [RSAOAEP_Encrypt](#) function returns.

RSAOAEP_Encrypt_SHA224

Carries out the RSA-OAEP encryption scheme using SHA-224 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEP_Encrypt_SHA224(const Ipp8u* pSrc, int srcLen, const Ipp8u* pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet message to be encrypted.
<i>srcLen</i>	Length of the message to be encrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pSeed</i>	Pointer to the random octet string of length <i>hashLen</i> , where <i>hashLen</i> is the length (in octets) of the hash message digest.

<i>pDst</i>	Pointer to the output octet ciphertext string.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-224 hash function and SHA-224-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general [RSAOAEP_Encrypt](#) function.

Return Values

The function may return any of the values that the general `RSAOAEP_Encrypt` function returns.

RSAOAEP_Encrypt_SHA256

Carries out the RSA-OAEP encryption scheme using SHA-256 hash algorithm.

Syntax

```
IppStatus ippSRSAOAEP_Encrypt_SHA256(const Ipp8u* pSrc, int srcLen, const Ipp8u* pLabel,
int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSASState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet message to be encrypted.
<i>srcLen</i>	Length of the message to be encrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pSeed</i>	Pointer to the random octet string of length <i>hashLen</i> , where <i>hashLen</i> is the length (in octets) of the hash message digest.
<i>pDst</i>	Pointer to the output octet ciphertext string.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-256 hash function and SHA-256-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general [RSAOAEP_Encrypt](#) function.

Return Values

The function may return any of the values that the general `RSAOAEP_Encrypt` function returns.

RSAOAEP_Encrypt_SHA384

Carries out the RSA-OAEP encryption scheme using SHA-384 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEP_Encrypt_SHA384(const Ipp8u* pSrc, int srcLen, const Ipp8u* pLabel,
int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet message to be encrypted.
<i>srcLen</i>	Length of the message to be encrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pSeed</i>	Pointer to the random octet string of length <i>hashLen</i> , where <i>hashLen</i> is the length (in octets) of the hash message digest.
<i>pDst</i>	Pointer to the output octet ciphertext string.
<i>pCtx</i>	Pointer to the properly initialized <i>IppsRSAState</i> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-384 hash function and SHA-384-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general [RSAOAEP_Encrypt](#) function.

Return Values

The function may return any of the values that the general [RSAOAEP_Encrypt](#) function returns.

RSAOAEP_Encrypt_SHA512

Carries out the RSA-OAEP encryption scheme using SHA-512 hash algorithm.

Syntax

```
IppStatus ippsRSAOAEP_Encrypt_SHA512(const Ipp8u* pSrc, int srcLen, const Ipp8u* pLabel,
int labLen, const Ipp8u* pSeed, Ipp8u* pDst, IppsRSAState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet message to be encrypted.
<i>srcLen</i>	Length of the message to be encrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pSeed</i>	Pointer to the random octet string of length <i>hashLen</i> , where <i>hashLen</i> is the length (in octets) of the hash message digest.

<code>pDst</code>	Pointer to the output octet ciphertext string.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-512 hash function and SHA-512-based MGF implemented in Intel IPP to carry out the RSA-OAEP encryption scheme and thus is a specific form of the general [RSAOAEPDecrypt](#) function.

Return Values

The function may return any of the values that the general `RSOAEPDecrypt` function returns.

RSOAEPDecrypt

Carries out the RSA-OAEP decryption scheme.

Syntax

```
IppStatus ippRSOAEPDecrypt(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen, Ipp8u* pDst, int* pDstLen, IppsRSASState* pCtx, IppHash hushFunc, int hashLen, IppMGF mgfFunc);
```

Parameters

<code>pSrc</code>	Pointer to the octet ciphertext to be decrypted.
<code>pLabel</code>	Pointer to the optional label to be associated with the message.
<code>labLen</code>	Length of the optional label.
<code>pDst</code>	Pointer to the output octet plaintext message.
<code>pDstLen</code>	Pointer to the length of the decrypted message.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.
<code>hashFunc</code>	Hash function, which meets the General Definition of a Hash Function provided in chapter 3 .
<code>hashLen</code>	Length of the hash function output, in octets.
<code>mgfFunc</code>	Mask generation function (MGF), which meets the definition provided in section User's Implementation of a Mask Generation Function in chapter 3 .

Description

This function is declared in the `ippcp.h` file. The function carries out the RSA-OAEP decryption scheme defined in [PKCS 1.2.1].

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.

<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input length parameters do not meet any of the following conditions:

$labLen \geq 0$,

$hashLen > 0$,

$2 * hashLen + 2 \geq N$, where N is the length of the RSA modulus in octets.

See Also

RSAOAEPDecrypt_MD5

Carries out the RSA-OAEP decryption scheme using MD5 hash algorithm.

Syntax

```
IppStatus ippRSOAEPDecrypt_MD5(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen, Ipp8u*
pDst, int* pDstLen, IppsRSASState* pCtx);
```

Parameters

<code>pSrc</code>	Pointer to the octet ciphertext to be decrypted.
<code>pLabel</code>	Pointer to the optional label to be associated with the message.
<code>labLen</code>	Length of the optional label.
<code>pDst</code>	Pointer to the output octet plaintext message.
<code>pDstLen</code>	Pointer to the length of the decrypted message.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses MD5 hash function and MD5-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general [RSAOAEPDecrypt](#) function.

Return Values

The function may return any of the values that the general `RSAOAEPDecrypt` function returns.

RSAOAEPDecrypt_SHA1

Carries out the RSA-OAEP decryption scheme using SHA-1 hash algorithm.

Syntax

```
IppStatus ippRSOAEPDecrypt_SHA1(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen,
Ipp8u* pDst, int* pDstLen, IppsRSASState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet ciphertext to be decrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pDst</i>	Pointer to the output octet plaintext message.
<i>pDstLen</i>	Pointer to the length of the decrypted message.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-1 hash function and SHA-1based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general [RSAOAEPDecrypt](#) function.

Return Values

The function may return any of the values that the general `RSAOAEPDecrypt` function returns.

RSAOAEPDecrypt_SHA224

Carries out the RSA-OAEP decryption scheme using SHA-224 hash algorithm.

Syntax

```
IppStatus ippRSOAEPDecrypt_SHA224(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen,
Ipp8u* pDst, int* pDstLen, IppsRSASState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet ciphertext to be decrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pDst</i>	Pointer to the output octet plaintext message.
<i>pDstLen</i>	Pointer to the length of the decrypted message.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-224 hash function and SHA-224-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general [RSAOAEPDecrypt](#) function.

Return Values

The function may return any of the values that the general [RSAOAEPDecrypt](#) function returns.

RSAOAEPDecrypt_SHA256

Carries out the RSA-OAEP decryption scheme using SHA-256 hash algorithm.

Syntax

```
IppStatus ippSRSAOAEPDecrypt_SHA256(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen,
Ipp8u* pDst, int* pDstLen, IppsRSASState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet ciphertext to be decrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pDst</i>	Pointer to the output octet plaintext message.
<i>pDstLen</i>	Pointer to the length of the decrypted message.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-256 hash function and SHA-256-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general [RSAOAEPDecrypt](#) function.

Return Values

The function may return any of the values that the general [RSAOAEPDecrypt](#) function returns.

RSAOAEPDecrypt_SHA384

Carries out the RSA-OAEP decryption scheme using SHA-384 hash algorithm.

Syntax

```
IppStatus ippSRSAOAEPDecrypt_SHA384(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen,
Ipp8u* pDst, int* pDstLen, IppsRSASState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet ciphertext to be decrypted.
-------------	--

<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pDst</i>	Pointer to the output octet plaintext message.
<i>pDstLen</i>	Pointer to the length of the decrypted message.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-384 hash function and SHA-384-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general [RSAOAEPDecrypt](#) function.

Return Values

The function may return any of the values that the general [RSAOAEPDecrypt](#) function returns.

RSAOAEPDecrypt_SHA512

Carries out the RSA-OAEP decryption scheme using SHA-512 hash algorithm.

Syntax

```
IppStatus ippRSAOAEPDecrypt_SHA512(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen,
Ipp8u* pDst, int* pDstLen, IppsRSAState* pCtx);
```

Parameters

<i>pSrc</i>	Pointer to the octet ciphertext to be decrypted.
<i>pLabel</i>	Pointer to the optional label to be associated with the message.
<i>labLen</i>	Length of the optional label.
<i>pDst</i>	Pointer to the output octet plaintext message.
<i>pDstLen</i>	Pointer to the length of the decrypted message.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-512 hash function and SHA-512-based MGF implemented in Intel IPP to carry out the RSA-OAEP decryption scheme and thus is a specific form of the general [RSAOAEPDecrypt](#) function.

Return Values

The function may return any of the values that the general [RSAOAEPDecrypt](#) function returns.

PKCS V1.5 Encryption Scheme Functions

This subsection describes functions implementing the RSA encryption scheme defined in version 1.5 of the PKCS#1 standard ([[PKCS 1.2.1](#)]).

The full list of these functions is given in [Table “Intel IPP PKCS V1.5 Encryption Scheme Functions”](#).

Intel IPP PKCS V1.5 Encryption Scheme Functions

Function Base Name	Operation
RSAEncrypt_PKCSv15	Carries out the encryption using the v1.5 version of the PKCS#1 standard.
RSADecrypt_PKCSv15	Carries out the decryption using the v1.5 version of the PKCS#1 standard.

To invoke a function that carries out a PKCS V1.5 encryption scheme, the `IppsRSAState` context must be initialized (by the [RSAInit](#) function) with a proper value of the *flag* parameter:

- For PKCS V1.5 encryption, *flag* == `IppRSAPublic`
- For PKCS V1.5 decryption, *flag* == `IppRSAprivate`.

RSAEncrypt_PKCSv15

Carries out the encryption using the v1.5 version of the PKCS#1 standard.

Syntax

```
IppStatus ippsRSAEncrypt_PKCSv15(const Ipp8u* pSrc, Ipp32u srcLen, const Ipp8u* pRandPS,
Ipp8u* pDst, IppsRSAState* pRSA);
```

Parameters

<i>pSrc</i>	Pointer to the message to be encrypted.
<i>srcLen</i>	Length (in bytes) of the message. The message can be empty, that is, <i>srcLen</i> ==0.
<i>pRandPS</i>	Pointer to the non-zero octet padding string of an appropriate length. <i>pRandPS</i> can be <code>NULL</code> . In this case, the function applies the padding string of 0xFF bytes.
<i>pDst</i>	Pointer to the output ciphertext string.
<i>pRSA</i>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function carries out the RSA encryption using the public key according to the v1.5 version of the PKCS#1 standard, defined in [PKCS 1.2.1].

If `RSAEncrypt_PKCSv15` receives a non-zero *pRandPS* pointer, the function assumes that the length of the padding string is at least $k - \text{srcLen} - 3$ bytes, where k is the length of the RSA modulus in bytes.



Important. The v1.5 version of the PKCS#1 standard requires that you provide a padding string that does not contain zero bytes. If the padding string contains a zero byte, the encryption operation completes successfully, but the inverse decryption fails.

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 other than <i>pRandPS</i> is <code>NULL</code> .

<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length n of the RSA modulus is too small in comparison with the length of plaintext, that is, $srcLen > n - 11$.

See Also

- [RSADecrypt_PKCSv15](#)

RSADecrypt_PKCSv15

Carries out the decryption using the v1.5 version of the PKCS#1 standard.

Syntax

```
IppStatus ippRSADecrypt_PKCSv15(const Ipp8u* pSrc, Ipp8u* pDst, Ipp32u* pDstLen,
IppsRSAState* pRSA);
```

Parameters

<code>pSrc</code>	Pointer to the cyphertext to be decrypted.
<code>pDst</code>	Pointer to the output text message.
<code>pDstLen</code>	Pointer to the length (in bytes) of the decrypted message.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function carries out the RSA decryption using the private key according to the v1.5 version of the PKCS#1 standard, defined in [\[PKCS 1.2.1\]](#).



NOTE. If an empty message is encrypted by the [RSAEncrypt_PKCSv15](#) function, `RSADe-crypt_PKCSv15` returns an empty string, that is, `*pDstLen==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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length n of the RSA modulus is too small, that is, $n < 11$.

`ippStsPaddingErr`

Indicates an error condition in decryption that occurs in cases as follows:

- The value to be decrypted exceeds the RSA modulus.
- Internally applied EME-PKCS-v1.5 procedure returns a decryption error (see [PKCS 1.2.1] for details).

RSA Signature Schemes

RSASSA-PSS Scheme Functions

This subsection describes functions implementing RSA-SSA signature scheme with appendix, featured in [PKCS 1.2.1].

The full list of these functions is given in the table below.

Intel IPP RSASSA-PSS Signature Scheme Functions

Function Base Name	Operation
<code>RSASSASign</code>	Carries out the RSASSA-PSS signature generation scheme.
<code>RSASSASign_MD5</code>	MD5-based helper of the RSASSA-PSS signature generation scheme.
<code>RSASSASign_SHA1</code>	SHA-1-based helper of the RSASSA-PSS signature generation scheme.
<code>RSASSASign_SHA224</code>	SHA-224-based helper of the RSASSA-PSS signature generation scheme.
<code>RSASSASign_SHA256</code>	SHA-256-based helper of the RSASSA-PSS signature generation scheme.
<code>RSASSASign_SHA384</code>	SHA-384-based helper of the RSASSA-PSS signature generation scheme.
<code>RSASSASign_SHA512</code>	SHA-512-based helper of the RSASSA-PSS signature generation scheme.
<code>RSASSAVerify</code>	Carries out the RSASSA-PSS signature verification scheme.
<code>RSASSAVerify_MD5</code>	MD5 based helper of the RSASSA-PSS signature verification scheme.
<code>RSASSAVerify_SHA1</code>	SHA-1-based helper of the RSASSA-PSS signature verification scheme.
<code>RSASSAVerify_SHA224</code>	SHA-224-based helper of the RSASSA-PSS signature verification scheme.
<code>RSASSAVerify_SHA256</code>	SHA-256-based helper of the RSASSA-PSS signature verification scheme.
<code>RSASSAVerify_SHA384</code>	SHA-384-based helper of the RSASSA-PSS signature verification scheme.
<code>RSASSAVerify_SHA512</code>	SHA-512-based helper of the RSASSA-PSS signature verification scheme.

To invoke a function that carries out RSA-SSA scheme, the `IppsRSASState` context must be initialized (by the `RSASInit` function) with a proper value of the `flag` parameter:

- For the RSA-SSA signing scheme, `flag == IppRSAPrivate`
- For the RSA-SSA verification scheme, `flag == IppRSAPublic`.

RSASSASign

Carries out the RSA-SSA signature generation scheme.

Syntax

```
ippStatus ippRSASSASign(const Ipp8u* pMsg, int hashLen, const Ipp8u* pSalt, int saltLen,
Ipp8u* pSign, IppsRSASState* pCtx, IppHash hushFunc, IppMGF mgfFunc);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash to be signed.
<i>hashLen</i>	Length of the message hash <i>*pHMsg</i> in octets.
<i>pSalt</i>	Pointer to the random octet salt string
<i>saltLen</i>	Length of the salt string in octets.
<i>pSign</i>	Pointer to the output octet signature.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.
<i>hashFunc</i>	Hash function, which meets the General Definition of a Hash Function provided in chapter 3 .
<i>mgfFunc</i>	MGF, which meets the definition provided in section User's Implementation of a Mask Generation Function in chapter 3 .

Description

This function is declared in the `ippcp.h` file. The function generates the message signature according to the RSASSA-PSS scheme defined in [PKCS 1.2.1]. Intel IPP implementation of the scheme assumes that its first step, that is, computing the hash digest of the original message, is executed prior to the function call and the resulting message hash **pHMsg* is passed to the function. The use of a message hash instead of the original message reduces the length of the function input message, limited by the upper bound of the integer data type $((2^{32} - 1) * 8 \text{ bit})$, and thus enables applying the entire RSA-SSA scheme to input messages of greater lengths. To compute the original message hash to be passed to the function, you should use the same hash function as the one specified by the *hashFunc* parameter and applied in the subsequent steps of the scheme.

The functions specified by the *hashFunc* and *mgfFunc* parameters must be based on the same hash algorithm.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input length parameters do not meet any of the following conditions: <div style="margin-left: 40px;"> $hashLen > 0,$ $saltLen \geq 0,$ $N > hashLen + saltLen + 2$, where N is the length of the RSA modulus in octets. </div>

RSASSASign_MD5

Carries out the RSASSA-PSS signature generation scheme using MD5 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_MD5(const Ipp8u* pHMsg, const Ipp8u* pSalt, int saltLen, Ipp8u* pSign, IppsRSASState* pCtx);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash to be signed.
<i>pSalt</i>	Pointer to the random octet salt string.
<i>saltLen</i>	Length of the salt string in octets.
<i>pSign</i>	Pointer to the output octet signature.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses MD5 hash function and MD5-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general [RSASSASign](#) function. Provided the `IppsRSASState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the [MD5MessageDigest](#) function to compute the message hash to be signed.
2. Call `RSASSASign_MD5` with *pHMsg* pointing to the message hash computed in the previous step.

See the description of the RSASSA-PSS Sign primitive in [\[PKCS 1.2.1\]](#) for more details.

Return Values

The function may return any of the values that the general [RSASSASign](#) function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSASign_SHA1

Carries out the RSASSA-PSS signature generation scheme using SHA-1 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_SHA1(const Ipp8u* pHMsg, const Ipp8u* pSalt, int saltLen, Ipp8u* pSign, IppsRSASState* pCtx);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash to be signed.
<i>pSalt</i>	Pointer to the random octet salt string.

<code>saltLen</code>	Length of the salt string in octets.
<code>pSign</code>	Pointer to the output octet signature.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-1 hash function and SHA-1-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general [RSASSASign](#) function. Provided the `IppsRSASState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the [SHA1MessageDigest](#) function to compute the message hash to be signed.
2. Call `RSASSASign_SHA1` with `pHMsg` pointing to the message hash computed in the previous step.

See the description of the RSASSA-PSS Sign primitive in [PKCS 1.2.1] for more details.

Return Values

The function may return any of the values that the general [RSASSASign](#) function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSASign_SHA224

Carries out the RSASSA-PSS signature generation scheme using SHA-224 hash algorithm.

Syntax

```
IppStatus ippRSASSASign_SHA224(const Ipp8u* pHMsg, const Ipp8u* pSalt, int saltLen, Ipp8u* pSign, IppsRSASState* pCtx);
```

Parameters

<code>pHMsg</code>	Pointer to the octet message hash to be signed.
<code>pSalt</code>	Pointer to the random octet salt string.
<code>saltLen</code>	Length of the salt string in octets.
<code>pSign</code>	Pointer to the output octet signature.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-224 hash function and SHA-224-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general [RSASSASign](#) function. Provided the `IppsRSASState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the [SHA224MessageDigest](#) function to compute the message hash to be signed.
2. Call `RSASSASign_SHA224` with `pHMsg` pointing to the message hash computed in the previous step.

See the description of the RSASSA-PSS Sign primitive in [PKCS 1.2.1] for more details.

Return Values

The function may return any of the values that the general [RSASSASign](#) function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSASign_SHA256

Carries out the RSASSA-PSS signature generation scheme using SHA-256 hash algorithm.

Syntax

```
IppStatus ippRSASSASign_SHA256(const Ipp8u* pHMsg, const Ipp8u* pSalt, int saltLen, Ipp8u* pSign, IppsRSASState* pCtx);
```

Parameters

<code>pHMsg</code>	Pointer to the octet message hash to be signed.
<code>pSalt</code>	Pointer to the random octet salt string.
<code>saltLen</code>	Length of the salt string in octets.
<code>pSign</code>	Pointer to the output octet signature.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-256 hash function and SHA-256-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general [RSASSASign](#) function. Provided the `IppsRSASState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the [SHA256MessageDigest](#) function to compute the message hash to be signed.
2. Call `RSASSASign_SHA256` with `pHMsg` pointing to the message hash computed in the previous step.

See the description of the RSASSA-PSS Sign primitive in [PKCS 1.2.1] for more details.

Return Values

The function may return any of the values that the general [RSASSASign](#) function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSASign_SHA384

Carries out the RSASSA-PSS signature generation scheme using SHA-384 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_SHA384(const Ipp8u* pHMsg, const Ipp8u* pSalt, int saltLen, Ipp8u* pSign, IppsRSASState* pCtx);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash to be signed.
<i>pSalt</i>	Pointer to the random octet salt string.
<i>saltLen</i>	Length of the salt string in octets.
<i>pSign</i>	Pointer to the output octet signature.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-384 hash function and SHA-384-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general [RSASSASign](#) function. Provided the `IppsRSASState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the [SHA384MessageDigest](#) function to compute the message hash to be signed.
2. Call `RSASSASign_SHA384` with *pHMsg* pointing to the message hash computed in the previous step.

See the description of the RSASSA-PSS Sign primitive in [\[PKCS 1.2.1\]](#) for more details.

Return Values

The function may return any of the values that the general [RSASSASign](#) function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSASign_SHA512

Carries out the RSASSA-PSS signature generation scheme using SHA-512 hash algorithm.

Syntax

```
IppStatus ippsRSASSASign_SHA512(const Ipp8u* pHMsg, const Ipp8u* pSalt, int saltLen, Ipp8u* pSign, IppsRSASState* pCtx);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash to be signed.
<i>pSalt</i>	Pointer to the random octet salt string.

<i>saltLen</i>	Length of the salt string in octets.
<i>pSign</i>	Pointer to the output octet signature.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses SHA-512 hash function and SHA-512-based MGF implemented in Intel IPP to generate a message signature according to the RSASSA-PSS scheme and thus is a specific form of the general [RSASSASign](#) function. Provided the `IppsRSASState` context is properly initialized, the entire RSA-SSA signing scheme is carried out in two steps:

1. Call the [SHA512MessageDigest](#) function to compute the message hash to be signed.
2. Call `RSASSASign_SHA512` with *pHMsg* pointing to the message hash computed in the previous step.

See the description of the RSASSA-PSS Sign primitive in [PKCS 1.2.1] for more details.

Return Values

The function may return any of the values that the general [RSASSASign](#) function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSAVerify

Carries out the RSA-SSA signature verification scheme.

Syntax

```
ippStatus ippRSASSAVerify(const Ipp8u* pHMsg, int hashLen, const Ipp8u* pSign, IppBool*
pIsValid, IppsRSASState* pCtx, IppHash hashFunc, IppMGF mgfFunc);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash that has been signed.
<i>hashLen</i>	Length in octets of the message hash <i>*pHMsg</i> .
<i>pSign</i>	Pointer to the octet signature string to be verified.
<i>pIsValid</i>	Pointer to the verification result.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.
<i>hashFunc</i>	Hash function, which meets the General Definition of a Hash Function provided in chapter 3 .
<i>mgfFunc</i>	MGF, which meets the definition provided in section User's Implementation of a Mask Generation Function in chapter 3 .

Description

This function is declared in the `ippcp.h` file. The function carries out the RSASSA-PSS signature verification scheme defined in [PKCS 1.2.1]. `RSASSAVerify` verifies the signature generated by the [RSASSASign](#) function that uses the same *hashFunc* and *mgfFunc* parameters.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsLengthErr</code>	Indicates an error condition if the input length parameter does not meet any of the following conditions: <div style="margin-left: 40px;"> $hashLen > 0$, $hashLen + 2 > N$, where N is the length of the RSA modulus in octets. </div>

RSASSAVerify_MD5

Carries out the RSASSA-PSS signature verification scheme using MD5 hash algorithm.

Syntax

```
ippStatus ippRSASSAVerify_MD5(const Ipp8u* pHMsg, const Ipp8u* pSign, IppBool* pIsValid,
IppsRSASState* pCtx);
```

Parameters

<code>pHMsg</code>	Pointer to the octet message hash that has been signed.
<code>pSign</code>	Pointer to the octet signature string to be verified.
<code>pIsValid</code>	Pointer to the verification result.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses the MD5 hash function and MD5 based MGF implemented in Intel IPP to verify the signature generated by the [RSASSASign_MD5](#) function. `RSASSAVerify_MD5` is a specific form of the general [RSASSAVerify](#) function.

See the description of the RSASSA-PSS Verify primitive in [[PKCS 1.2.1](#)] for more details.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSAVerify_SHA1

Carries out the RSASSA-PSS signature verification scheme using SHA-1 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA1(const Ipp8u* pHMsg, const Ipp8u* pSign, IppBool* pIsValid, IppsRSASState* pCtx);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash that has been signed.
<i>pSign</i>	Pointer to the octet signature string to be verified.
<i>pIsValid</i>	Pointer to the verification result.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses the SHA-1 hash function and SHA-1 based MGF implemented in Intel IPP to verify the signature generated by the [RSASSASign_SHA1](#) function. `RSASSAVerify_SHA1` is a specific form of the general [RSASSAVerify](#) function.

See the description of the RSASSA-PSS Verify primitive in [[PKCS 1.2.1](#)] for more details.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSAVerify_SHA224

Carries out the RSASSA-PSS signature verification scheme using SHA-224 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA224(const Ipp8u* pHMsg, const Ipp8u* pSign, IppBool* pIsValid, IppsRSASState* pCtx);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash that has been signed.
<i>pSign</i>	Pointer to the octet signature string to be verified.
<i>pIsValid</i>	Pointer to the verification result.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses the SHA-224 hash function and SHA-224 based MGF implemented in Intel IPP to verify the signature generated by the [RSASSASign_SHA224](#) function. `RSASSAVerify_SHA224` is a specific form of the general [RSASSAVerify](#) function.

See the description of the RSASSA-PSS Verify primitive in [[PKCS 1.2.1](#)] for more details.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSAVerify_SHA256

Carries out the RSASSA-PSS signature verification scheme using SHA-256 hash algorithm.

Syntax

```
ippStatus ippRSASSAVerify_SHA256(const Ipp8u* pHMsg, const Ipp8u* pSign, IppBool* pIsValid,
IppsRSASState* pCtx);
```

Parameters

<code>pHMsg</code>	Pointer to the octet message hash that has been signed.
<code>pSign</code>	Pointer to the octet signature string to be verified.
<code>pIsValid</code>	Pointer to the verification result.
<code>pCtx</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses the SHA-256 hash function and SHA-256 based MGF implemented in Intel IPP to verify the signature generated by the [RSASSASign_SHA256](#) function. `RSASSAVerify_SHA256` is a specific form of the general [RSASSAVerify](#) function.

See the description of the RSASSA-PSS Verify primitive in [[PKCS 1.2.1](#)] for more details.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSAVerify_SHA384

Carries out the RSASSA-PSS signature verification scheme using SHA-384 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA384(const Ipp8u* pHMsg, const Ipp8u* pSign, IppBool* pIsValid, IppsRSASState* pCtx);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash that has been signed.
<i>pSign</i>	Pointer to the octet signature string to be verified.
<i>pIsValid</i>	Pointer to the verification result.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses the SHA-384 hash function and SHA-384 based MGF implemented in Intel IPP to verify the signature generated by the [RSASSASign_SHA384](#) function. `RSASSAVerify_SHA384` is a specific form of the general [RSASSAVerify](#) function.

See the description of the RSASSA-PSS Verify primitive in [PKCS 1.2.1] for more details.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

RSASSAVerify_SHA512

Carries out the RSASSA-PSS signature verification scheme using SHA-512 hash algorithm.

Syntax

```
IppStatus ippsRSASSAVerify_SHA512(const Ipp8u* pHMsg, const Ipp8u* pSign, IppBool* pIsValid, IppsRSASState* pCtx);
```

Parameters

<i>pHMsg</i>	Pointer to the octet message hash that has been signed.
<i>pSign</i>	Pointer to the octet signature string to be verified.
<i>pIsValid</i>	Pointer to the verification result.
<i>pCtx</i>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function uses the SHA-512 hash function and SHA-512 based MGF implemented in Intel IPP to verify the signature generated by the [RSASSASign_SHA512](#) function. `RSASSAVerify_SHA512` is a specific form of the general [RSASSAVerify](#) function.

See the description of the RSASSA-PSS Verify primitive in [\[PKCS 1.2.1\]](#) for more details.

Return Values

The function may return any of the values that the general `RSASSAVerify` function returns. Specifically:

<code>ippStsLengthErr</code>	Indicates an error condition if any inconsistencies in data lengths are detected.
------------------------------	---

PKCS V1.5 Signature Scheme Functions

This subsection describes functions implementing the RSA signature scheme defined in version 1.5 of the PKCS#1 standard ([\[PKCS 1.2.1\]](#)).

The full list of these functions is given in [Table “Intel IPP PKCS V1.5 Signature Scheme Functions”](#).

Intel IPP PKCS V1.5 Signature Scheme Functions

Function Base Name	Operation
RSASSASign_MD5_PKCSv15	Generates a signature using the v1.5 version of the RSA scheme with the MD5 message digest.
RSASSASign_SHA1_PKCSv15	Generates a signature using the v1.5 version of the RSA scheme with the SHA-1 message digest.
RSASSASign_SHA224_PKCSv15	Generates a signature using the v1.5 version of the RSA scheme with the SHA-224 message digest.
RSASSASign_SHA256_PKCSv15	Generates a signature using the v1.5 version of the RSA scheme with the SHA-256 message digest.
RSASSASign_SHA384_PKCSv15	Generates a signature using the v1.5 version of the RSA scheme with the SHA-384 message digest.
RSASSASign_SHA512_PKCSv15	Generates a signature using the v1.5 version of the RSA scheme with the SHA-512 message digest.
RSASSAVerify_MD5_PKCSv15	Verifies a signature using the v1.5 version of the RSA scheme with the MD5 message digest.
RSASSAVerify_SHA1_PKCSv15	Verifies a signature using the v1.5 version of the RSA scheme with the SHA-1 message digest.
RSASSAVerify_SHA224_PKCSv15	Verifies a signature using the v1.5 version of the RSA scheme with the SHA-224 message digest.
RSASSAVerify_SHA256_PKCSv15	Verifies a signature using the v1.5 version of the RSA scheme with the SHA-256 message digest.
RSASSAVerify_SHA384_PKCSv15	Verifies a signature using the v1.5 version of the RSA scheme with the SHA-384 message digest.
RSASSAVerify_SHA512_PKCSv15	Verifies a signature using the v1.5 version of the RSA scheme with the SHA-512 message digest.

To invoke a function that carries out a PKCS V1.5 signature scheme, the `IppsRSASState` context must be initialized (by the [RSAInit](#) function) with a proper value of the `flag` parameter:

- For the PKCS V1.5 signing scheme, `flag == IppRSAprivate`
- For the PKCS V1.5 verification scheme, `flag == IppRSApublic`.

RSASSASign_MD5_PKCSv15

Generates a signature using the v1.5 version of the RSA scheme with the MD5 message digest.

Syntax

```
IppStatus ippsRSASSASign_MD5_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, Ipp8u* pSign,
IppsRSAState* pRSA);
```

Parameters

<i>pMsg</i>	Pointer to the plain message to be signed.
<i>msgLen</i>	Length of the message. The message can be empty, that is, <i>msgLen</i> ==0.
<i>pSign</i>	Pointer to the output signature.
<i>pRSA</i>	Pointer to the properly initialized <i>IppsRSAState</i> context.

Description

This function is declared in the `ippcp.h` file. The function computes the MD5 digest of the input message and generates the signature of the digest using the v1.5 version of the RSA scheme, defined in [PKCS 1.2.1].



Important. The length of the signature being generated equals the length of the RSA modulus, supplied with the *IppsRSAState* context. Make sure that *pSign* points to a buffer of the appropriate length.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSASign_SHA1_PKCSv15

Generates a signature using the v1.5 version of the RSA scheme with the SHA-1 message digest.

Syntax

```
IppStatus ippsRSASSASign_SHA1_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, Ipp8u* pSign,
IppsRSAState* pRSA);
```

Parameters

<i>pMsg</i>	Pointer to the plain message to be signed.
<i>msgLen</i>	Length of the message. The message can be empty, that is, <i>msgLen</i> ==0.
<i>pSign</i>	Pointer to the output signature.
<i>pRSA</i>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the SHA-1 digest of the input message and generates the signature of the digest using the v1.5 version of the RSA scheme, defined in [PKCS 1.2.1].



Important. The length of the signature being generated equals the length of the RSA modulus, supplied with the `IppsRSAState` context. Make sure that *pSign* points to a buffer of the appropriate length.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSASign_SHA224_PKCSv15

Generates a signature using the v1.5 version of the RSA scheme with the SHA-224 message digest.

Syntax

```
IppStatus ippRSASSASign_SHA224_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, Ipp8u* pSign, IppsRSAState* pRSA);
```

Parameters

<i>pMsg</i>	Pointer to the plain message to be signed.
<i>msgLen</i>	Length of the message. The message can be empty, that is, <i>msgLen</i> ==0.
<i>pSign</i>	Pointer to the output signature.
<i>pRSA</i>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the SHA-224 digest of the input message and generates the signature of the digest using the v1.5 version of the RSA scheme, defined in [PKCS 1.2.1].



Important. The length of the signature being generated equals the length of the RSA modulus, supplied with the `IppsRSASState` context. Make sure that `pSign` points to a buffer of the appropriate length.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSASign_SHA256_PKCSv15

Generates a signature using the v1.5 version of the RSA scheme with the SHA-256 message digest.

Syntax

```
IppStatus ippRSASSASign_SHA256_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, Ipp8u* pSign, IppsRSASState* pRSA);
```

Parameters

<code>pMsg</code>	Pointer to the plain message to be signed.
<code>msgLen</code>	Length of the message. The message can be empty, that is, <code>msgLen==0</code> .
<code>pSign</code>	Pointer to the output signature.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the SHA-256 digest of the input message and generates the signature of the digest using the v1.5 version of the RSA scheme, defined in [PKCS 1.2.1].



Important. The length of the signature being generated equals the length of the RSA modulus, supplied with the `IppsRSASState` context. Make sure that `pSign` points to a buffer of the appropriate length.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSASign_SHA384_PKCSv15

Generates a signature using the v1.5 version of the RSA scheme with the SHA-384 message digest.

Syntax

```
IppStatus ippRSASSASign_SHA384_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, Ipp8u* pSign,
IppsRSAState* pRSA);
```

Parameters

<code>pMsg</code>	Pointer to the plain message to be signed.
<code>msgLen</code>	Length of the message. The message can be empty, that is, <code>msgLen=0</code> .
<code>pSign</code>	Pointer to the output signature.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the SHA-384 digest of the input message and generates the signature of the digest using the v1.5 version of the RSA scheme, defined in [PKCS 1.2.1].



Important. The length of the signature being generated equals the length of the RSA modulus, supplied with the `IppsRSAState` context. Make sure that `pSign` points to a buffer of the appropriate length.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.

`ippStsSizeErr` Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSASign_SHA512_PKCSv15

Generates a signature using the v1.5 version of the RSA scheme with the SHA-512 message digest.

Syntax

```
IppStatus ippRSASSASign_SHA512_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, Ipp8u* pSign,
IppsRSAState* pRSA);
```

Parameters

<code>pMsg</code>	Pointer to the plain message to be signed.
<code>msgLen</code>	Length of the message. The message can be empty, that is, <code>msgLen==0</code> .
<code>pSign</code>	Pointer to the output signature.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function computes the SHA-512 digest of the input message and generates the signature of the digest using the v1.5 version of the RSA scheme, defined in [PKCS 1.2.1].



Important. The length of the signature being generated equals the length of the RSA modulus, supplied with the `IppsRSAState` context. Make sure that `pSign` points to a buffer of the appropriate length.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSAVerify_MD5_PKCSv15

Verifies a signature using the v1.5 version of the RSA scheme with the MD5 message digest.

Syntax

```
IppStatus ippsRSASSAVerify_MD5_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, const Ipp8u* pSign, IppBool* pIsValid, IppsRSAState* pRSA);
```

Parameters

<i>pMsg</i>	Pointer to the input message.
<i>msgLen</i>	Length of the message.
<i>pSign</i>	Pointer to the signature string to be verified.
<i>pIsValid</i>	Pointer to the verification result.
<i>pRSA</i>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function verifies the signature generated by the [ASSASign_MD5_PKCSv15](#) function against the input message.



NOTE. You can verify the signature of an empty message (with `msgLen==0`) against that message. The signature of an empty message is a constant string that depends only on the applied hash algorithm.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSAVerify_SHA1_PKCSv15

Verifies a signature using the v1.5 version of the RSA scheme with the SHA-1 message digest.

Syntax

```
IppStatus ippsRSASSAVerify_SHA1_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, const Ipp8u* pSign, IppBool* pIsValid, IppsRSAState* pRSA);
```

Parameters

<code>pMsg</code>	Pointer to the input message.
<code>msgLen</code>	Length of the message.
<code>pSign</code>	Pointer to the signature string to be verified.
<code>pIsValid</code>	Pointer to the verification result.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function verifies the signature generated by the [RSASSA-ign_SHA1_PKCSv15](#) function against the input message.



NOTE. You can verify the signature of an empty message (with `msgLen==0`) against that message. The signature of an empty message is a constant string that depends only on the applied hash algorithm.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSAVerify_SHA224_PKCSv15

Verifies a signature using the v1.5 version of the RSA scheme with the SHA-224 message digest.

Syntax

```
IppStatus ippRSASSAVerify_SHA224_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, const Ipp8u* pSign, IppBool* pIsValid, IppsRSAState* pRSA);
```

Parameters

<code>pMsg</code>	Pointer to the input message.
<code>msgLen</code>	Length of the message.
<code>pSign</code>	Pointer to the signature string to be verified.
<code>pIsValid</code>	Pointer to the verification result.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function verifies the signature generated by the [RSASSASign_SHA224_PKCSv15](#) function against the input message.



NOTE. You can verify the signature of an empty message (with `msgLen==0`) against that message. The signature of an empty message is a constant string that depends only on the applied hash algorithm.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSAVerify_SHA256_PKCSv15

Verifies a signature using the v1.5 version of the RSA scheme with the SHA-256 message digest.

Syntax

```
ippStatus ippRSASSAVerify_SHA256_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, const Ipp8u* pSign, IppBool* pIsValid, IppsRSAState* pRSA);
```

Parameters

<code>pMsg</code>	Pointer to the input message.
<code>msgLen</code>	Length of the message.
<code>pSign</code>	Pointer to the signature string to be verified.
<code>pIsValid</code>	Pointer to the verification result.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function verifies the signature generated by the [RSASSASign_SHA256_PKCSv15](#) function against the input message.



NOTE. You can verify the signature of an empty message (with `msgLen==0`) against that message. The signature of an empty message is a constant string that depends only on the applied hash algorithm.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSAVerify_SHA384_PKCSv15

Verifies a signature using the v1.5 version of the RSA scheme with the SHA-384 message digest.

Syntax

```
ippStatus ippRSASSAVerify_SHA384_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, const Ipp8u* pSign, IppBool* pIsValid, IppsRSAState* pRSA);
```

Parameters

<code>pMsg</code>	Pointer to the input message.
<code>msgLen</code>	Length of the message.
<code>pSign</code>	Pointer to the signature string to be verified.
<code>pIsValid</code>	Pointer to the verification result.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSAState</code> context.

Description

This function is declared in the `ippcp.h` file. The function verifies the signature generated by the `RSASSASign_SHA384_PKCSv15` function against the input message.



NOTE. You can verify the signature of an empty message (with `msgLen==0`) against that message. The signature of an empty message is a constant string that depends only on the applied hash algorithm.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.

`ippStsSizeErr`

Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

RSASSAVerify_SHA512_PKCSv15

Verifies a signature using the v1.5 version of the RSA scheme with the SHA-512 message digest.

Syntax

```
IppStatus ippRSASSAVerify_SHA512_PKCSv15 (const Ipp8u* pMsg, Ipp32u msgLen, const Ipp8u* pSign, IppBool* pIsValid, IppsRSASState* pRSA);
```

Parameters

<code>pMsg</code>	Pointer to the input message.
<code>msgLen</code>	Length of the message.
<code>pSign</code>	Pointer to the signature string to be verified.
<code>pIsValid</code>	Pointer to the verification result.
<code>pRSA</code>	Pointer to the properly initialized <code>IppsRSASState</code> context.

Description

This function is declared in the `ippcp.h` file. The function verifies the signature generated by the [RSASSASign_SHA512_PKCSv15](#) function against the input message.



NOTE. You can verify the signature of an empty message (with `msgLen==0`) against that message. The signature of an empty message is a constant string that depends only on the applied hash algorithm.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the RSA context parameter does not match the operation.
<code>ippStsInvalidCryptoKeyErr</code>	Indicates an error condition if the RSA context has not been properly set up for the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).

Discrete-Logarithm-Based Cryptography Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) functions allowing for different operations with Discrete Logarithm (DL) based cryptosystem over a prime finite field $GF(p)$. The functions are mainly based on the [IEEE P1363A] standard. Implementation of the Digital Signature operations is based on [FIPS PUB 186-2]. The Diffie-Hellman (DH) Agreement scheme is based on [X9.42].

The full list of Intel IPP DL-based cryptography functions is given in Table “Intel IPP Discrete-Logarithm-Based Cryptography Functions”.

Intel IPP Discrete-Logarithm-Based Cryptography Functions

Function Base Name	Operation
DLPGetSize	Gets the size of the <code>IppsDLPState</code> context.
DLPInit	Initializes user-supplied memory as the <code>IppsDLPState</code> context for future use.
DLPPack , DLPUnpack	Packs/unpacks the <code>IppsDLPState</code> context into/from a user-defined buffer.
DLPSet	Sets up domain parameters of the DL-based cryptosystem over $GF(p)$.
DLPGet	Retrieves domain parameters of the DL-based cryptosystem over $GF(p)$.
DLPSetDP	Sets up a particular domain parameter of the DL-based cryptosystem over $GF(p)$.
DLPGetDP	Retrieves a particular domain parameter of the DL-based cryptosystem over $GF(p)$.
DLPGenKeyPair	Generates a private key and computes public keys of the DL-based cryptosystem over $GF(p)$.
DLPPublicKey	Computes a public key from the given private key of the DL-based cryptosystem over $GF(p)$.
DLPValidateKeyPair	Validates private and public keys of the DL-based cryptosystem over $GF(p)$.
DLPSetKeyPair	Sets private and/or public keys of the DL-based cryptosystem over $GF(p)$.
DLPGenerateDSA	Generates domain parameters of the DL-based cryptosystem over $GF(p)$ to use DSA.
DLPValidateDSA	Validates domain parameters of the DL-based cryptosystem over $GF(p)$ to use DSA.
DLPSignDSA	Performs the DSA digital signature signing operation.
DLPVerifyDSA	Verifies the input DSA digital signature.
DLPGenerateDH	Generates domain parameters of the DL-based cryptosystem over $GF(p)$ to use the DH Agreement scheme.
DLPValidateDH	Validates domain parameters of the DL-based cryptosystem over $GF(p)$ to use the DH Agreement scheme.
DLPSharedSecretDH	Computes a shares secret field element by using the Diffie-Hellman scheme.

All functions described in this section employ the `IppsDLPState` context as operational vehicle that carries domain parameters of the DL cryptosystem, a pair of keys, and working buffers.

The application code intended for executing typical operations should perform the following sequence of operations:

1. Call the function [DLPGetSize](#) to get the size required to configure the `IppsDLPState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the [DLPInit](#) function to initialize the context of the DL-based cryptosystem.
3. Set domain parameters of the DL-based cryptosystem by calling the [DLPSet](#) function, or generate domain parameters by calling the [DLPGenerateDSA](#) or [DLPGenerateDH](#).
4. Call one of the functions [DLPSignDSA](#), [DLPVerifyDSA](#), and [DLPSharedSecretDH](#) to compute digital signature, to verify authenticity of the digital signature, and to compute the shared element accordingly.
5. Free the memory allocated for the `IppsDLPState` context by calling the operating system memory free service function unless the context is no longer needed.

The `IppsDLPState` context is position-dependent. The [DLPPack](#)/[DLPUnpack](#) functions transform the position-dependent context to a position-independent form and vice versa.

DLPGetSize

Gets the size of the `IppsDLPState` context.

Syntax

```
IppStatus ippsDLPGetSize(int peBits, int reBits, int *pSize);
```

Parameters

<i>peBits</i>	Bitsize of the $GF(p)$ element (that is, the length of the DL-based cryptosystem in bits)
<i>reBits</i>	Bitsize of the multiplicative subgroup $GF(r)$.
<i>pSize</i>	Pointer to the <code>IppsDLPState</code> context size in bytes.

Description

This function is declared in the `ippcp.h` file. The function gets the `IppsDLPState` context size in bytes and stores in **pSize*. DL-based cryptosystem over $GF(p)$ assumes that $r/p-1$ where both p and r are primes.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if $peBits \leq reBits$.

DLPInit

Initializes user-supplied memory as the `IppsDLPState` context for future use.

Syntax

```
IppStatus IppsDLPInit(int peBits, int reBits, IppsDLPState* pCtx);
```

Parameters

<i>peBits</i>	Bitsize of the $GF(p)$ element (that is, the length of the DL-based cryptosystem in bits)
<i>reBits</i>	Bitsize of the multiplicative subgroup $GF(r)$.
<i>pCtx</i>	Pointer to the <code>IppsDLPState</code> context being initialized.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory pointed by *pCtx* as the `IppsDLPState` context.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if $peBits \leq reBits$.

DLPPack, DLPUnpack

Packs/unpacks the `IppsDLPState` context into/from a user-defined buffer.

Syntax

```
IppStatus ippDLPPack (const IppsDLPState* pCtx, Ipp8u* pBuffer);
IppStatus ippDLPUnpack (const Ipp8u* pBuffer, IppsDLPState* pCtx);
```

Parameters

<code>pCtx</code>	Pointer to the <code>IppsDLPState</code> context.
<code>pBuffer</code>	Pointer to the user-defined buffer.

Description

This functions are declared in the `ippcp.h` file. The `DLPPack` function transforms the `*pCtx` context to a position-independent form and stores it in the the `*pBuffer` buffer. The `DLPUnpack` function performs the inverse operation, that is, transforms the contents of the `*pBuffer` buffer into a normal `IppsDLPState` context. The `DLPPack` and `DLPUnpack` functions enable replacing the position-dependent `IppsDLPState` context in the memory.

Call the `DLPGetSize` function prior to `DLPPack`/`DLPUnpack` to determine the size of the buffer.

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 <code>NULL</code> .

DLPSet

Sets up domain parameters of the DL-based cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippDLPSet(const IppsBigNumState* pP, const IppBigNumState* pQ, const
IppsBigNumState* pG, IppsDLPState* pCtx);
```

Parameters

<code>pP</code>	Pointer to the characteristic p of the prime finite field $GF(p)$.
<code>pQ</code>	Pointer to the characteristic q of the multiplicative subgroup $GF(q)$.
<code>pG</code>	Pointer to the generator G of the multiplicative subgroup $GF(r)$.

pCtx

Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function sets up DL-based cryptosystem domain parameters into the cryptosystem context.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsRangeErr</code>	Indicates an error condition if any of the Big Numbers specified by <i>pP</i> , <i>pR</i> , and <i>pG</i> is too big to be stored in the <code>IppsDLPState</code> context.

DLPGet

Retrieves domain parameters of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippDLPGet(IppsBigNumState* pP, IppsBigNumState* pQ, IppsBigNumState* pG,
IppsDLPState* pCtx);
```

Parameters

<i>pP</i>	Pointer to the characteristic <i>p</i> of the prime finite field $GF(p)$.
<i>pQ</i>	Pointer to the characteristic <i>q</i> of the multiplicative subgroup $GF(q)$.
<i>pG</i>	Pointer to the generator <i>G</i> of the multiplicative subgroup $GF(r)$.
<i>pCtx</i>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function retrieves DL-based cryptosystem domain parameters into the cryptosystem context.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.

`ippStsRangeErr`

Indicates an error condition if any of the Big Numbers specified by pP , pR , and pG is too small for the DL parameter.

DLPSetDP

Sets up a particular domain parameter of the DL-based cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippSDLPSetDP(const IppsBigNumState* pDP, IppsDLPKeyTag tag, IppsDLPState* pCtx);
```

Parameters

<code>pDP</code>	Pointer to the domain parameter value to be set.
<code>tag</code>	Tag specifying the desired domain parameter.
<code>pCtx</code>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function assigns the value specified by `pDP` to a particular domain parameter of the DL-based cryptosystem. The domain parameter to be set up is determined by `tag` as follows:

- If `tag == IppDLPkeyP`, the function assigns value to the characteristic p , the size of the prime finite field $GF(p)$.
- If `tag == IppDLPkeyR`, the function assigns value to the characteristic r , the prime divisor of $(p-1)$ and the order of g .
- If `tag == IppDLPkeyG`, the function assigns value to the characteristic g , the element of $GF(p)$ generating a multiplicative subgroup of order r .

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsRangeErr</code>	Indicates an error condition if the Big Number specified by <code>pDP</code> is too big to be stored in the <code>IppsDLPState</code> context.
<code>ippStsBadArgErr</code>	Indicates an error condition if some of the function parameters are invalid: <ul style="list-style-type: none"> Big Number specified by <code>pDP</code> is negative Domain parameter specified by <code>tag</code> does not match the <code>IppsDLPState</code> context.

DLPGetDP

Retrieves a particular domain parameter of the DL-based cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippDLGetDP(const IppsBigNumState* pDP, IppsDLPKeyTag tag, IppsDLPState* pCtx);
```

Parameters

<i>pDP</i>	Pointer to the output Big Number context.
<i>tag</i>	Tag specifying the domain parameter to be retrieved.
<i>pCtx</i>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function retrieves value of a particular domain parameter of the DL-based cryptosystem from the *IppsDLPState* context and stores the value in the Big Number context **pDP*. The domain parameter to be retrieved is determined by *tag* as follows:

- If *tag* == `IppDLPkeyP`, the function retrieves value of the characteristic *p*, the size of the prime finite field $GF(p)$.
- If *tag* == `IppDLPkeyR`, the function retrieves value of the characteristic *r*, the prime divisor of $(p-1)$ and the order of *g*.
- If *tag* == `IppDLPkeyG`, the function retrieves value of the characteristic *g*, the element of $GF(p)$ generating a multiplicative subgroup of order *r*.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.
<code>ippStsOutOfRangeErr</code>	Indicates an error condition if the Big Number specified by <i>pDP</i> is too small for the DL parameter.
<code>ippStsBadArgErr</code>	Indicates an error condition if the domain parameter specified by the tag does not match the <i>IppsDLPState</i> context.

DLPGenKeyPair

Generates a private key and computes public keys of the DL-based cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippsDLPGenKeyPair(IppsBigNumState* pPrivate, IppsBigNumState* pPublic,
IppsDLPState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>pCtx</i>	Pointer to the cryptosystem context.
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function generates a private key *privKey* and computes a public key *pubKey* of the DL-based cryptosystem. The function employs specified *rndFunc* Random Generator to generate a pseudorandom private key. The value of the private key *privKey* is a random number that lies in the range of $[2, R-2]$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.
<code>ippStsRangeErr</code>	Indicates an error condition if any of the Big Numbers specified by <i>pPrivate</i> and <i>pPublic</i> is too small for the DL key.

DLPPublicKey

Computes a public key from the given private key of the DL-based cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippsDLPPublicKey(const IppsBigNumState* pPrivate, IppsBigNumState* pPublic,
IppsDLPState* pCtx);
```

Parameters

<i>pPrivate</i>	Pointer to the input private key <i>privKey</i> .
-----------------	---

pPublic Pointer to the output public key *pubKey*.
pCtx Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function computes a public key *pubKey* of the DL-based cryptosystem.

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 the context parameter does not match the operation.
`ippStsIncompleteContextErr` Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsInvalidPrivateKey` Indicates an error condition if the *privKey* has an illegal value.
`ippStsRangeErr` Indicates an error condition if Big Number specified by *pPublic* is too small for the DL public key.

DLPValidateKeyPair

Validates private and public keys of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippDLPValidateKeyPair(const IppsBigNumState* pPrivate, const IppsBigNumState*
pPublic, IppDLPResult* pResult, IppsDLPState* pCtx);
```

Parameters

pPrivate Pointer to the input private key *privKey*.
pPublic Pointer to the output public key *pubKey*.
pResult Pointer to the validation result.
pCtx Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function validates the private key *privKey* and the public key *pubKey* of the DL-based cryptosystem. The result of the validation is stored in the **pResult* and may be assigned to one of the enumerators listed below:

`ippDLValid` Validation has passed successfully.
`ippDLInvalidPrivateKey` $(1 < private < (R - 1))$ is false.
`ippDLInvalidPublicKey` $(1 < public \leq (P - 1))$ is false.
`ippDLInvalidKeyPair` $public \neq G^{private} \pmod{P}$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.

DLPSetKeyPair

Sets private and/or public keys of the DL-based cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippDLPSetKeyPair(const IppsBigNumState* pPrivate, const IppsBigNumState* pPublic,
IppsDLPState* pCtx);
```

Parameters

<code>pPrivate</code>	Pointer to the input private key <i>privKey</i> .
<code>pPublic</code>	Pointer to the output public key <i>pubKey</i> .
<code>pCtx</code>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function stores the private key *privKey* and public key *pubKey* in the cryptosystem context.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.

DLPGenerateDSA

Generates domain parameters of the DL-based cryptosystem over $GF(p)$ to use DSA.

Syntax

```
IppStatus ippDLPGenerateDSA(const IppsBigNumState* pSeedIn, int nTrials, IppsDLPState*
pCtx, IppsBigNumState* pSeedOut, int* pCounter, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>pSeedIn</i>	Pointer to the input <i>Seed</i> .
<i>nTrials</i>	Security parameter specified for the Miller-Rabin probable primality.
<i>pCtx</i>	Pointer to the cryptosystem context.
<i>pSeedOut</i>	Pointer to the output <i>Seed</i> value (if requested).
<i>pCounter</i>	Pointer to the <i>counter</i> value (if requested).
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function generates domain parameters of the DL-based cryptosystem over $GF(p)$ to use DSA. The function uses a procedure specified in [FIPS PUB 186-2] for generating both a 160-bit randomized prime r and a *LpeBits* prime p based on the input **pSeedIn*.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if: <i>peBits</i> < 512, <i>peBits</i> is not divided by 64, <i>reBits</i> != 160.
<code>ippStsRangeErr</code>	Indicates an error condition if: bitsize of the input <i>Seed</i> value is less than 160, bitsize of the input <i>Seed</i> value is greater than <i>peBits</i> , not enough space to store the output <i>Seed</i> value (if requested).
<code>ippStsBadArgErr</code>	Indicates an error condition if <i>nTrials</i> < 1.
<code>ippStsInsuffucientEntropy</code>	Indicates a warning condition if prime generation fails due to a poor choice of the entropy.

DLPValidateDSA

Validates domain parameters of the DL-based cryptosystem over $GF(p)$ to use DSA.

Syntax

```
IppStatus ippDLValidateDSA(int nTrials, IppDLResult* pResult, IppsDLPState* pCtx,
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>nTrials</i>	Security parameter specified for the Miller-Rabin probable primality.
----------------	---

<i>pResult</i>	Pointer to the validation result.
<i>pCtx</i>	Pointer to the cryptosystem context.
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function validates domain parameters of the DL-based cryptosystem over $GF(p)$ to use DSA. The result of validation is stored in the **pResult* and may be assigned to one of the enumerators listed below:

<code>ippDLValid</code>	Validation has passed successfully.
<code>ippDLBaseIsEven</code>	P is even.
<code>ippDLOrderIsEven</code>	R is even.
<code>ippDLInvalidBaseRange</code>	$P \leq 2^{peBits-1}$ or $P \geq 2^{peBits}$.
<code>ippDLInvalidOrderRange</code>	$R \leq 2^{reBits-1}$ or $R \geq 2^{reBits}$.
<code>ippDLCompositeBase</code>	P is not a prime.
<code>ippDLCompositeOrder</code>	R is not a prime.
<code>ippDLInvalidCofactor</code>	R is not divisible by $(P - 1)$.
<code>ippDLInvalidGenerator</code>	$(1 < G < (P - 1))$ is false or $G^R \neq 1 \pmod{P}$.

To ensure that both p and r are primes, the function applies $nTrial$ -round Miller-Rabin primality test. Test data for primality test is provided by the specified *rndFunc* Random Generator.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.
<code>ippStsBadArgErr</code>	Indicates an error condition if $nTrials < 1$.

DLPSignDSA

Performs the DSA digital signature signing operation.

Syntax

```
IppStatus ippDLPSignDSA(const IppsBigNumState* pMsg, const IppsBigNumState* pPrivate,
IppsBigNumState* pSignR, IppsBigNumState* pSignS, IppsDLPState* pCtx);
```

Parameters

<i>pMsg</i>	Pointer to the message representation <i>msgRep</i> to be signed.
-------------	---

<i>pPrivate</i>	Pointer to the signer's private key <i>privKey</i> .
<i>pSignR</i>	Pointer to the <i>r</i> -component of the signature.
<i>pSignS</i>	Pointer to the <i>s</i> -component of the signature.
<i>pCtx</i>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function performs the DSA digital signature signing operation provided that the ephemeral signer's key pair (both private and public) was previously computed (generated by [DLPGenKeyPair](#) or computed by [DLPPublicKey](#)) and then set up into the DLP context by the [DLPSetKeyPair](#) function.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.
<code>ippStsMessageErr</code>	Indicates an error condition if the value of <i>msgRep</i> is greater than the multiplicative subgroup characteristic (<i>q</i>).
<code>ippStsInvalidPrivateKey</code>	Indicates an error condition if an illegal value has been assigned to <i>privKey</i> .
<code>ippStsRangeErr</code>	Indicates an error condition if any of the signature components has not enough space.

DLPVerifyDSA

Verifies the input DSA digital signature.

Syntax

```
ippStatus ippDLPVerifyDSA(const IppsBigNumState* pMsg, const IppsBigNumState* pSignR,
const IppsBigNumState* pSignS, IppDLResult* pResult, IppsDLPState* pCtx);
```

Parameters

<i>pMsg</i>	Pointer to the message representation <i>msgRep</i> .
<i>pSignR</i>	Pointer to the signature <i>r</i> -component to be verified.
<i>pSignS</i>	Pointer to the signature <i>s</i> -component to be verified.
<i>pResult</i>	Pointer to the result of the verification.
<i>pCtx</i>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function verifies the input DSA digital signature's components **pSignR* and **pSignS* with the supplied message representation *msgRep*. Signer's public key must be stored by the `DLPSetKeyPair` function before the `DLPVerifyDSA` operation.

The function sets the **pResult* to `ippDLValid` if it validates the input DSA digital signature, or to `ippDLInvalidSignature` if the DSA digital signature verification fails.

[Example](#) illustrates the use of functions [DLPSignDSA](#) and `DLPVerifyDSA`. The example uses the `BigNumber` class and functions creating some cryptographic contexts, whose source code can be found in [Appendix B](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.
<code>ippStsMessageErr</code>	Indicates an error condition if the value of <i>msgRep</i> is greater than the multiplicative subgroup characteristic (<i>q</i>).

Example of Using RSA Primitive Functions

Use of DLPSignDSA and DLPVerifyDSA

```
//
// known domain parameters
//
static const int M = 512; // DSA system bitsize
static const int L = 160; // DSA order  bitsize
static
BigNumber P("0x8DF2A494492276AA3D25759BB06869CBEAC0D83AFB8D0CF7" \
            "CBB8324F0D7882E5D0762FC5B7210EAF2E9ADAC32AB7AAC" \
            "49693DFBF83724C2EC0736EE31C80291");

static
BigNumber Q("0xC773218C737EC8EE993B4F2DED30F48EDACE915F");

static
BigNumber G("0x626D027839EA0A13413163A55B4CB500299D5522956CEFCB" \
            "3BFF10F399CE2C2E71CB9DE5FA24BABF58E5B79521925C9C" \
            "C42E9F6F464B088CC572AF53E6D78802");

//
// known DSA regular key pair
//
static
BigNumber X("0x2070B3223DBA372FDE1C0FFC7B2E3B498B260614");

static
BigNumber Y("0x19131871D75B1612A819F29D78D1B0D7346F7AA77BB62A85" \
            "9BFD6C5675DA9D212D3A36EF1672EF660B8C7C255CC0EC74" \
            "858FBA33F44C06699630A76B030EE333");
```

```

int DSAsign_verify_sample(void)
{
    // DLP context
    IppsDLPState *DLPState = newDLP(M, L);

    // set up DLP crypto system
    ippsDLPSet(P, Q, G, DLPState);

    // message
    Ipp8u message[] = "abc";

    // compute message digest to be signed
    Ipp8u md[SHA1_DIGEST_LENGTH/8];
    ippsSHA1MessageDigest(message, sizeof(message)-1, md);
    BigNumber digest(0, BITS_2_WORDS(SHA1_DIGEST_LENGTH));
    ippsSetOctString_BN(md, SHA1_DIGEST_LENGTH/8, digest);

    // generate ephemeral key pair (ephX,ephY)
    BigNumber ephX(0, BITS_2_WORDS(L));
    BigNumber ephY(0, BITS_2_WORDS(M));

    IppsPRNGState* pRand = newPRNG();
    ippsDLPGenKeyPair(ephX, ephY, DLPState, ippsPRNGen, pRand);
    deletePRNG(pRand);
    //
    // generate signature
    //
    BigNumber signR(0, BITS_2_WORDS(L)); // R and S signature's component
    BigNumber signS(0, BITS_2_WORDS(L));
    ippsDLPSetKeyPair(ephX, ephY, DLPState); // set up ephemeral keys
    ippsDLPSignDSA(digest, X, // sign digest
                  signR, signS,
                  DLPState);

```

```

//
// verify signature
//
ippsDLPSetKeyPair(0, Y, DLPState);          // set up regular public key
IppDLResult result;
ippsDLPVerifyDSA(digest, signR, signS,      // verify
                 &result, DLPState);

deleteDLP(DLPState);
return result==ippDLValid;
}

```

DLPGenerateDH

Generates domain parameters of the DL-based cryptosystem over $GF(p)$ to use the DH Agreement scheme.

Syntax

```
IppStatus IppsDLPGenerateDH(const IppsBigNumState* pSeedIn, int nTrials, IppsDLPState*
pCtx, IppsBigNumState* pSeedOut, int* pCounter, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>pSeedIn</i>	Pointer to the input <i>Seed</i> .
<i>nTrials</i>	Security parameter specified for the Miller-Rabin probable primality.
<i>pCtx</i>	Pointer to the cryptosystem context.
<i>pSeedOut</i>	Pointer to the output <i>Seed</i> value (if requested).
<i>pCounter</i>	Pointer to the <i>counter</i> value (if requested).
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function generates domain parameters of the DL-based cryptosystem over $GF(p)$ to use Diffie-Hellman Agreement scheme. The function uses a procedure specified in [X9.42] for generating both randomized prime p and r based on the input **pSeedIn*.

Generated primes r and p are further validated through a *nTrial*-round Miller-Rabin primality test. Both generation and primality test procedures employ specified *rndFunc* Random Generator.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if: $peBits < 512$ or $reBits < 160$, $peBits$ is not divided by 256.
<code>ippStsRangeErr</code>	Indicates an error condition if: bitsize of the input <i>Seed</i> value is less than $reBits$, not enough space to store the output <i>Seed</i> value (if requested).
<code>ippStsBadArgErr</code>	Indicates an error condition if $nTrials < 1$.
<code>ippStsInsuffucientEntropy</code>	Indicates a warning condition if prime generation fails due to a poor choice of the entropy.

DLPValidateDH

Validates domain parameters of the DL-based cryptosystem over $GF(p)$ to use the DH Agreement scheme.

Syntax

```
ippStatus ippDLValidateDH(int nTrials, IppDLResult* pResult, IppsDLPState* pCtx,
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<code>nTrials</code>	Security parameter specified for the Miller-Rabin probable primality.
<code>pResult</code>	Pointer to the validation result.
<code>pCtx</code>	Pointer to the cryptosystem context.
<code>rndFunc</code>	Specified Random Generator.
<code>pRndParam</code>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file. The function validates domain parameters of the DL-based cryptosystem over $GF(p)$ to use Diffie-Hellman Agreement scheme. The result of validation is stored in the `*pResult` and may be assigned to one of the enumerators listed below:

<code>ippDLValid</code>	Validation has passed successfully.
<code>ippDLBaseIsEven</code>	P is even.
<code>ippDLOrderIsEven</code>	R is even.
<code>ippDLInvalidBaseRange</code>	$P \leq 2^{peBits-1}$ or $P \geq 2^{peBits}$.
<code>ippDLInvalidOrderRange</code>	$R \leq 2^{reBits-1}$ or $R \geq 2^{reBits}$.

<code>ippDLCompositeBase</code>	P is not a prime.
<code>ippDLCompositeOrder</code>	R is not a prime.
<code>ippDLInvalidCofactor</code>	R is not divisible by $(P - 1)$.
<code>ippDLInvalidGenerator</code>	$(1 < G < (P - 1))$ is false or $G^R \neq 1 \pmod{P}$.

To ensure that both p and r are primes, the function applies $nTrial$ -round Miller-Rabin primality test. Test data for primality test is provided by the specified `rndFunc` Random Generator.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.
<code>ippStsBadArgErr</code>	Indicates an error condition if $nTrials < 1$.

DLPSharedSecretDH

Computes a shared field element by using the Diffie-Hellman scheme.

Syntax

```
IssStatus ippDLPSharedSecretDH(const IppsBigNumState* pPrivateA, const IppsBigNumState*
pPublicB, IppsBigNumState* pShare, IppsDLPState* pCtx);
```

Parameters

<code>pPrivateA</code>	Pointer to your own private key <code>privateKeyA</code> .
<code>pPublicB</code>	Pointer to the public key <code>pubKeyB</code> belonging to the other party.
<code>pShare</code>	Pointer to the shared secret element <code>Share</code> .
<code>pCtx</code>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file. The function computes a shared secret element $FG(p)$ $pubKeyB^{privateKeyA} \pmod{p}$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the context parameter does not match the operation.
<code>ippStsIncompleteContextErr</code>	Indicates an error condition if the cryptosystem context has not been properly set up.

ippStsRangeErr

Indicates an error condition if *Share* does not have enough space.

Elliptic Curve Cryptography Functions

Intel® Integrated Performance Primitives (Intel® IPP) for cryptography offer functions allowing for different operations with an elliptic curve defined over a prime finite field $GF(p)$ and binary finite field $GF(2^m)$. The functions are based on standards [IEEE P1363A], [SEC1], and [ANSI]. For more information on parameters recommended for the functions, see [SEC2].

The full list of Elliptic Curve Cryptography functions is given in [Table “Intel IPP Elliptic Curve Cryptography Functions”](#).

Intel IPP Elliptic Curve Cryptography Functions

Function Base Name	Operation
Functions Operating over $GF(p)$	
ECCPGetSize	Gets the size of the <code>IppsECCPState</code> context.
ECCPInit	Initializes context for the elliptic curve cryptosystem over $GF(p)$.
ECCPSet	Sets up elliptic curve domain parameters over $GF(p)$.
ECCPSetStd	Sets up a recommended set of elliptic curve domain parameters over $GF(p)$.
ECCPGet	Retrieves elliptic curve domain parameters over $GF(p)$.
ECCPGetOrderBitSize	Retrieves order size of the elliptic curve base point over $GF(p)$ in bits
ECCPValidate	Checks validity of the elliptic curve domain parameters over $GF(p)$.
ECCPPointGetSize	Gets the size of the <code>IppsECCPPoint</code> context in bytes for a point on the elliptic curve point defined over $GF(p)$.
ECCPPointInit	Initializes context for a point on the elliptic curve defined over $GF(p)$.
ECCPSetPoint	Sets coordinates of a point on the elliptic curve defined over $GF(p)$.
ECCPSetPointAtInfinity	Sets the point at infinity.
ECCPGetPoint	Retrieves coordinates of the point on the elliptic curve defined over $GF(p)$.
ECCPCheckPoint	Checks correctness of the point on the elliptic curve defined over $GF(p)$.
ECCPComparePoint	Compares two points on the elliptic curve defined over $GF(p)$.
ECCPNegativePoint	Finds an elliptic curve point which is an additive inverse for the given point over $GF(p)$.
ECCPAddPoint	Computes the addition of two elliptic curve points over $GF(p)$.
ECCPMulPointScalar	Performs scalar multiplication of a point on the elliptic curve defined over $GF(p)$.
ECCPGenKeyPair	Generates a private key and computes public keys of the elliptic cryptosystem over $GF(p)$.
ECCPPublicKey	Computes a public key from the given private key of the elliptic cryptosystem over $GF(p)$.
ECCPValidateKeyPair	Validates private and public keys of the elliptic cryptosystem over $GF(p)$.

Function Base Name	Operation
ECCPSetKeyPair	Sets private and/or public keys of the elliptic cryptosystem over $GF(p)$.
ECCPSharedSecretDH	Computes a shared secret field element by using the Diffie-Hellman scheme.
ECCPSharedSecretDHC	Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.
ECCPSignDSA	Computes a digital signature over a message digest.
ECCPVerifyDSA	Verifies authenticity of the digital signature over a message digest (ECDSA).
ECCPSignNR	Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).
ECCPVerifyNR	Verifies authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).
Functions Operating over $GF(2^m)$	
ECCBGetSize	Gets the size of the <code>IppsECCBState</code> context.
ECCBInit	Initializes context for the elliptic curve cryptosystem over $GF(2^m)$.
ECCBSet	Sets up elliptic curve domain parameters over $GF(2^m)$.
ECCBSetStd	Sets up a recommended set of elliptic curve domain parameters over $GF(2^m)$.
ECCBGet	Retrieves elliptic curve domain parameters over $GF(2^m)$.
ECCBGetOrderBitSize	Retrieves order size of the elliptic curve base point over $GF(2^m)$ in bits.
ECCBValidate	Checks validity of the elliptic curve domain parameters over $GF(2^m)$.
ECCBPointGetSize	Gets the size of the <code>IppsECCBPoint</code> context in bytes for a point on the elliptic curve point defined over $GF(2^m)$.
ECCBPointInit	Initializes context for a point on the elliptic curve defined over $GF(2^m)$.
ECCBSetPoint	Sets coordinates of a point on the elliptic curve defined over $GF(2^m)$.
ECCBSetPointAtInfinity	Sets the point at infinity.
ECCBGetPoint	Retrieves coordinates of the point on the elliptic curve defined over $GF(2^m)$.
ECCBCheckPoint	Checks correctness of the point on the elliptic curve defined over $GF(2^m)$.
ECCBComparePoint	Compares two points on the elliptic curve defined over $GF(2^m)$.
ECCBNegativePoint	Finds an elliptic curve point which is an additive inverse for the given point over $GF(2^m)$.
ECCBAddPoint	Computes the addition of two elliptic curve points over $GF(2^m)$.
ECCBMulPointScalar	Performs scalar multiplication of a point on the elliptic curve defined over $GF(2^m)$.
ECCBGenKeyPair	Generates a private key and computes public keys of the elliptic cryptosystem over $GF(2^m)$.
ECCBPublicKey	Computes a public key from the given private key of the elliptic cryptosystem over $GF(2^m)$.

Function Base Name	Operation
ECCBValidateKeyPair	Validates private and secret keys of the elliptic cryptosystem over $GF(2^m)$.
ECCBSetKeyPair	Sets private and/or public keys in the elliptic cryptosystem over $GF(2^m)$.
ECCBSharedSecretDH	Computes a shared secret field element by using the Diffie-Hellman scheme.
ECCBSharedSecretDHC	Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.
ECCBSignDSA	Computes a digital signature over a message digest.
ECCBVerifyDSA	Verifies authenticity of the digital signature over a message digest (ECDSA).
ECCBSignNR	Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).
ECCBVerifyNR	Computes authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).

Public key cryptography successfully allows to solve problems of information security by enabling secure communication over insecure channels. Although elliptic curves are well studied as a branch of mathematics, an interest to the cryptographic schemes based on elliptic curves is constantly rising due to the advantages that the elliptic curve algorithms provide in the wireless communications: shorter processing time and key length.

Elliptic curve cryptosystems (ECCs) implement a different way of creating public keys. Because elliptic curve calculation is based on the addition of the rational points in the (x,y) plane and it is difficult to solve a discrete logarithm from these points, a higher level of security is achieved through the cryptographic schemes that use the elliptic curves. The cryptographic systems that encrypt messages by using the properties of elliptic curves are hard to attack due to the extreme complexity of deciphering the private key.

Use of elliptic curves allows for shorter public key length and encourage cryptographers to create cryptosystems with the same or higher encryption strength as the RSA or DSA cryptosystems. Because of the relatively short key length, ECCs do encryption and decryption faster on the hardware that requires less computation processing volumes. For example, with a key length of 150-350 bits, ECCs provide the same encryption strength as the cryptosystems who have to use 600 -1400 bits. Alternatively, it requires considerably less time to create a digital signature with the ECDSA (Elliptic Curve Digital Signature Algorithm) algorithm rather than with the RSA algorithm even though you are using the same processor machine.

Functions Based on $GF(p)$

This section describes functions designed to specify the elliptic curve cryptosystem and perform various operations on the elliptic curve defined over a prime finite field. The examples of the operations are shown below:

1. Setting up operations [ECCPSet](#) sets up elliptic curve domain parameters. [ECCPSetKeyPair](#) sets a pair of public and private keys for the given cryptosystem.
2. Computation operations [ECCPAddPoint](#) adds two points on the elliptic curve. [ECCPMulPointScalar](#) performs the scalar multiplication of a point on the elliptic curve. [ECCPSignDSA](#) computes the digital signature of a message.
3. Validation operations [ECCPValidate](#) checks validity of the elliptic curve domain parameters. [ECCPValidateKeyPair](#) validates correctness of the public and private keys.
4. Generation operations [ECCPGenKeyPair](#) generates a private key and computes a public key for the given elliptic cryptosystem.
5. Retrieval operations [ECCPGet](#) retrieves elliptic curve domain parameters. [ECCPGetOrderBitSize](#) retrieves the size of a base point in bytes.

All functions described in this section employ a context `IppsECCPState` that catches several auxiliary components specifying operations performed on the elliptic curve or entire elliptic cryptosystem. ECCP stands for Elliptic Curve Cryptography Prime and means that all functions whose name include this abbreviation perform operations over a prime finite field $GF(p)$.

ECCPGetSize

Gets the size of the `IppsECCPState` context.

Syntax

```
IppStatus ippsECCPGetSize(int feBitSize, int *pSize);
```

Parameters

<code>feBitSize</code>	Size (in bits) of the field element.
<code>pSize</code>	Pointer to the size (in bytes) of the context.

Description

This function is declared in the `ippcp.h` file.

The function computes the size of the context in bytes for the elliptic cryptosystem over a prime finite field $GF(p)$.

Context is a structure `IppsECCPState` designed to store information about the cryptosystem status.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if the value of the parameter <code>feBitSize</code> is less than 2.

ECCPInit

Initializes context for the elliptic curve cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippsECCPInit(int feBitSize, IppsECCPState* pECC);
```

Parameters

<code>feBitSize</code>	Size (in bits) of a field element.
<code>pECC</code>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file.

The function initializes the context of the elliptic curve cryptosystem over the prime finite field $GF(p)$.

Context is a structure `IppsECCPState` designed to store information about the cryptosystem status.

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>ippStsSizeErr</code>	Indicates an error condition if the value of the parameter <i>feBitSize</i> is less than 2.

ECCPSet

Sets up elliptic curve domain parameters over $GF(p)$.

Syntax

```
IppStatus ippseccpset(const IppsBigNumState* pPrime, const IppsBigNumState* pA, const
IppsBigNumState* pB, const IppsBigNumState* pGX, const IppsBigNumState* pGY, const
IppsBigNumState* pOrder, int cofactor, IppsECCPState* pECC);
```

Parameters

<i>pPrime</i>	Pointer to the characteristic p of the prime finite field $GF(p)$.
<i>pA</i>	Pointer to the coefficient A of the equation defining the elliptic curve.
<i>pB</i>	Pointer to the coefficient B of the equation defining the elliptic curve.
<i>pGX</i>	Pointer to the x-coordinate of the elliptic curve base point.
<i>pGY</i>	Pointer to the y-coordinate of the elliptic curve base point.
<i>pOrder</i>	Pointer to the order of the elliptic curve base point.
<i>cofactor</i>	Cofactor.
<i>pECC</i>	Pointer to the context of the cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function sets up the elliptic curve domain parameters over a prime finite field $GF(p)$. These are as follows:

- *pPrime* sets up the characteristic p of a finite field $GF(p)$ where p is a prime number.
- *pA*, *pB* set up the coefficients A and B of the equation defining the elliptic curve:

$$y^2 = x^3 + A \cdot x + B \pmod{p}.$$
- *pGX*, *pGY* are pointers to the affine coordinates of the elliptic curve base point G .
- *pOrder* is a pointer to the order n of the elliptic curve base point G such that $n \cdot G = O$, where O is the point at infinity and n is a prime number.
- *cofactor* sets up the ratio h of a general number of points $\#E$ on the elliptic curve (including the point at infinity) to the order n of the base point:

$$h = \#E/n.$$

The domain parameters are set in the cryptosystem context which must be already created by the [ECCPGetSize](#) and [ECCPInit](#) functions.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pPrime</code> , <code>pA</code> , <code>pB</code> , <code>pGX</code> , <code>pGY</code> , <code>pOrder</code> , and <code>pECC</code> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if of one of the parameters pointed by <code>pPrime</code> , <code>pA</code> , <code>pB</code> , <code>pGX</code> , <code>pGY</code> , and <code>pOrder</code> cannot embed the <code>feBitSize</code> bits length or the value of <code>cofactor</code> is less than 1.

ECCPSetStd

Sets up a recommended set of elliptic curve domain parameters over GF(p).

Syntax

```
IppStatus ippseECCPSetStd(IppECCType flag, IppsECCPState* pECC);
```

Parameters

<code>flag</code>	Set specifier.
<code>pECC</code>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file.

The function sets a recommended set of elliptic curve domain parameters over a prime finite field GF(p).

The set is defined by the value of the parameter `flag`. Possible values of the parameter are as follows:

<code>IppECCPStd112r1</code>	For the cryptosystem context where <code>feBitSize==112</code>
<code>IppECCPStd112r2</code>	For the cryptosystem context where <code>feBitSize==112</code>
<code>IppECCPStd128r1</code>	For the cryptosystem context where <code>feBitSize==128</code>
<code>IppECCPStd128r2</code>	For the cryptosystem context where <code>feBitSize==128</code>
<code>IppECCPStd160r1</code>	For the cryptosystem context where <code>feBitSize==160</code>
<code>IppECCPStd160r2</code>	For the cryptosystem context where <code>feBitSize==160</code>
<code>IppECCPStd192r1</code>	For the cryptosystem context where <code>feBitSize==192</code>
<code>IppECCPStd224r1</code>	For the cryptosystem context where <code>feBitSize==224</code>
<code>IppECCPStd256r1</code>	For the cryptosystem context where <code>feBitSize==256</code>
<code>IppECCPStd384r1</code>	For the cryptosystem context where <code>feBitSize==384</code>
<code>IppECCPStd521r1</code>	For the cryptosystem context where <code>feBitSize==521</code> .

For more information on parameter values for the recommended elliptic curves, see [\[SEC2\]](#).

The cryptosystem context must be already created by the [ECCPGetSize](#) and [ECCPInit](#) functions. The value of *feBitSize* is applied when these functions are called and predetermines the possible choice of the *flag* value.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the cryptosystem context is not valid.
<code>ippStsECCInvalidFlagErr</code>	Indicates an error condition if the value of the parameter <i>flag</i> is not valid.

ECCPGet

Retrieves elliptic curve domain parameters over $GF(p)$.

Syntax

```
IppStatus ippseccpGet(IppsBigNumState* pPrime, IppsBigNumState* pA, IppsBigNumState* pB,
IppsBigNumState* pGX, IppsBigNumState* pGY, IppsBigNumState* pOrder, int* cofactor,
IppsECCPState* pECC);
```

Parameters

<i>pPrime</i>	Pointer to the characteristic p of the prime finite field $GF(p)$.
<i>pA</i>	Pointer to the coefficient A of the equation defining the elliptic curve.
<i>pB</i>	Pointer to the coefficient B of the equation defining the elliptic curve.
<i>pGX</i>	Pointer to the x -coordinate of the elliptic curve base point.
<i>pGY</i>	Pointer to the y -coordinate of the elliptic curve base point.
<i>pOrder</i>	Pointer to the order n of the elliptic curve base point.
<i>cofactor</i>	Pointer to the cofactor h .
<i>pECC</i>	Pointer to the context of the cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function retrieves elliptic curve domain parameters from the context of the elliptic cryptosystem over a finite field $GF(p)$ and allocates them in accordance with the pointers *pPrime*, *pA*, *pB*, *pGX*, *pGY*, *pOrder*, and *cofactor*. The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .

<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pPrime</code> , <code>pA</code> , <code>pB</code> , <code>pGX</code> , <code>pGY</code> , <code>pOrder</code> , or <code>pECC</code> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if the memory size of one of the parameters pointed by <code>pPrime</code> , <code>pA</code> , <code>pB</code> , <code>pGX</code> , <code>pGY</code> , <code>pOrder</code> , and <code>pECC</code> is less than the value of <code>feBitSize</code> in the <code>ECCPInit</code> function.

ECCPGetOrderBitSize

Retrieves order size of the elliptic curve base point over $GF(p)$ in bits.

Syntax

```
IppStatus ippseccpGetOrderBitSize(int* pBitSize, IppsECCPState* pECC);
```

Parameters

<code>pBitSize</code>	Pointer to the size of the base point (in bits).
<code>pECC</code>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file.

The function retrieves the order size (in bits) of the elliptic curve base point G from the context of elliptic cryptosystem over a prime finite field $GF(p)$ and allocates it in accordance with the pointer `pBitsSize`. The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCPSet` or `ECCPSetStd`.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the cryptosystem context is not valid.

ECCPValidate

Checks validity of the elliptic curve domain parameters over $GF(p)$.

Syntax

```
IppStatus ippseccpValidate(int nTrials, IppECResult* pResult, IppsECCPState* pECC,
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<code>nTrials</code>	A number of attempts made to check the number for primality.
<code>pResult</code>	Pointer to the result received upon the check of the elliptic curve domain parameters.

<i>pECC</i>	Pointer to the cryptosystem context.
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to Random Generator context.

Description

This function is declared in the `ippcp.h` file.

The function checks validity of the elliptic curve domain parameters over a prime finite field $GF(p)$ and stores the result of the check in accordance with the pointer *pResult*.

Elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#). The purpose of the parameters *rndFunc*, *pRndParam*, and *nTrials* is analogous to that of the parameters *rndFunc*, *pRndParam*, and *nTrials* in the [PrimeTest](#) function.

The result of the elliptic curve domain parameters check can take one of the following values:

<code>ippECValid</code>	The parameters are valid.
<code>ippECCompositeBase</code>	The prime finite field characterisitc p is a composite number.
<code>ippECIsNotAG</code>	The solutions of the elliptic curve equation do not form the abelian group because the only requirement that $4 \cdot a^3 + 27 \cdot b^3 \neq 0$ is not met.
<code>ippECPPointIsValid</code>	The base point G is not on the elliptic curve.
<code>ippECCompositeOrder</code>	The order n of the base point G is a composite number.
<code>ippECInvalidOrder</code>	The order n of the base point G is not valid because the requirement that $n \cdot G = O$ where O is the point at infinity is not met.
<code>ippECIsWeakSSSA</code>	The order n of the base point G is equal to the finite field characteristic p .
<code>ippECIsWeakMOV</code>	The curve is excluded because it is subject to the MOV reduction attack.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>c</i> or <i>pECC</i> is not valid.
<code>ippStsBadArgErr</code>	Indicates an error condition if the memory size of the parameter <i>seed</i> is less than five words (32 bytes in each) or the value of the parameter <i>nTrials</i> is less than 1.

ECCPointGetSize

Gets the size of the `IppsECCPoint` context in bytes for a point on the elliptic curve point defined over $GF(p)$.

Syntax

```
IppStatus ippsECCPointGetSize(int feBitSize, int* pSize);
```

Parameters

<i>feBitSize</i>	Size (in bits) of the field element.
<i>pSize</i>	Pointer to the context size.

Description

This function is declared in the `ippcp.h` file. The function computes the context size in bytes for a point on the elliptic curve defined over a prime finite field $GF(p)$.

Context is a structure `IppsECCPoint` intended for storing the information about a point on the elliptic curve defined over $GF(p)$.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if the value of the parameter <i>feBitSize</i> is less than 2.

ECCPointInit

Initializes the context for a point on the elliptic curve defined over $GF(p)$.

Syntax

```
IppStatus ippsECCPointInit(int feBitSize, IppsECCPoint* pPoint);
```

Parameters

<i>feBitSize</i>	Size (in bits) of the field element.
<i>pECC</i>	Pointer to the context of the elliptic curve point.

Description

This function is declared in the `ippcp.h` file. The function initializes the context for a point on the elliptic curve defined over a finite field $GF(p)$.

Context is a structure `IppsECCPoint` intended for storing the information about a point on the elliptic curve defined over $GF(p)$.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if the value of the parameter <code>feBitSize</code> is less than 2.

ECCPSetPoint

Sets coordinates of a point on the elliptic curve defined over $GF(p)$.

Syntax

```
IppStatus ippseccpSetPoint(const IppsBigNumState* pX, const IppsBigNumState* pY,
IppsECCPPoint* pPoint, IppsECCPState* pECC);
```

Parameters

<code>pX</code>	Pointer to the x-coordinate of the point on the elliptic curve.
<code>pY</code>	Pointer to the y-coordinate of the point on the elliptic curve.
<code>pPoint</code>	Pointer to the context of the elliptic curve point.
<code>pECC</code>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function sets the coordinates of a point on the elliptic curve defined over a prime finite field $GF(p)$.

The context of the point on the elliptic curve must be already created by functions: [ECCPPointGetSize](#) and [ECCPPointInit](#). The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pX</code> , <code>pY</code> , <code>pPoint</code> , or <code>pECC</code> is not valid.

ECCPSetPointAtInfinity

Sets the point at infinity.

Syntax

```
IppStatus ippseccpSetPointAtInfinity(IppsECCPPoint* pPoint, IppsECCPState* pECC);
```

Parameters

<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function sets the point at infinity. The context of the elliptic curve point must be already created by functions: [ECCPPointGetSize](#) and [ECCPPointInit](#). The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPoint</i> or <i>pECC</i> is not valid.

ECCPGetPoint

Retrieves coordinates of the point on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippseCCPGetPoint(IppsBigNumState* pX, IppsBigNumState* pY, const IppsECCPPoint* pPoint, IppsECCPState* pECC);
```

Parameters

<i>pX</i>	Pointer to the x-coordinate of the point on the elliptic curve.
<i>pY</i>	Pointer to the y-coordinate of the point on the elliptic curve.
<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function retrieves the coordinates of the point on the elliptic curve defined over a prime finite field $GF(p)$ from the point context and allocates them in accordance with the set pointers *pX* and *pY*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .

`ippStsContextMatchErr` Indicates an error condition if one of the contexts pointed by `pX`, `pY`, `pPoint`, or `pECC` is not valid.

ECCPCheckPoint

Checks correctness of the point on the elliptic curve defined over $GF(p)$.

Syntax

```
IppStatus ippseCCPCheckPoint(const IppseCCPPoint* pP, IppECResult* pResult, IppseCCPState* pECC);
```

Parameters

`pP` Pointer to the elliptic curve point.
`pResult` Pointer to the result of the check.
`pECC` Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function checks the correctness of the point on the elliptic curve defined over a prime finite field $GF(p)$ and allocates the result of the check in accordance with the pointer `pResult`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

The result of the check for the correctness of the point can take one of the following values:

`ippECValid` Point is on the elliptic curve.
`ippECPointIsNotValid` Point is not on the elliptic curve and is not the point at infinity.
`ippECPointIsAtInfinite` Point is the point at infinity.

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 one of the contexts pointed by `pP` or `pECC` is not valid.

ECCPComparePoint

Compares two points on the elliptic curve defined over $GF(p)$.

Syntax

```
IppStatus ippseCCPComparePoint(const IppseCCPPoint* pP, const IppseCCPPoint* pQ, IppECResult* pResult, IppseCCPState* pECC);
```

Parameters

<i>pP</i>	Pointer to the elliptic curve point <i>P</i> .
<i>pQ</i>	Pointer to the elliptic curve point <i>Q</i> .
<i>pResult</i>	Pointer to the comparison result of two points: <i>P</i> and <i>Q</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function compares two points *P* and *Q* on the elliptic curve defined over a prime finite field $GF(p)$ and allocates the comparison result in accordance with the pointer *pResult*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

The comparison result of two points *P* and *Q* can take one of the following values:

<code>ippECPPointIsEqual</code>	Points <i>P</i> and <i>Q</i> are equal.
<code>ippECPPointIsNotEqual</code>	Points <i>P</i> and <i>Q</i> are different.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pP</i> or <i>pECC</i> is not valid.

ECCPNegativePoint

Finds an elliptic curve point which is an additive inverse for the given point over $GF(p)$.

Syntax

```
IppStatus ippseCCPNegativePoint(const IppsECCPPoint* pP, IppsECCPPoint* pR, IppsECCPState* pECC);
```

Parameters

<i>pP</i>	Pointer to the elliptic curve point <i>P</i> .
<i>pR</i>	Pointer to the elliptic curve point <i>R</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function finds an elliptic curve point *R* over a prime finite field $GF(p)$, which is an additive inverse of the given point *P*, that is, $R = -P$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pP</code> , <code>pR</code> , or <code>pECC</code> is not valid.

ECCPAddPoint

Computes the addition of two elliptic curve points over $GF(p)$.

Syntax

```
IppStatus ippseccpAddPoint(const IppsECCPPoint* pP, const IppsECCPPoint* pQ, IppsECCPPoint* pR, IppsECCPState* pECC);
```

Parameters

<code>pP</code>	Pointer to the elliptic curve point P .
<code>pQ</code>	Pointer to the elliptic curve point Q .
<code>pR</code>	Pointer to the elliptic curve point R .
<code>pECC</code>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function calculates the addition of two elliptic curve points P and Q over a finite field $GF(p)$ with the result in a point R such that $R = P + Q$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pP</code> , <code>pQ</code> , <code>pR</code> , or <code>pECC</code> is not valid.

ECCPMulPointScalar

Performs scalar multiplication of a point on the elliptic curve defined over $GF(p)$.

Syntax

```
IppStatus ippseccpMulPointScalar(const IppsECCPPoint* pP, const IppsBigNumState* pK, IppsECCPPoint* pR, IppsECCPState* pECC);
```

Parameters

<i>pP</i>	Pointer to the elliptic curve point <i>P</i> .
<i>pK</i>	Pointer to the scalar <i>K</i> .
<i>pR</i>	Pointer to the elliptic curve point <i>R</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function performs the *K* scalar multiplication of an elliptic curve point *P* over $GF(p)$ with the result in a point *R* such that $R = K \cdot P$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCPSet` or `ECCPSetStd`.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pP</i> , <i>pK</i> , <i>pR</i> , or <i>pECC</i> is not valid.

ECCPGenKeyPair

Generates a private key and computes public keys of the elliptic cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippseECCPGenKeyPair(IppsBigNumState* pPrivate, IppsECCPPointState* pPublic,
IppsECCPState* pECC, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file.

The function generates a private key *privKey* and computes a public key *pubKey* of the elliptic cryptosystem over a finite field $GF(p)$. The generation process employs the user specified *rndFunc* Random Generator.

The private key *privKey* is a number that lies in the range of $[1, n-1]$ where *n* is the order of the elliptic curve base point.

The public key *pubKey* is an elliptic curve point such that $\text{pubKey} = \text{privKey} \cdot G$, where G is the base point of the elliptic curve.

The memory size of the parameter *privKey* pointed by *pPrivate* must be less than that of the base point which can also be defined by the function [ECCPGetOrderBitSize](#).

The context of the point *pubKey* as an elliptic curve point must be created by using the functions [ECCPPointGetSize](#) and [ECCPPointInit](#).

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPrivate</i> , <i>pPublic</i> , or <i>pECC</i> is not valid.
<code>ippStsSizeErr</code>	Indicates an error condition if the memory size of the parameter <i>privKey</i> pointed by <i>pPrivate</i> is less than that of the order of the elliptic curve base point.

ECCPPublicKey

Computes a public key from the given private key of the elliptic cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippseCCPPublicKey(const IppsBigNumState* pPrivate, IppsECCPPoint* pPublic,
IppsECCPState* pECC);
```

Parameters

<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function computes the public key *pubKey* from the given private key *privKey* of the elliptic cryptosystem over a finite field $GF(p)$.

The private key *privKey* is a number that lies in the range of $[1, n-1]$ where n is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $\text{pubKey} = \text{privKey} \cdot G$, where G is the base point of the elliptic curve.

The context of the point *pubKey* as an elliptic curve point must be created by using the functions [ECCPPointGetSize](#) and [ECCPPointInit](#).

The elliptic curve domain parameters must be defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pPrivate</code> , <code>pPublic</code> , or <code>pECC</code> is not valid.
<code>ippStsInvalidPrivateKey</code>	Indicates an error condition if the value of the private key falls outside the range of $[1, n-1]$.

ECCPValidateKeyPair

Validates private and public keys of the elliptic cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippseCCPValidateKeyPair(const IppsBigNumState* pPrivate, const IppsECCPPoint* pPublic, IppECResult* pResult, IppsECCPState* pECC);
```

Parameters

<code>pPrivate</code>	Pointer to the private key <code>privKey</code> .
<code>pPublic</code>	Pointer to the public key <code>pubKey</code> .
<code>pResult</code>	Pointer to the validation result.
<code>pECC</code>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function validates the private key `privKey` and public key `pubKey` of the elliptic cryptosystem over a finite field $GF(p)$ and allocates the result of the validation in accordance with the pointer `pResult`.

The private key `privKey` is a number that lies in the range of $[1, n-1]$ where n is the order of the elliptic curve base point. The public key `pubKey` is an elliptic curve point such that $pubKey = privKey \cdot G$, where G is the base point of the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

The result of the cryptosystem keys validation for correctness can take one of the following values:

<code>ippECValid</code>	Keys are valid.
<code>ippECInvalidKeyPair</code>	Keys are not valid because $privKey \cdot G \neq pubKey$
<code>ippECInvalidPrivateKey</code>	Key <code>privKey</code> falls outside the range of $[1, n-1]$.
<code>ippECPointIsAtInfinite</code>	Key <code>pubKey</code> is the point at infinity.
<code>ippECInvalidPublicKey</code>	Key <code>pubKey</code> is not valid because $n \cdot pubKey \neq O$, where O is the point at infinity.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pPrivate</code> , <code>pPublic</code> , or <code>pECC</code> is not valid.

ECCPSetKeyPair

Sets private and/or public keys of the elliptic cryptosystem over $GF(p)$.

Syntax

```
IppStatus ippseECCPSetKeyPair(const IppsBigNumState* pPrivate, const IppsECCPPoint* pPublic,
IppBool regular, IppsECCPState* pECC);
```

Parameters

<code>pPrivate</code>	Pointer to the private key <code>privKey</code> .
<code>pPublic</code>	Pointer to the public key <code>pubKey</code> .
<code>regular</code>	Key status flag.
<code>pECC</code>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function sets a private key `privKey` and/or public key `pubKey` in the elliptic cryptosystem defined over a prime finite field $GF(p)$.

The private key `privKey` is a number that lies in the range of $[1, n-1]$ where n is the order of the elliptic curve base point. The public key `pubKey` is an elliptic curve point such that $pubKey = privKey \cdot G$, where G is the base point of the elliptic curve.

The two possible values of the parameter `regular` define the key timeliness status:

<code>ippTrue</code>	Keys are regular.
<code>ippFalse</code>	Keys are ephemeral.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pPrivate</code> , <code>pPublic</code> , or <code>pECC</code> is not valid.

ECCPSharedSecretDH

Computes a shared secret field element by using the Diffie-Hellman scheme.

Syntax

```
IppStatus ippsECCPSharedSecretDH(const IppsBigNumState* pPrivate, const IppsECCPointState* pPublic, IppsBigNumState* pShare, IppsECCPState* pECC);
```

Parameters

<i>pPrivate</i>	Pointer to your own public key <i>pubKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>pShare</i>	Pointer to the secret number <i>bnShare</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function computes a secret number *bnShare*, which is a secret key shared between two participants of the cryptosystem.

In cryptography, metasyntactic names such as Alice as Bob are normally used as examples and in discussions and stand for participant A and participant B.

Both participants (Alice and Bob) use the cryptosystem for receiving a common secret point on the elliptic curve called a secret key. To receive a secret key, participants apply the Diffie-Hellman key-agreement scheme involving public key exchange. The value of the secret key entirely depends on participants.

According to the scheme, Alice and Bob perform the following operations:

1. Alice calculates her own public key *pubKeyA* by using her private key *privKeyA*: $pubKeyA = privKeyA \cdot G$, where *G* is the base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key *pubKeyB* by using his private key *privKeyB*: $pubKeyB = privKeyB \cdot G$, where *G* is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point *shareA*. When calculating, she uses her own private key and Bob's public key and applies the following formula: $shareA = privKeyA \cdot pubKeyB = privKeyA \cdot privKeyB \cdot G$.
4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $shareB = privKeyB \cdot pubKeyA = privKeyB \cdot privKeyA \cdot G$.

Because the following equation is true $privKeyA \cdot privKeyB \cdot G = privKeyB \cdot privKeyA \cdot G$, the result of both calculations is the same, that is, the equation $shareA = shareB$ is true. The secret point serves as a secret key.

Shared secret *bnShare* is an x-coordinate of the secret point on the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pPublic</code> , <code>pPShare</code> , or <code>pECC</code> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if the memory size of <code>bnShare</code> pointed by <code>pShare</code> is less than the value of <code>feBitSize</code> in the function EC-CPInit .
<code>ippStsShareKeyErr</code>	Indicates an error condition if the shared secret key is not valid. (For example, the shared secret key is invalid if the result of the secret point calculation is the point at infinity.)

ECCPSharedSecretDHC

Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.

Syntax

```
IppStatus ippseccpSharedSecretDHC(const IppsBigNumState* pPrivate, const IppsECCPPointState* pPublic, IppsBigNumState* pShare, IppsECCPState* pECC);
```

Parameters

<code>pPrivate</code>	Pointer to your own public key <code>pubKey</code> .
<code>pPublic</code>	Pointer to the public key <code>pubKey</code> .
<code>pShare</code>	Pointer to the secret number <code>bnShare</code> .
<code>pECC</code>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function computes a secret number `bnShare` which is a secret key shared between two participants of the cryptosystem. Both participants (Alice and Bob) use the cryptosystem for getting a common secret point on the elliptic curve by using the Diffie-Hellman scheme and elliptic curve cofactor h .

Alice and Bob perform the following operations:

1. Alice calculates her own public key `pubKeyA` by using her private key `privKeyA`: $\text{pubKeyA} = \text{privKeyA} \cdot G$, where G is the base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key `pubKeyB` by using his private key `privKeyB`: $\text{pubKeyB} = \text{privKeyB} \cdot G$, where G is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point `shareA`. When calculating, she uses her own private key and Bob's public key and applies the following formula: $\text{shareA} = h \cdot \text{privKeyA} \cdot \text{pubKeyB} = h \cdot \text{privKeyA} \cdot \text{privKeyB} \cdot G$, where h is the elliptic curve cofactor.

4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $shareB = h \cdot privKeyB \cdot pubKeyA = h \cdot privKeyB \cdot privKeyA \cdot G$, where *h* is the elliptic curve cofactor.

Shared secret *bnShare* is an x-coordinate of the secret point on the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPublic</i> , <i>pPShare</i> , or <i>pECC</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if the memory size of <i>bnShare</i> pointed by <i>pShare</i> is less than the value of <i>feBitSize</i> in the function ECCPInit .
<code>ippStsShareKeyErr</code>	Indicates an error condition if the shared secret key is not valid. (For example, the shared secret key is invalid if the result of the secret point calculation is the point at infinity.)

ECCPSignDSA

Computes a digital signature over a message digest.

Syntax

```
IppStatus ippseccpsigndsa(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pPrivate, IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCPState* pECC);
```

Parameters

<i>pMsgDigest</i>	Pointer to the message digest <i>msg</i> to be digitally signed, that is, to be encrypted with a private key.
<i>pPrivate</i>	Pointer to the signer's regular private key.
<i>pSignX</i>	Pointer to the integer <i>r</i> of the digital signature.
<i>pSignY</i>	Pointer to the integer <i>s</i> of the digital signature.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

A message digest is a fixed size number derived from the original message with an applied hash function over the binary code of the message. The signer's private key and the message digest are used to create a signature.

A digital signature over a message consists of a pair of large numbers *r* and *s* which the given function computes.

The scheme used for computing a digital signature is analogue of the ECDSA scheme, an elliptic curve analogue of the DSA scheme. ECDSA assumes that the following keys are hitherto set by a message signer:

ephPrivKey Ephemeral private key.
ephPubKey Ephemeral public key.

The keys can be generated and set up by the unctons [ECCPGenKeyPair](#) and [ECCPSetKeyPair](#) with only requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

For more information on digital signatures, please refer to the [\[ANSI\]](#) standard.

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 <code>NULL</code> .
<i>ippStsContextMatchErr</i>	Indicates an error condition if one of the contexts pointed by <i>pMsgDigest</i> , <i>pSignX</i> , <i>pSignY</i> , or <i>ECC</i> is not valid.
<i>ippStsMessageErr</i>	Indicates an error condition if the value of <i>msg</i> pointed by <i>pMsgDigest</i> falls outside the range of $[1, n-1]$ where n is the order of the elliptic curve base point G .
<i>ippStsRangeErr</i>	Indicates an error condition if one of the parameters pointed by <i>pSignX</i> or <i>pSignY</i> has a less memory size than the order n of the elliptic curve base point G .
<i>ippStsEphemeralKeyErr</i>	Indicates an error condition if the values of the ephemeral keys <i>ephPrivKey</i> and <i>ephPubKey</i> are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

See Also

- [Code Example](#)

ECCPVerifyDSA

Verifies authenticity of the digital signature over a message digest (ECDSA).

Syntax

```
IppStatus ippseccpVerifyDSA(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pSignX, const IppsBigNumState* pSignY, IppeCResult* pResult, IppsECCPState* pECC);
```

Parameters

<i>pMsgDigest</i>	Pointer to the message digest <i>msg</i> .
<i>pSignX</i>	Pointer to the integer r of the digital signature.
<i>pSignY</i>	Pointer to the integer s of the digital signature.
<i>pResult</i>	Pointer to the digital signature verification result.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function verifies authenticity of the digital signature over a message digest *msg*. The signature consists of two large integers: *r* and *s*.

The scheme used to verify the signature is an elliptic curve analogue of the DSA scheme and assumes that the following cryptosystem key be hitherto set:

regPubKey Message sender's regular public key.

The *regPubKey* is set by the function [ECCPSetKeyPair](#).

The result of the digital signature verification can take one of two possible values:

`ippECValid` Digital signature is valid.

`ippECInvalidSignature` Digital signature is not valid.

The call to the `ECCPVerifyDSA` function must be preceded by the call to the [ECCPSignDSA](#) function which computes the digital signature over the message digest *msg* and represents the signature with two numbers: *r* and *s*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

For more information on digital signatures, please refer to the [\[ANSI\]](#) standard.

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 one of the contexts pointed by *pMsgDigest*, *pSignX*, *pSignY*, or *ECC* is not valid.

`ippStsMessageErr` Indicates an error condition if the value of *msg* pointed by *pMsgDigest* falls outside the range of $[1, n-1]$ where *n* is the order of the elliptic curve base point *G*.

See Also

- [Code Example](#)

ECCPSignNR

Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippseccpsignnr(const IppsBigNumState* pMsgDigest, const IppsBigNumState* pPrivate,
IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCPState* pECC);
```

Parameters

pMsgDigest Pointer to the message digest *msg*.

<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pSignX</i>	Pointer to the integer <i>r</i> of the digital signature.
<i>pSignY</i>	Pointer to the integer <i>s</i> of the digital signature.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function computes two large numbers *r* and *s* which form the digital signature over a message digest *msg*.

The scheme used to compute the digital signature is an elliptic curve analogue of the El-Gamal Digital Signature scheme with the message recovery (the Nyberg-Rueppel signature scheme). The scheme that the given function uses assumes that the following cryptosystem keys are hitherto set up by the message sender:

<i>regPrivKey</i>	Regular private key.
<i>ephPrivKey</i>	Ephemeral private key.
<i>ephPubKey</i>	Ephemeral public key.

The keys can be generated and set up by the functions [ECCPGenKeyPair](#) and [ECCPSetKeyPair](#) with only requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

For more information on digital signatures, please refer to the [\[ANSI\]](#) standard.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pMsgDigest</i> , <i>pSignX</i> , <i>pSignY</i> , or <i>ECC</i> is not valid.
<code>ippStsMessageErr</code>	Indicates an error condition if the value of <i>msg</i> pointed by <i>pMsgDigest</i> falls outside the range of $[1, n-1]$ where <i>n</i> is the order of the elliptic curve base point <i>G</i> .
<code>ippStsRangeErr</code>	Indicates an error condition if one of the parameters pointed by <i>pSignX</i> or <i>pSignY</i> has a less memory size than the order <i>n</i> of the elliptic curve base point <i>G</i> .
<code>ippStsEphemeralKeyErr</code>	Indicates an error condition if the values of the ephemeral keys <i>ephPrivKey</i> and <i>ephPubKey</i> are not valid. (Either <i>r</i> = 0 or <i>s</i> = 0 is received as a result of the digital signature calculation).

ECCPVerifyNR

Verifies authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippseCCPVerifyNR(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pSignX, const IppsBigNumState* pSignY, IppeCResult* pResult, IppsECCPState* pECC);
```

Parameters

<i>pMsgDigest</i>	Pointer to the message digest <i>msg</i> .
<i>pSignX</i>	Pointer to the integer <i>r</i> of the digital signature.
<i>pSignY</i>	Pointer to the integer <i>s</i> of the digital signature.
<i>pResult</i>	Pointer to the digital signature verification result.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function verifies authenticity of the digital signature over a message digest *msg*. The signature is presented with two large integers *r* and *s*.

The scheme used to compute the digital signature is an elliptic curve analogue of the El-Gamal Digital Signature scheme with the message recovery (the Nyberg-Rueppel signature scheme). The scheme that the given function uses assumes that the following cryptosystem keys be hitherto set up by the message sender:

regPubKey Message sender's regular private key.

The key can be generated and set up by the function [ECCPGenKeyPair](#).

The result of the digital signature verification can take one of two possible values:

<code>ippECValid</code>	The digital signature is valid.
<code>ippECInvalidSignature</code>	The digital signature is not valid.

The call to the `ECCPVerifyNR` function must be preceded by the call to the [ECCPSignNR](#) function which computes the digital signature over the message digest *msg* and represents the signature with two numbers: *r* and *s*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCPSet](#) or [ECCPSetStd](#).

For more information on digital signatures, please refer to the [\[ANSI\]](#) standard.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pMsgDigest</i> , <i>pSignX</i> , <i>pSignY</i> , or <i>ECC</i> is not valid.
<code>ippStsMessageErr</code>	Indicates an error condition if the value of <i>msg</i> pointed by <i>pMsgDigest</i> falls outside the range of $[1, n-1]$ where <i>n</i> is the order of the elliptic curve base point <i>G</i> .

Signing/Verification Using the Elliptic Curve Cryptography Functions over a Prime Finite Field

Use of ECCPSignDSA, ECCPVerifyDSA

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

#include "ippcp.h"

static IppsECCPState* newStd_256_ECP(void)
{
    int ctxSize;
    ippseccpGetSize(256, &ctxSize);
    IppsECCPState* pCtx = (IppsECCPState*)( new Ipp8u [ctxSize] );
    ippseccpInit(256, pCtx);
    ippseccpSetStd(IppsECCPStd256r1, pCtx);
    return pCtx;
}

static IppsECCPPointState* newECP_256_Point(void)
{
    int ctxSize;
    ippseccpPointGetSize(256, &ctxSize);
    IppsECCPPointState* pPoint = (IppsECCPPointState*)( new Ipp8u [ctxSize] );
    ippseccpPointInit(256, pPoint);
    return pPoint;
}

static IppsBigNumState* newBN(int len, const Ipp32u* pData)
{
    int ctxSize;
    ippsbignumGetSize(len, &ctxSize);
    IppsBigNumState* pBN = (IppsBigNumState*)( new Ipp8u [ctxSize] );
```

```

    ippsBigNumInit(len, pBN);
    if(pData)
        ippsSet_BN(IppsBigNumPOS, len, pData, pBN);
    return pBN;
}

IppsPRNGState* newPRNG(void)
{
    int ctxSize;
    ippsPRNGGetSize(&ctxSize);
    IppsPRNGState* pCtx = (IppsPRNGState*)( new Ipp8u [ctxSize] );
    ippsPRNGInit(160, pCtx);
    return pCtx;
}

int main(void)
{
    // define standard 256-bit EC
    IppsECCPState* pECP = newStd_256_ECP();

    // extract or use any other way to get order(EC)
    const Ipp32u secp256r1_r[] = {0xFC632551, 0xF3B9CAC2, 0xA7179E84, 0xBCE6FAAD
        0xFFFFFFFF, 0xFFFFFFFF, 0x00000000, 0xFFFFFFFF};
    const int ordSize = sizeof(secp256r1_r)/sizeof(Ipp32u);
    IppsBigNumState* pECPorder = newBN(ordSize, secp256r1_r);

    // define a message to be signed; let it be random, for example
    IppsPRNGState* pRandGen = newPRNG(); // 'external' PRNG

    Ipp32u tmpData[ordSize];
    ippsPRNGGen(tmpData, 256, pRandGen);
    IppsBigNumState* pRandMsg = newBN(ordSize, tmpData); // random 256-bit message
    IppsBigNumState* pMsg = newBN(ordSize, 0);           // msg to be signed
    ippsMod_BN(pRandMsg, pECPorder, pMsg);

```

```
// declare Signer's regular and ephemeral key pair
IppsBigNumState* regPrivate = newBN(ordSize, 0);
IppsBigNumState* ephPrivate = newBN(ordSize, 0);
// define Signer's ephemeral key pair
IppsECCPointState* regPublic = newECP_256_Point();
IppsECCPointState* ephPublic = newECP_256_Point();

// generate regular & ephemeral key pairs, should be different each other
ippsECCGenKeyPair(regPrivate, regPublic, pECP, ippsPRNGen, pRandGen);
ippsECCGenKeyPair(ephPrivate, ephPublic, pECP, ippsPRNGen, pRandGen);

//
// signature
//

// set ephemeral key pair
ippsECCSetKeyPair(ephPrivate, ephPublic, ippFalse, pECP);
// compute signature
IppsBigNumState* signX = newBN(ordSize, 0);
IppsBigNumState* signY = newBN(ordSize, 0);
ippsECCPSignDSA(pMsg, regPrivate, signX, signY, pECP);

//
// verification
//
ippsECCSetKeyPair(NULL, regPublic, ippTrue, pECP);
IppECResult eccResult;
ippsECCPVerifyDSA(pMsg, signX, signY, &eccResult, pECP);
if(ippECValid == eccResult)
    cout << "signature verificatioin passed" <<endl;
else
    cout << "signature verificatioin failed" <<endl;
```

```

delete [] (Ipp8u*)signX;
delete [] (Ipp8u*)signY;
delete [] (Ipp8u*)ephPublic;
delete [] (Ipp8u*)regPublic;
delete [] (Ipp8u*)ephPrivate;
delete [] (Ipp8u*)regPrivate;
delete [] (Ipp8u*)pRandMsg;
delete [] (Ipp8u*)pMsg;
delete [] (Ipp8u*)pRandGen;
delete [] (Ipp8u*)pECPorder;
delete [] (Ipp8u*)pECP;
return 0;
}

```

Functions Based on GF(2^m)

This section describes functions designed to specify the elliptic curve cryptosystem and perform various operations on the elliptic curve defined over a binary finite field. The examples of the operations are illustrated below:

1. Setting up operations [ECCBSet](#) sets up elliptic curve domain parameters. [ECCBSetKeyPair](#) sets a pair of public and private keys for the given cryptosystem.
2. Computation operations [ECCBAddPoint](#) adds two points on the elliptic curve. [ECCBMulPointScalar](#) performs the scalar multiplication of a point on the elliptic curve. [ECCBSignDSA](#) computes the digital signature of a message.
3. Validation operations [ECCBValidate](#) checks validity of the elliptic curve domain parameters. [ECCBValidateKeyPair](#) validates correctness of the public and private keys.
4. Generation operations [ECCBGenKeyPair](#) generates a private key and computes a public key for the given elliptic cryptosystem.
5. Retrieval operations [ECCBGet](#) retrieves elliptic curve domain parameters. [ECCBGetOrderBitSize](#) retrieves the size of a base point in bytes.

All functions described in this section employ a context `IppsECCBState` that catches several auxiliary components specifying operations performed on the elliptic curve or entire elliptic cryptosystem. ECCB stands for Elliptic Curve Cryptography Binary and means that all functions whose name include this abbreviation perform operations over a binary finite field GF(2^m).

ECCBGetSize

Gets the size of the `IppsECCBState` context.

Syntax

```
IppStatus ippsECCBGetSize(int feBitSize, int *pSize);
```

Parameters

<i>feBitSize</i>	Size (in bits) of the field element.
<i>pSize</i>	Pointer to the size of the context (in bytes).

Description

This function is declared in the `ippcp.h` file.

The function computes the size of the context in bytes for the elliptic cryptosystem over a binary finite field $GF(2^m)$.

Context is a structure `IppsECCBState` designed to store information about the cryptosystem status.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if the value of the parameter <i>feBitSize</i> is less than 1.

ECCBInit

Initializes context for the elliptic curve cryptosystem over $GF(2^m)$.

Syntax

```
IppStatus ippseCCBInit(int feBitSize, IppsECCBState* pECC);
```

Parameters

<i>feBitSize</i>	Size (in bits) of a field element.
<i>pECC</i>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file.

The function initializes the context of the elliptic curve cryptosystem over a binary finite field $GF(2^m)$.

Context is a structure `IppsECCBState` designed to store information about the cryptosystem status.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if the value of the parameter <i>feBitSize</i> is less than 1.

ECCBSet

Sets up elliptic curve domain parameters over $GF(2^m)$.

Syntax

```
IppStatus ippseccbset(const IppsBigNumState* pPrime, const IppsBigNumState* pA, const
IppsBigNumState* pB, const IppsBigNumState* pGX, const IppsBigNumState* pGY, const
IppsBigNumState* pOrder, int cofactor, IppsECCBState* pECC);
```

Parameters

<i>pPrime</i>	Pointer to the irreducible binary polynomial $f(x)$ of degree m which specifies the presentation of the field $GF(2^m)$.
<i>pA</i>	Pointer to the coefficient A of the equation defining the elliptic curve.
<i>pB</i>	Pointer to the coefficient B of the equation defining the elliptic curve.
<i>pGX</i>	Pointer to the x -coordinate of the elliptic curve base point.
<i>pGY</i>	Pointer to the y -coordinate of the elliptic curve base point.
<i>pOrder</i>	Pointer to the order of the elliptic curve base point.
<i>cofactor</i>	Cofactor.
<i>pECC</i>	Pointer to the context of the cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function sets up the elliptic curve domain parameters over a binary finite field $GF(2^m)$. These are as follows:

- *pPrime* sets up the characteristic $f(x)$ of degree m , which specifies the presentation of the field $GF(2^m)$.
- *pA*, *pB* set up the coefficients A and B of the equation defining the elliptic curve:

$$y^2 + x \cdot y = x^3 + A \cdot x^2 + B \text{ in } GF(2^m)$$
- *pGX*, *pGY* are pointers to the affine coordinates of the elliptic curve base point G .
- *pOrder* is a pointer to the order n of the elliptic curve base point G such that

$$n \cdot G = O, \text{ where } O \text{ is the point at infinity and } n \text{ is a prime number.}$$
- *cofactor* sets up the ratio h of a general number of points $\#E$ on the elliptic curve (including the point at infinity) to the order n of the base point:

$$h = \#E/n.$$

The domain parameters are set in the cryptosystem context which must be already created by the [ECCBGetSize](#) and [ECCBInit](#) functions.

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 <code>NULL</code> .

<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPrime</i> , <i>pA</i> , <i>pB</i> , <i>pGX</i> , <i>pGY</i> , <i>pOrder</i> , and <i>pECC</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if the memory size of one of the parameters pointed by <i>pPrime</i> , <i>pA</i> , <i>pB</i> , <i>pGX</i> , <i>pGY</i> , <i>pOrder</i> , and <i>pECC</i> is more than the value of <i>feBitSize</i> in the ECCBInit function or the value of <i>cofactor</i> is less than or equal to zero.

ECCBSetStd

Sets up a recommended set of elliptic curve domain parameters over $GF(2^m)$.

Syntax

```
IppStatus ippseCCBSetStd(IppECCType flag, IppsECCBState* pECC);
```

Parameters

<i>flag</i>	Set specifier.
<i>pECC</i>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file.

The function sets a recommended set of elliptic curve domain parameters over a binary finite field $GF(2^m)$.

The set is defined by the value of the parameter *flag*. Possible values of the parameter are as follows:

<code>IppECCBStd113r1</code>	For the cryptosystem context where <i>feBitSize</i> ==113
<code>IppECCBStd113r2</code>	For the cryptosystem context where <i>feBitSize</i> ==113
<code>IppECCBStd131r1</code>	For the cryptosystem context where <i>feBitSize</i> ==131
<code>IppECCBStd131r2</code>	For the cryptosystem context where <i>feBitSize</i> ==131
<code>IppECCBStd163k1</code>	For the cryptosystem context where <i>feBitSize</i> ==163
<code>IppECCBStd163r1</code>	For the cryptosystem context where <i>feBitSize</i> ==163
<code>IppECCBStd163r2</code>	For the cryptosystem context where <i>feBitSize</i> ==163
<code>IppECCBStd193r1</code>	For the cryptosystem context where <i>feBitSize</i> ==193
<code>IppECCBStd193r2</code>	For the cryptosystem context where <i>feBitSize</i> ==193
<code>IppECCBStd233k1</code>	For the cryptosystem context where <i>feBitSize</i> ==233
<code>IppECCBStd233r1</code>	For the cryptosystem context where <i>feBitSize</i> ==233
<code>IppECCBStd239k1</code>	For the cryptosystem context where <i>feBitSize</i> ==239
<code>IppECCBStd283k1</code>	For the cryptosystem context where <i>feBitSize</i> ==283
<code>IppECCBStd283r1</code>	For the cryptosystem context where <i>feBitSize</i> ==283
<code>IppECCBStd409k1</code>	For the cryptosystem context where <i>feBitSize</i> ==409
<code>IppECCBStd409r1</code>	For the cryptosystem context where <i>feBitSize</i> ==409
<code>IppECCBStd571k1</code>	For the cryptosystem context where <i>feBitSize</i> ==571

IppECCBStd571r1

For the cryptosystem context where *feBitSize*==571.

For more information on parameter values for the recommended elliptic curves, see [SEC2] .

The cryptosystem context must be already created by the [ECCBGetSize](#) and [ECCBInit](#) functions. The value of *feBitSize* is applied when these function are called and predetermines the possible choice of the *flag* value.

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 <code>NULL</code> .
<i>ippStsContextMatchErr</i>	Indicates an error condition if the cryptosystem context is not valid.
<i>ippStsECCInvalidFlagErr</i>	Indicates an error condition if the value of the parameter <i>flag</i> is not valid.

ECCBGet

Retrieves elliptic curve domain parameters over $GF(2^m)$.

Syntax

```
IppStatus ippseECCBGet(IppsBigNumState* pPrime, IppsBigNumState* pA, IppsBigNumState* pB,
IppsBigNumState* pGX, IppsBigNumState* pGY, IppsBigNumState* pOrder, int* cofactor,
IppsECCBState* pECC);
```

Parameters

<i>pPrime</i>	Pointer to the irreducible binary polynomial $f(x)$ of degree m which specifies the presentation of the field $GF(2^m)$.
<i>pA</i>	Pointer to the coefficient A of the equation defining the elliptic curve.
<i>pB</i>	Pointer to the coefficient B of the equation defining the elliptic curve.
<i>pGX</i>	Pointer to the x -coordinate of the elliptic curve base point.
<i>pGY</i>	Pointer to the y -coordinate of the elliptic curve base point.
<i>pOrder</i>	Pointer to the order n of the elliptic curve base point.
<i>cofactor</i>	Pointer to the cofactor h .
<i>pECC</i>	Pointer to the context of the cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function retrieves elliptic curve domain parameters from the context of the elliptic cryptosystem over a binary finite field $GF(2^m)$ and allocates them in accordance with the pointers *pPrime*, *pA*, *pB*, *pGX*, *pGY*, *pOrder*, and *cofactor*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pPrime</code> , <code>pA</code> , <code>pB</code> , <code>pGX</code> , <code>pGY</code> , <code>pOrder</code> , or <code>pECC</code> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if the memory size of one of the parameters pointed by <code>pPrime</code> , <code>pA</code> , <code>pB</code> , <code>pGX</code> , <code>pGY</code> , <code>pOrder</code> , and <code>pECC</code> is less than the value of <code>feBitSize</code> in the <code>ECCBInit</code> function.

ECCBGetOrderBitSize

Retrieves order size of the elliptic curve base point over $GF(2^m)$ in bits.

Syntax

```
IppStatus ippseccBGetOrderBitSize(int* pBitSize, IppsECCBState* pECC);
```

Parameters

<code>pBitSize</code>	Pointer to the size of the base point (in bits).
<code>pECC</code>	Pointer to the cryptosystem context.

Description

This function is declared in the `ippcp.h` file.

The function retrieves the order size (in bits) of the elliptic curve base point G from the context of elliptic cryptosystem over a binary finite field $GF(2^m)$ and allocates it in accordance with the pointer `pBitsSize`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCBSet` or `ECCBSetStd`.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the cryptosystem context is not valid.

ECCBValidate

Checks validity of the elliptic curve domain parameters over $GF(2^m)$.

Syntax

```
IppStatus ippseccBValidate(int nTrials, IppECCResult* pResult, IppsECCBState* pECC,
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>nTrials</i>	A number of attempts made to check the number for primality.
<i>pResult</i>	Pointer to the result received upon the check of the elliptic curve domain parameters.
<i>pECC</i>	Pointer to the cryptosystem context.
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file.

The function checks validity of the elliptic curve domain parameters over a binary finite field $GF(2^m)$ and stores the result of the check in accordance with the pointer *pResult*.

Elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#). The purpose of the parameters *rndFunc*, *pRndParam*, and *nTrials* is analogous to that of the parameters *rndFunc*, *pRndParam*, and *nTrials* in the [PrimeTest](#) function.

The result of the elliptic curve domain parameters check can take one of the following values:

<code>ippECValid</code>	The parameters are valid.
<code>ippECComplicatedBase</code>	The irreducible binary polynomial $f(x)$ of degree m which specifies the presentation of the field $GF(2^m)$ is not valid because the set of polynomials consists of more than five elements.
<code>ippECCompositeBase</code>	The binary polynomial $f(x)$ is not irreducible.
<code>ippECIsSupersingular</code>	The coefficient in the elliptic curve equation is <code>NULL</code> .
<code>ippECPointAtInfinite</code>	The elliptic curve base point G is the point at infinity.
<code>ippECPointIsNotValid</code>	Base point G is not on the elliptic curve.
<code>ippECCompositeOrder</code>	The order n of the base point G is a composite number.
<code>ippECInvalidOrder</code>	The order n of the base point G is not valid because the requirement that $n \cdot G = O$, where O is the point at infinity is not met.
<code>ippECIsWeakSSSA</code>	$h \cdot n = 2^m$ where h is a cofactor and n is the order n of the base point.
<code>ippECIsWeakMOV</code>	The curve is excluded because it is subject to the MOV reduction attack.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>c</i> or <i>pECC</i> is not valid.

`ippStsBadArgErr` Indicates an error condition if the memory size of the parameter *seed* is less than five words (32 bytes in each) or the value of the parameter *nTrails* is less than 1.

ECCBPointGetSize

Gets the size of the `IppsECCBPoint` context in bytes for a point on the elliptic curve point defined over $GF(2^m)$.

Syntax

```
IppStatus ippseccbpointgetsize(int feBitSize, int* pSize);
```

Parameters

feBitSize Size (in bits) of the field element.
pSize Pointer to the context size.

Description

This function is declared in the `ippcp.h` file.

The function computes the context size in bytes for a point on the elliptic curve defined over a binary finite field $GF(2^m)$.

Context is a structure `IppsECCBPoint` intended for storing the information about a point on the elliptic curve defined over $GF(2^m)$.

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`.
`ippStsSizeErr` Indicates an error condition if the value of the parameter *feBitSize* is less than 1.

ECCBPointInit

Initializes the context for a point on the elliptic curve defined over $GF(2^m)$.

Syntax

```
IppStatus ippseccbpointinit(int feBitSize, IppsECCBPoint* pPoint);
```

Parameters

feBitSize Size (in bits) of the field element.
pECC Pointer to the context of the elliptic curve point.

Description

This function is declared in the `ippcp.h` file.

The function initializes the context for a point on the elliptic curve defined over a binary finite field $GF(2^m)$.

Context is a structure `IppsECCBPoint` intended for storing the information about a point on the elliptic curve defined over $GF(2^m)$.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if the value of the parameter <i>feBitSize</i> is less than 1.

ECCBSetPoint

Sets coordinates of a point on the elliptic curve defined over $GF(2^m)$.

Syntax

```
IppStatus ippseECCBSetPoint(const IppsBigNumState* pX, const IppsBigNumState* pY,
IppsECCBPoint* pPoint, IppsECCBState* pECC);
```

Parameters

<i>pX</i>	Pointer to the x-coordinate of the point on the elliptic curve.
<i>pY</i>	Pointer to the y-coordinate of the point on the elliptic curve.
<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function sets the coordinates of a point on the elliptic curve defined over a binary finite field $GF(2^m)$.

The context of the point on the elliptic curve must be already created by functions: [ECCBPointGetSize](#) and [ECCBPointInit](#). The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pX</i> , <i>pY</i> , <i>pPoint</i> , or <i>pECC</i> is not valid.

ECCBSetPointAtInfinity

Sets the point at infinity.

Syntax

```
IppStatus ippseCCBSetPointAtInfinity(IppseCCBPoint* pPoint, IppseCCBState* pECC);
```

Parameters

<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function sets the point at infinity. The context of the elliptic curve point must be already created by functions: [ECCBPointGetSize](#) and [ECCBPointInit](#). The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPoint</i> or <i>pECC</i> is not valid.

ECCBGetPoint

Retrieves coordinates of the point on the elliptic curve defined over $GF(2^m)$.

Syntax

```
IppStatus ippseCCBGetPoint(IppseBigNumState* pX, IppseBigNumState* pY, const IppseCCBPoint* pPoint, IppseCCBState* pECC);
```

Parameters

<i>pX</i>	Pointer to the x-coordinate of the point on the elliptic curve.
<i>pY</i>	Pointer to the y-coordinate of the point on the elliptic curve.
<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function retrieves the coordinates of the point on the elliptic curve defined over a binary finite field $GF(2^m)$ from the point context and allocates them in accordance with the set pointers *pX* and *pY*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pX</code> , <code>pY</code> , <code>pPoint</code> , or <code>pECC</code> is not valid.

ECCBCheckPoint

Checks correctness of the point on the elliptic curve defined over $GF(2^m)$.

Syntax

```
IppStatus ippseCCBCheckPoint(const IppsECCBPoint* pP, IppeECResult* pResult, IppsECCBState* pECC);
```

Parameters

<code>pP</code>	Pointer to the elliptic curve point.
<code>pResult</code>	Pointer to the result of the check.
<code>pECC</code>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function checks the correctness of the point on the elliptic curve defined over a binary finite field $GF(2^m)$ and allocates the result of the check in accordance with the pointer `pResult`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

The result of the check for the correctness of the point can take one of the following values:

<code>ippeECValid</code>	Point is on the elliptic curve.
<code>ippeECPointIsNotValid</code>	Point is not on the elliptic curve and is not the point at infinity.
<code>ippeECPointIsAtInfinite</code>	Point is the point at infinity.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pP</code> or <code>pECC</code> is not valid.

ECCBComparePoint

Compares two points on the elliptic curve defined over $GF(2^m)$.

Syntax

```
IppStatus ippseCCBComparePoint(const IppseCCBPoint* pP, const IppseCCBPoint* pQ,
IppECResult* pResult, IppseCCBState* pECC);
```

Parameters

<i>pP</i>	Pointer to the elliptic curve point <i>P</i> .
<i>pQ</i>	Pointer to the elliptic curve point <i>Q</i> .
<i>pResult</i>	Pointer to the comparison result of two points: <i>P</i> and <i>Q</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function compares two points *P* and *Q* on the elliptic curve defined over a binary finite field $GF(2^m)$ and allocates the comparison result in accordance with the pointer *pResult*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCBSet` or `ECCBSetStd`.

The comparison result of two points *P* and *Q* can take one of the following values:

<code>ippECPPointIsEqual</code>	Points <i>P</i> and <i>Q</i> are equal.
<code>ippECPPointIsNotEqual</code>	Points <i>P</i> and <i>Q</i> are different.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pP</i> or <i>pECC</i> is not valid.

ECCBNegativePoint

Finds the elliptic curve point which is an additive inverse for the given point over $GF(2^m)$.

Syntax

```
IppStatus ippseCCBNegativePoint(const IppseCCBPoint* pP, IppseCCBPoint* pR, IppseCCBState*
pECC);
```

Parameters

<i>pP</i>	Pointer to the elliptic curve point <i>P</i> .
-----------	--

<i>pR</i>	Pointer to the elliptic curve point <i>R</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function finds an elliptic curve point *R* over a binary finite field $GF(2^m)$ which is an additive inverse of the given point *P*, i.e., $R = -P$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pP</i> , <i>pR</i> , or <i>pECC</i> is not valid.

ECCBAddPoint

Computes the addition of two elliptic curve points over $GF(2^m)$.

Syntax

```
IppStatus ippseCCBAddPoint(const IppsECCBPoint* pP, const IppsECCBPoint* pQ, IppsECCBPoint* pR, IppsECCBState* pECC);
```

Parameters

<i>pP</i>	Pointer to the elliptic curve point <i>P</i> .
<i>pQ</i>	Pointer to the elliptic curve point <i>Q</i> .
<i>pR</i>	Pointer to the elliptic curve point <i>R</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function calculates the addition of two elliptic curve points *P* and *Q* over a binary finite field $GF(2^m)$ with the result in a point *R* such that $R=P+Q$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pP</i> , <i>pQ</i> , <i>pR</i> , or <i>pECC</i> is not valid.

ECCBMulPointScalar

Performs scalar multiplication of a point on the elliptic curve defined over $GF(2^m)$.

Syntax

```
IppStatus ippsECCBMulPointScalar(const IppsECCBPoint* pP, const IppsBigNumState* pK,
IppsECCBPoint* pR, IppsECCBState* pECC);
```

Parameters

<i>pP</i>	Pointer to the elliptic curve point <i>P</i> .
<i>pK</i>	Pointer to the scalar <i>K</i> .
<i>pR</i>	Pointer to the elliptic curve point <i>R</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function performs the *K* scalar multiplication of an elliptic curve point *P* over $GF(2^m)$ with the result in a point *R* such that $R = K \cdot P$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCPSet` or `ECCPSetStd`.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pP</i> , <i>pK</i> , <i>pR</i> , or <i>pECC</i> is not valid.

ECCBGenKeyPair

Generates a private key and computes public keys of the elliptic cryptosystem over $GF(2^m)$.

Syntax

```
IppStatus ippsECCBGenKeyPair(IppsBigNumState* pPrivate, IppsECCBPointState* pPublic,
IppsECCBState* pECC, IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.
<i>rndFunc</i>	Specified Random Generator.
<i>pRndParam</i>	Pointer to the Random Generator context.

Description

This function is declared in the `ippcp.h` file.

The function generates a private key *privKey* and computes a public key *pubKey* of the elliptic cryptosystem over a binary finite field $GF(2^m)$. The generation process employs user specified *rndFunc* Random Generator.

The private key *privKey* is a number that lies in the range of $[1, n-1]$ where n is the order of the elliptic curve base point.

The public key *pubKey* is an elliptic curve point such that $pubKey = privKey \cdot G$, where G is the base point of the elliptic curve.

The memory size of the parameter *privKey* pointed by *pPrivate* must be less than that of the base point, which can also be defined by the function `ECCBGetOrderBitSize`.

The context of the point *privKey* as an elliptic curve point must be created by using the functions `ECCBPointGetSize` and `ECCBPointInit`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCBSet` or `ECCBSetStd`.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPrivate</i> , <i>pPublic</i> , or <i>pECC</i> is not valid.
<code>ippStsSizeErr</code>	Indicates an error condition if the memory size of the parameter <i>privKey</i> pointed by <i>pPrivate</i> is less than that of the order of the elliptic curve base point.

ECCBPublicKey

Computes a public key from the given private key of the elliptic cryptosystem over $GF(2^m)$.

Syntax

```
IppStatus ippseECCBPublicKey(const IppsBigNumState* pPrivate, IppsECCBPoint* pPublic,
IppsECCBState* pECC);
```

Parameters

<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function computes the public key *pubKey* from the given private key *privKey* of the elliptic cryptosystem over a binary finite field $GF(2^m)$.

The private key *privKey* is a number that lies in the range of $[1, n-1]$ where n is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $\text{pubKey} = \text{privKey} \cdot G$, where G is the base point of the elliptic curve.

The context of the point *pubKey* as an elliptic curve point must be created by using the functions [ECCBPointGetSize](#) and [ECCBPointInit](#).

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPrivate</i> , <i>pPublic</i> , or <i>pECC</i> is not valid.
<code>ippStsInvalidPrivateKey</code>	Indicates an error condition if the value of the private key falls outside the range of $[1, n-1]$.

ECCBValidateKeyPair

Validates private and secret keys of the elliptic cryptosystem over $GF(2^m)$.

Syntax

```
IppStatus ippseECCBValidateKeyPair(const IppsBigNumState* pPrivate, const IppsECCBPoint* pPublic, IppECResult* pResult, IppsECCBState* pECC);
```

Parameters

<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>pResult</i>	Pointer to the validation result.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function validates the private key *privKey* and public key *pubKey* of the elliptic cryptosystem over a binary finite field $GF(2^m)$ and allocates the result of the validation in accordance with the pointer *pResult*.

The private key *privKey* is a number that lies in the range of $[1, n-1]$ where n is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $\text{pubKey} = \text{privKey} \cdot G$, where G is the base point of the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

The result of the cryptosystem keys validation for correctness can take one of the following values:

<code>ippECValid</code>	Keys are valid.
<code>ippECInvalidKeyPair</code>	Keys are not valid because $\text{privKey} \cdot G \neq \text{pubKey}$

<code>ippECInvalidPrivateKey</code>	Key <i>privKey</i> falls outside the range of $[1, n-1]$.
<code>ippECPoinIsAtInfinite</code>	Key <i>pubKey</i> is the point at infinity.
<code>ippECInvalidPublicKey</code>	Key <i>pubKey</i> is not valid because $n \cdot \text{pubKey} \neq O$, where <i>O</i> is the point at infinity.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPrivate</i> , <i>pPublic</i> , or <i>pECC</i> is not valid.

ECCBSetKeyPair

Sets private and/or public keys in the elliptic cryptosystem over $GF(2^m)$.

Syntax

```
IppStatus ippsECCBSetKeyPair(const IppsBigNumState* pPrivate, const IppsECCBPoint* pPublic,
IppBool regular, IppsECCBState* pECC);
```

Parameters

<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>regular</i>	Key status flag.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function sets the private key *privKey* and/or public key *pubKey* in the elliptic cryptosystem defined over a binary finite field $GF(2^m)$.

The private key *privKey* is a number that lies in the range of $[1, n-1]$ where *n* is the order of the elliptic curve base point. The public key *pubKey* is an elliptic curve point such that $\text{pubKey} = \text{privKey} \cdot G$, where *G* is the base point of the elliptic curve.

The two possible values of the parameter *regular* define the key timeliness status:

<code>ippTrue</code>	Keys are regular.
<code>ippFalse</code>	Keys are ephemeral.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCBSet` or `ECCBSetStd`.

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 <code>NULL</code> .

`ippStsContextMatchErr`

Indicates an error condition if one of the contexts pointed by *pPrivate*, *pPublic*, or *pECC* is not valid.

ECCBSharedSecretDH

Computes a shared secret field element by using the Diffie-Hellman scheme.

Syntax

```
IppStatus ippseCCBSharedSecretDH(const IppsBigNumState* pPrivate, const IppsECCBPointState*
pPublic, IppsBigNumState* pShare, IppsECCBState* pECC);
```

Parameters

<i>pPrivate</i>	Pointer to your own public key <i>pubKey</i> .
<i>pPublic</i>	Pointer to the public key <i>pubKey</i> .
<i>pShare</i>	Pointer to the secret number <i>bnShare</i> .
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function computes a secret number *bnShare*, which is a secret key shared between two participants of the cryptosystem.

In cryptography, metasyntactic names such as Alice as Bob are normally used as examples and in discussions and stand for participant A and participant B.

Both participants (Alice and Bob) use the cryptosystem for receiving a common secret point on the elliptic curve called a secret key. To receive a secret key, participants apply the Diffie-Hellman key-agreement scheme involving public key exchange. The value of the secret key entirely depends on participants.

According to the scheme, Alice and Bob perform the following operations:

1. Alice calculates her own public key *pubKeyA* by using her private key *privKeyA*: $\text{pubKeyA} = \text{privKeyA} \cdot G$, where *G* is the base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key *pubKeyB* by using his private key *privKeyB*: $\text{pubKeyB} = \text{privKeyB} \cdot G$, where *G* is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point *shareA*. When calculating, she uses her own private key and Bob's public key and applies the following formula: $\text{shareA} = \text{privKeyA} \cdot \text{pubKeyB} = \text{privKeyA} \cdot \text{privKeyB} \cdot G$.
4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $\text{shareB} = \text{privKeyB} \cdot \text{pubKeyA} = \text{privKeyB} \cdot \text{privKeyA} \cdot G$.

Because the following equation is true $\text{privKeyA} \cdot \text{privKeyB} \cdot G = \text{privKeyB} \cdot \text{privKeyA} \cdot G$, the result of both calculations is the same, that is, the equation $\text{shareA} = \text{shareB}$ is true. The secret point serves as a secret key.

Shared secret *bnShare* is an x-coordinate of the secret point on the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pPublic</code> , <code>pPShare</code> , or <code>pECC</code> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if the memory size of <code>bnShare</code> pointed by <code>pShare</code> is less than the value of <code>feBitSize</code> in the function EC-CBInit .
<code>ippStsShareKeyErr</code>	Indicates an error condition if the shared secret key is not valid. (For example, the shared secret key is invalid if the result of the secret point calculation is the point at infinity.)

ECCBSharedSecretDHC

Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.

Syntax

```
IppStatus ippseccBSharedSecretDHC(const IppsBigNumState* pPrivate, const IppsECCBPointState* pPublic, IppsBigNumState* pShare, IppsECCBState* pECC);
```

Parameters

<code>pPrivate</code>	Pointer to your own public key <code>pubKey</code> .
<code>pPublic</code>	Pointer to the public key <code>pubKey</code> .
<code>pShare</code>	Pointer to the secret number <code>bnShare</code> .
<code>pECC</code>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function computes a secret number `bnShare` which is a secret key shared between two participants of the cryptosystem. Both participants (Alice and Bob) use the cryptosystem for getting a common secret point on the elliptic curve by using the Diffie-Hellman scheme and elliptic curve cofactor h .

Alice and Bob perform the following operations:

1. Alice calculates her own public key `pubKeyA` by using her private key `privKeyA`: $pubKeyA = privKeyA \cdot G$, where G is the base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key `pubKeyB` by using his private key `privKeyB`: $pubKeyB = privKeyB \cdot G$, where G is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point `shareA`. When calculating, she uses her own private key and Bob's public key and applies the following formula: $shareA = h \cdot privKeyA \cdot pubKeyB = h \cdot privKeyA \cdot privKeyB \cdot G$, where h is the elliptic curve cofactor.

4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $shareB = h \cdot privKeyB \cdot pubKeyA = h \cdot privKeyB \cdot privKeyA \cdot G$, where *h* is the elliptic curve cofactor.

Shared secret *bnShare* is an x-coordinate of the secret point on the elliptic curve.

To define a secret key, the call to the `ECCBSharedSecretDH` function must be preceded by the call to the `ECCBSetKeyPair` function.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCBSet` or `ECCBSetStd`.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <i>pPublic</i> , <i>pPShare</i> , or <i>pECC</i> is not valid.
<code>ippStsRangeErr</code>	Indicates an error condition if the memory size of <i>bnShare</i> pointed by <i>pShare</i> is less than the value of <i>feBitSize</i> in the function <code>EC-CBInit</code> .
<code>ippStsShareKeyErr</code>	Indicates an error condition if the shared secret key is not valid. (For example, the shared secret key is invalid if the result of the secret point calculation is the point at infinity.)

ECCBSignDSA

Computes a digital signature over a message digest.

Syntax

```
IppStatus ippseccbsigndsa(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pPrivate, IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCBState* pECC);
```

Parameters

<i>pMsgDigest</i>	Pointer to the message digest <i>msg</i> to be digitally signed, that is, to be encrypted with a private key.
<i>pPrivate</i>	Pointer to the signer's regular private key.
<i>pSignX</i>	Pointer to the integer <i>r</i> of the digital signature.
<i>pSignY</i>	Pointer to the integer <i>s</i> of the digital signature.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

A message digest is a fixed size number derived from the original message with an applied hash function over the binary code of the message. The signer's private key and the message digest are used to create a signature.

A digital signature over a message consists of a pair of large numbers *r* and *s* which the given function computes.

The scheme used for computing a digital signature is analogue of the ECDSA scheme, an elliptic curve analogue of the DSA scheme. ECDSA assumes that the following keys are hitherto set by a message signer:

<i>ephPrivKey</i>	Ephemeral private key.
<i>ephPubKey</i>	Ephemeral public key.

The keys can be generated and set up by the functions [ECCBGenKeyPair](#) and [ECCBSetKeyPair](#) with only requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

For more information on digital signatures, please refer to the [\[ANSI\]](#) standard.

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 <code>NULL</code> .
<i>ippStsContextMatchErr</i>	Indicates an error condition if one of the contexts pointed by <i>pMsgDigest</i> , <i>pSignX</i> , <i>pSignY</i> , or <i>ECC</i> is not valid.
<i>ippStsMessageErr</i>	Indicates an error condition if the value of <i>msg</i> pointed by <i>pMsgDigest</i> falls outside the range of $[1, n-1]$ where n is the order of the elliptic curve base point G .
<i>ippStsRangeErr</i>	Indicates an error condition if one of the parameters pointed by <i>pSignX</i> or <i>pSignY</i> has a less memory size than the order n of the elliptic curve base point G .
<i>ippStsEphemeralKeyErr</i>	Indicates an error condition if the values of the ephemeral keys <i>ephPrivKey</i> and <i>ephPubKey</i> are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

ECCBVerifyDSA

Verifies authenticity of the digital signature over a message digest (ECDSA).

Syntax

```
IppStatus ippseCCBVerifyDSA(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pSignX, const IppsBigNumState* pSignY, IppeCResult* pResult, IppsECCBState* pECC);
```

Parameters

<i>pMsgDigest</i>	Pointer to the message digest <i>msg</i> .
<i>pSignX</i>	Pointer to the integer r of the digital signature.
<i>pSignY</i>	Pointer to the integer s of the digital signature.
<i>pResult</i>	Pointer to the digital signature verification result.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function verifies authenticity of the digital signature over a message digest *msg*. The signature consists of two large integers: *r* and *s*.

The scheme used to verify the signature is an elliptic curve analogue of the DSA scheme and assumes that the following cryptosystem key be hitherto set:

regPubKey Message sender's regular public key.

The *regPubKey* is set by the function [ECCBSetKeyPair](#).

The result of the digital signature verification can take one of two possible values:

ippECValid Digital signature is valid.

ippECInvalidSignature Digital signature is not valid.

The call to the [ECCBVerifyDSA](#) function must be preceded by the call to the [ECCBSignDSA](#) function which computes the digital signature over the message digest *msg* and represents the signature with two numbers: *r* and *s*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

For more information on digital signatures, please refer to the [\[ANSI\]](#) standard.

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 <code>NULL</code> .
<i>ippStsContextMatchErr</i>	Indicates an error condition if one of the contexts pointed by <i>pMsgDigest</i> , <i>pSignX</i> , <i>pSignY</i> , or <i>ECC</i> is not valid.
<i>ippStsMessageErr</i>	Indicates an error condition if the value of <i>msg</i> pointed by <i>pMsgDigest</i> falls outside the range of $[1, n-1]$ where <i>n</i> is the order of the elliptic curve base point <i>G</i> .

ECCBSignNR

Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippseccbsignnr(const IppsBigNumState* pMsgDigest, const IppsBigNumState* pPrivate,
IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCBState* pECC);
```

Parameters

<i>pMsgDigest</i>	Pointer to the message digest <i>msg</i> .
<i>pPrivate</i>	Pointer to the private key <i>privKey</i> .
<i>pSignX</i>	Pointer to the integer <i>r</i> of the digital signature.
<i>pSignY</i>	Pointer to the integer <i>s</i> of the digital signature.
<i>pECC</i>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function computes two large numbers r and s which form the digital signature over a message digest msg .

The scheme used to compute the digital signature is an elliptic curve analogue of the El-Gamal Digital Signature scheme with the message recovery (the Nyberg-Rueppel signature scheme). The scheme that the given function uses assumes that the following cryptosystem keys are hitherto set up by the message sender:

<code>regPrivKey</code>	Regular private key.
<code>ephPrivKey</code>	Ephemeral private key.
<code>ephPubKey</code>	Ephemeral public key.

The keys can be generated and set up by the functions `ECCBGenKeyPair` and `ECCBSetKeyPair` with only requirement that the key `regPrivKey` be different from the key `ephPrivKey`.

The elliptic curve domain parameters must be hitherto defined by one of the functions: `ECCBSet` or `ECCBSetStd`.

For more information on digital signatures, please refer to the [ANSI] standard.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pMsgDigest</code> , <code>pSignX</code> , <code>pSignY</code> , or <code>ECC</code> is not valid.
<code>ippStsMessageErr</code>	Indicates an error condition if the value of <code>msg</code> pointed by <code>pMsgDigest</code> falls outside the range of $[1, n-1]$ where n is the order of the elliptic curve base point G .
<code>ippStsRangeErr</code>	Indicates an error condition if one of the parameters pointed by <code>pSignX</code> or <code>pSignY</code> has a less memory size than the order n of the elliptic curve base point G .
<code>ippStsEphemeralKeyErr</code>	Indicates an error condition if the values of the ephemeral keys <code>ephPrivKey</code> and <code>ephPubKey</code> are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

ECCBVerifyNR

Computes authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippseccbverifynr(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pSignX, const IppsBigNumState* pSignY, IppeCResult* pResult, IppsECCBState* pECC);
```

Parameters

<code>pMsgDigest</code>	Pointer to the message digest msg .
<code>pSignX</code>	Pointer to the integer r of the digital signature.

<code>pSignY</code>	Pointer to the integer s of the digital signature.
<code>pResult</code>	Pointer to the digital signature verification result.
<code>pECC</code>	Pointer to the context of the elliptic cryptosystem.

Description

This function is declared in the `ippcp.h` file.

The function verifies authenticity of the digital signature over a message digest `msg`. The signature is presented with two large integers r and s .

The scheme used to compute the digital signature is an elliptic curve analogue of the El-Gamal Digital Signature scheme with the message recovery (the Nyberg-Rueppel signature scheme). The scheme that the given function uses assumes that the following cryptosystem keys be hitherto set up by the message sender:

`regPubKey` Message sender's regular private key.

The key can be generated and set up by the function [ECCBGenKeyPair](#).

The result of the digital signature verification can take one of two possible values:

<code>ippECValid</code>	The digital signature is valid.
<code>ippECInvalidSignature</code>	The digital signature is not valid.

The call to the `ECCBVerifyNR` function must be preceded by the call to the [ECCBSignNR](#) function which computes the digital signature over the message digest `msg` and represents the signature with two numbers: r and s .

The elliptic curve domain parameters must be hitherto defined by one of the functions: [ECCBSet](#) or [ECCBSetStd](#).

For more information on digital signatures, please refer to the [\[ANSI\]](#) standard.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if one of the contexts pointed by <code>pMsgDigest</code> , <code>pSignX</code> , <code>pSignY</code> , or <code>ECC</code> is not valid.
<code>ippStsMessageErr</code>	Indicates an error condition if the value of <code>msg</code> pointed by <code>pMsgDigest</code> falls outside the range of $[1, n-1]$ where n is the order of the elliptic curve base point G .

Finite Field Arithmetic

This section describes the Intel IPP primitives that implement arithmetic operations with elements of finite fields [\[ANT\]](#). Arithmetic of the following finite fields is implemented:

$\text{GF}(p)$	A finite field of p elements represented by integers modulo p , where p is an odd prime number. This field is also known as a <i>prime finite field</i> .
$\text{GF}(p^d)$	A finite field of p^d elements represented by equivalence classes modulo $g(x)$ of polynomials whose coefficients belong to $\text{GF}(p)$, where $g(x)$ is an irreducible polynomial of degree d . Coefficients of $g(x)$ are elements of the $\text{GF}(p)$ field. This field is also known as an <i>extension field of degree d of $\text{GF}(p)$</i> or the <i>Galois field</i> .

$GF(p^{d^2})$ A quadratic extension of $GF(p^d)$.

Table “Intel IPP Finite Field Arithmetic Functions” lists all the finite field arithmetic functions.

Intel IPP Finite Field Arithmetic Functions

Function Base Name	Operation
Arithmetic of Finite Fields $GF(p)$	
<code>GFPGetSize</code>	Gets the size of the context of a $GF(p)$ field.
<code>GFPInit</code>	Initializes the context of a $GF(p)$ field.
<code>GFPGet</code>	Extracts parameters of the $GF(p)$ field from the context.
<code>GFPElementGetSize</code>	Gets the size of the context for an element of the $GF(p)$ field.
<code>GFPElementInit</code>	Initializes the context of an element of the $GF(p)$ field.
<code>GFPSetElement</code>	Assigns a value to an element of the $GF(p)$ field.
<code>GFPSetElementZero</code>	Assigns the zero value to an element of the $GF(p)$ field.
<code>GFPSetElementPower2</code>	Assigns the value of a given power of two to an element of the $GF(p)$ field.
<code>GFPSetElementRandom</code>	Assigns a random value to an element of the $GF(p)$ field.
<code>GFPCpyElement</code>	Copies one element of the $GF(p)$ field to another.
<code>GFPGetElement</code>	Extracts the element of the $GF(p)$ field from the context.
<code>GFPCmpElement</code>	Compares elements of the $GF(p)$ field.
<code>GFPNeg</code>	Computes the additive inverse for an element of the $GF(p)$ field.
<code>GFPIinv</code>	Computes the multiplicative inverse for an element of the $GF(p)$ field.
<code>GFPSqrt</code>	Takes the square root of an element of the $GF(p)$ field.
<code>GFPAAdd</code>	Adds elements of the $GF(p)$ field.
<code>GFPSub</code>	Subtracts elements of the $GF(p)$ field.
<code>GFPMul</code>	Multiplies elements of the $GF(p)$ field.
<code>GFPExp</code>	Exponentiates an element of the $GF(p)$ field.
<code>GFPMontEncode</code>	Converts an element of the $GF(p)$ field to the Montgomery residue number system.
<code>GFPMontDecode</code>	Converts an element of the $GF(p)$ field represented in the Montgomery residue number system to the regular $GF(p)$ element.
Arithmetic of Finite Fields $GF(p^d)$	
<code>GFPIXGetSize</code>	Gets the size of the context of a $GF(p^d)$ field.
<code>GFPIXInit</code>	Initializes the context of a $GF(p^d)$ field.
<code>GFPIXGet</code>	Extracts parameters of the $GF(p^d)$ field from the context.
<code>GFPIXElementGetSize</code>	Gets the size of the context for an element of the $GF(p^d)$ field.
<code>GFPIXElementInit</code>	Initializes the context of an element of the $GF(p^d)$ field.
<code>GFPIXSetElement</code>	Assigns a value to an element of the $GF(p^d)$ field.
<code>GFPIXSetElementZero</code>	Assigns the zero value to an element of the $GF(p^d)$ field.
<code>GFPIXSetElementPowerX</code>	Assigns the value of a given power of x to an element of the $GF(p^d)$ field.
<code>GFPIXSetElementRandom</code>	Assigns a random value to an element of the $GF(p^d)$ field.
<code>GFPIXCpyElement</code>	Copies one element of the $GF(p^d)$ field to another.
<code>GFPIXGetElement</code>	Extracts the element of the $GF(p^d)$ field from the context.
<code>GFPIXCmpElement</code>	Compares elements of the $GF(p^d)$ field.

Function Base Name	Operation
GFPXNeg	Computes the additive inverse for an element of the $GF(p^d)$ field.
GFPXInv	Computes the multiplicative inverse for an element of the $GF(p^d)$ field.
GFPXAdd	Adds elements of the $GF(p^d)$ field.
GFPXAdd_GFP	Adds elements of the $GF(p^d)$ and $GF(p)$ fields.
GFPXSub	Subtracts elements of the $GF(p^d)$ field.
GFPXSub_GFP	Subtracts elements of the $GF(p^d)$ and $GF(p)$ fields.
GFPXMul	Multiplies elements of the $GF(p^d)$ field.
GFPXMul_GFP	Multiplies elements of the $GF(p^d)$ and $GF(p)$ fields.
GFPXExp	Exponentiates an element of the $GF(p^d)$ field.
GFPXDiv	Divides elements of the $GF(p^d)$ field.
Arithmetic of Finite Fields $GF(p^{d^2})$	
GFPXQGetSize	Gets the size of the context of a $GF(p^{d^2})$ field.
GFPXQInit	Initializes the context of a $GF(p^{d^2})$ field.
GFPXQGet	Extracts parameters of the $GF(p^{d^2})$ field from the context.
GFPXQElementGetSize	Gets the size of the context for an element of the $GF(p^{d^2})$ field.
GFPXQElementInit	Initializes the context of an element of the $GF(p^{d^2})$ field.
GFPXQSetElement	Assigns a value to an element of the $GF(p^{d^2})$ field.
GFPXQSetElementZero	Assigns the zero value to an element of the $GF(p^{d^2})$ field.
GFPXQSetElementPowerX	Assigns the value of a given power of x to an element of the $GF(p^{d^2})$ field.
GFPXQSetElementRandom	Assigns a random value to an element of the $GF(p^{d^2})$ field.
GFPXQCpyElement	Copies one element of the $GF(p^{d^2})$ field to another.
GFPXQGetElement	Extracts the element of the $GF(p^{d^2})$ field from the context.
GFPXQCmpElement	Compares elements of the $GF(p^{d^2})$ field.
GFPXQNeg	Computes the additive inverse for an element of the $GF(p^{d^2})$ field.
GFPXQInv	Computes the multiplicative inverse for an element of the $GF(p^{d^2})$ field.
GFPXQAdd	Adds elements of the $GF(p^{d^2})$ field.
GFPXQSub	Subtracts elements of the $GF(p^{d^2})$ field.
GFPXQMul	Multiplies elements of the $GF(p^{d^2})$ field.
GFPXQMul_GFP	Multiplies elements of the $GF(p^{d^2})$ and $GF(p)$ fields.
GFPXQExp	Exponentiates an element of the $GF(p^{d^2})$ field.

Each element E of $GF(p)$ is represented by an unsigned big number, which, in turn, is represented by a data array `Ipp32u pe[length]`, so that

$$E = \sum_{0 \leq i < \text{length}} pe[i] * 2^{(32*i)}$$

Each element E of $GF(p^d)$ is represented by a polynomial of degree less than d , which, in turn, is represented by an array of coefficients `pe[d]` that belong to $GF(p)$.

Each element E of $GF(p^{d^2})$ is represented by a polynomial of degree less than 2, which, in turn, is represented by an array of coefficients `pe[2]` that belong to $GF(p^d)$.

For polynomials that represent elements of both fields, a coefficient of a lower degree is stored in an element of the respective array with a smaller index.

The Intel IPP finite field arithmetic functions use context structures of the following types to carry data of the field and field element:

$GF(p)$	<code>IppsGFpState</code> and <code>IppsGFpElement</code> , respectively.
$GF(p^d)$	<code>IppsGFpXState</code> and <code>IppsGFpXElement</code> , respectively.
$GF(p^{d^2})$	<code>IppsGFpXQState</code> and <code>IppsGFpXQElement</code> , respectively.

Comparison functions `GFpCmpElement`, `GFpXCmpElement`, and `GFpXQCmpElement` return the result of comparison:

```
typedef enum {
    IppsElementEQ = 0, // elements are equal
    IppsElementNE = 1, // elements are not equal
    IppsElementGT = 2, // the first element is greater than the second one
    IppsElementLT = 3, // the first element is less than the second one
    IppsElementNA = 4 // elements are not comparable
} IppsElementCmpResult;
```

Functions for the $GF(p)$ Field

GFpGetSize

Gets the size of the `IppsGFpState` context of the finite field $GF(p)$.

Syntax

```
IppStatus ippsGFpGetSize(Ipp32u bitSize, Ipp32u *stateSizeInBytes);
```

Parameters

<code>bitSize</code>	Size in bytes of the odd prime number p (modulus of $GF(p)$).
<code>stateSizeInBytes</code>	Buffer size in bytes needed for the <code>IppsGFpState</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFpState` context, suitable for storing data for the finite field $GF(p)$ determined by the odd prime number p of size not greater than `bitSize` bit.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if <code>bitSize</code> is less than 2.

GFPInit

Initializes the IppsGFPState context of the finite field $GF(p)$.

Syntax

```
IppStatus ippsGFPInit(IppsGFPState *pGFP, Ipp32u *pPrime, Ipp32u bitSize);
```

Parameters

<i>pGFP</i>	Pointer to the context of the $GF(p)$ field being initialized.
<i>pPrime</i>	Pointer to the data storing the $GF(p)$ modulus.
<i>bitSize</i>	Size of the $GF(p)$ modulus.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory buffer *pGFP* associated with the IppsGFPState context and sets up the specific value of the $GF(p)$ modulus. The initialized context is used in the functions that create contexts of elements of the $GF(p)$ field and perform operations with the field elements.



NOTE. The function does not check primality of the modulus.

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 <code>NULL</code> .
<code>ippStsSizeErr</code>	Indicates an error condition if <i>bitSize</i> is less than 2.

GFPGet

Extracts parameters of the finite field $GF(p)$ from the input structure.

Syntax

```
IppStatus ippsGFPGet(const IppsGFPState *pGFP, const Ipp32u **ppPrime, Ipp32u *elementLen);
```

Parameters

<i>pGFP</i>	Pointer to the context of the $GF(p)$ field.
<i>ppPrime</i>	Address of the pointer to the data array storing the $GF(p)$ modulus.
<i>elementLen</i>	Length of a $GF(p)$ element.

Description

This function is declared in the `ippcp.h` file. The function extracts parameters of the $GF(p)$ field from the input `IppsGFpState` context. You can get the following parameters of the finite field or any of them: a reference to the modulus and the element length. To turn off extraction of a particular field parameter, set the appropriate function parameter to `NULL`.

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 <code>pGFp</code> pointer is <code>NULL</code> .

GFPElementGetSize

Gets the size of the `IppsGFPElement` context of a $GF(p)$ element.

Syntax

```
IppStatus ippGFPElementGetSize(const IppsGFpState *pGFp, Ipp32u *stateSizeInBytes);
```

Parameters

<code>pGFp</code>	Pointer to the <code>IppsGFpState</code> context of the $GF(p)$ field.
<code>stateSizeInBytes</code>	Buffer size in bytes needed for the of the <code>IppsGFPElement</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPElement` context, suitable for storing an element of the finite field $GF(p)$.

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 <code>NULL</code> .

GFPElementInit

Initializes the `IppsGFPElement` context of a $GF(p)$ element.

Syntax

```
IppStatus ippGFPElementInit(IppsGFPElement *pGFpElement, const Ipp32u *pData, Ipp32u dataLen, IppsGFpState *pGFp);
```

Parameters

<code>pGFpElement</code>	Pointer to the context of the $GF(p)$ element being initialized.
<code>pData</code>	Pointer to the data array storing the $GF(p)$ element.

dataLen Length of the element.
pGFp Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory buffer *pGFpElement* associated with the `IppsGFPElement` context and sets up the specific element of the $GF(p)$ field. The initialized `IppsGFPElement` context is used in all the operations with this element of the $GF(p)$ field.

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`.
`ippContextMatchErr` Indicates an error condition if any of the `IppsGFpState` or `IppsGFPElement` context parameters does not match the operation.
`ippStsSizeErr` Indicates an error condition if *dataLen* is ten times greater than the $GF(p)$ modulus.

GFPSetElement

Assigns a value to an element of the $GF(p)$ field.

Syntax

```
IppStatus ippsgfpSetElement(const Ipp32u *pData, Ipp32u dataLen, IppsGFPElement
*pGFpElement, IppsGFpState *pGFp);
```

Parameters

pData Pointer to the data array storing the element of the $GF(p)$ field.
dataLen Length of the $GF(p)$ element.
pGFpElement Pointer to the context of the $GF(p)$ element.
pGFp Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function copies (and converts if needed) the value from the user-defined *pData* buffer to the `IppsGFPElement` context of the $GF(p)$ element.

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`.
`ippContextMatchErr` Indicates an error condition if any of the `IppsGFpState` or `IppsGFPElement` context parameters does not match the operation.
`ippStsSizeErr` Indicates an error condition if *dataLen* is ten times greater than the $GF(p)$ modulus.

GFPSetElementZero

Assigns the zero value to an element of the $GF(p)$ field.

Syntax

```
IppStatus ippGFPSetElementZero(IppsGFPElement *pGFPElement, IppsGFPState *pGFp);
```

Parameters

<i>pGFPElement</i>	Pointer to the context of the $GF(p)$ element.
<i>pGFp</i>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function assigns the zero value to the given element of the $GF(p)$ 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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPSetElementPower2

Assigns the value of a given power of two to an element of the $GF(p)$ field.

Syntax

```
IppStatus ippGFPSetElementPower2(Ipp32u power, IppsGFPElement *pGFPElement, IppsGFPState *pGFp);
```

Parameters

<i>power</i>	The exponent.
<i>pGFPElement</i>	Pointer to the context of the $GF(p)$ element.
<i>pGFp</i>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function sets (and converts if needed) the value of the given element of the $GF(p)$ field to 2^{power} .

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 <code>NULL</code> .

<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFpElement</code> context parameters does not match the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if $(power + 1)$ is ten times greater than the $GF(p)$ modulus in bits.

GFPSetElementRandom

Assigns a random value to an element of the $GF(p)$ field.

Syntax

```
IppStatus ippsgfpSetElementRandom(IppsGFpElement *pGFpElement, IppsGFpState *pGFp,
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<code>pGFpElement</code>	Pointer to the context of the $GF(p)$ element.
<code>pGFp</code>	Pointer to the context of the $GF(p)$ field.
<code>rndFunc</code>	Function that generates random sequences.
<code>pRndParam</code>	Pointer to the context of a random sequence generator.

Description

This function is declared in the `ippcp.h` file. The function assigns a random value to the given element of the $GF(p)$ 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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFpElement</code> context parameters does not match the operation.

See Also

- [Pseudorandom Number Generation Functions](#)

GFPCmpElement

Compares two elements of the $GF(p)$ field.

Syntax

```
IppStatus ippsgfpCmpElement(const IppsGFpElement *pGFpElementA, const IppsGFpElement
*pGFpElementB, IppsElementCmpResult *cmpResult, const IppsGFpState *pGFp);
```

Parameters

<code>pGFpElementA</code>	Pointer to the context of the first $GF(p)$ element.
<code>pGFpElementB</code>	Pointer to the context of the second $GF(p)$ element.

<code>cmpResult</code>	Result of the comparison. For details, see comparison functions .
<code>pGFp</code>	Pointer to the context of the GF(p) field.

Description

This function is declared in the `ippcp.h` file. The function compares two elements of the GF(p) field and returns the result in `*cmpResult`.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFpElement</code> context parameters does not match the operation.

GFPCpyElement

Copies one element of the GF(p) field to another.

Syntax

```
IppStatus ippGFPCpyElement(const IppsGFpElement *pGFpElementA, IppsGFpElement
*pGFpElementB, const IppsGFpState *pGFp);
```

Parameters

<code>pGFpElementA</code>	Pointer to the context of the GF(p) element being copied.
<code>pGFpElementB</code>	Pointer to the context of the GF(p) element being changed.
<code>pGFp</code>	Pointer to the context of the GF(p) field.

Description

This function is declared in the `ippcp.h` file. The function copies one element of the GF(p) field to another.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFpElement</code> context parameters does not match the operation.

GFPGetElement

*Extracts the element of the GF(p) field from an input
IppsGFpElement context.*

Syntax

```
IppStatus ippGFPGetElement(const IppsGFpElement *pGFpElement, Ipp32u *pData, Ipp32u
dataLen, const IppsGFpState *pGFp);
```

Parameters

<code>pGFpElement</code>	Pointer to the context of the $GF(p)$ element.
<code>pData</code>	Pointer to the data array to copy the $GF(p)$ element from.
<code>dataLen</code>	Length of the data array.
<code>pGFp</code>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function copies the element of the $GF(p)$ field from the `IppsGFPElement` context to the user-defined `*pData` buffer.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPNeg

Computes the additive inverse for an element of the $GF(p)$ field.

Syntax

```
IppStatus ippsgFPNeg(const IppsGFPElement *pGFpElementA, IppsGFPElement *pGFpElementR,
IppsGFpState *pGFp);
```

Parameters

<code>pGFpElementA</code>	Pointer to the context of the given $GF(p)$ element.
<code>pGFpElementR</code>	Pointer to the context of the resulting $GF(p)$ element.
<code>pGFp</code>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the additive inverse for a given element of the $GF(p)$ field. The following pseudocode represents this operation: $R = 0 - A$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPIInv

Computes the multiplicative inverse for an element of the GF(p) field.

Syntax

```
IppStatus ippsGFPIInv(const IppsGFPElement *pGFpElementA, IppsGFPElement *pGFpElementR,
IppsGFpState *pGFp);
```

Parameters

<i>pGFpElementA</i>	Pointer to the context of the given GF(<i>p</i>) element.
<i>pGFpElementR</i>	Pointer to the context of the resulting GF(<i>p</i>) element.
<i>pGFp</i>	Pointer to the context of the GF(<i>p</i>) field.

Description

This function is declared in the `ippcp.h` file. The function computes the multiplicative inverse for a given element of the GF(*p*) field. The following pseudocode represents this operation: $R = A^{(-1)}$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.
<code>ippStsDivByZeroErr</code>	Indicates an error condition if the zero element is attempted.

GFPSqrt

Takes the square root of an element of the GF(p) field.

Syntax

```
IppStatus ippsGFPSqrt(const IppsGFPElement *pGFpElementA, IppsGFPElement *pGFpElementR,
IppsGFpState *pGFp);
```

Parameters

<i>pGFpElementA</i>	Pointer to the context of the given GF(<i>p</i>) element.
<i>pGFpElementR</i>	Pointer to the context of the resulting GF(<i>p</i>) element.
<i>pGFp</i>	Pointer to the context of the GF(<i>p</i>) field.

Description

This function is declared in the `ippcp.h` file. The function computes the square root of a given element of the GF(*p*) field. The following pseudocode represents this operation: $R = A^{(1/2)}$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.
<code>ippStsSqrtNegErr</code>	Indicates an error condition if a square non-residue element is attempted.

GFPAAdd

Adds two elements of the $GF(p)$ field.

Syntax

```
IppStatus ippsgfpaadd(const IppsGFPElement *pGFpElementA, const IppsGFPElement *pGFpElementB,
IppsGFPElement *pGFpElementR, IppsGFPState *pGFP);
```

Parameters

<code>pGFpElementA</code>	Pointer to the context of the first summand element of the $GF(p)$ field.
<code>pGFpElementB</code>	Pointer to the context of the second summand element of the $GF(p)$ field.
<code>pGFpElementR</code>	Pointer to the context of the resulting element of the $GF(p)$ field.
<code>pGFP</code>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the sum of the two elements of the $GF(p)$ field. The following pseudocode represents this operation: $R = A + B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPSub

Subtracts one element of the $GF(p)$ field from another.

Syntax

```
IppStatus ippsgfpsub(const IppsGFPElement *pGFpElementA, const IppsGFPElement *pGFpElementB,
IppsGFPElement *pGFpElementR, IppsGFPState *pGFP);
```

Parameters

<code>pGFpElementA</code>	Pointer to the context of the minuend element of the $GF(p)$ field.
<code>pGFpElementB</code>	Pointer to the context of the subtrahend element of the $GF(p)$ field.
<code>pGFpElementR</code>	Pointer to the context of the resulting element of the $GF(p)$ field.
<code>pGFp</code>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the difference of the two elements of the $GF(p)$ field. The following pseudocode represents this operation: $R = A - B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFpElement</code> context parameters does not match the operation.

GFPMul

Multiplies two elements of the $GF(p)$ field.

Syntax

```
IppsStatus ippGFPMul(const IppsGFpElement *pGFpElementA, const IppsGFpElement *pGFpElementB,
IppsGFpElement *pGFpElementR, IppsGFpState *pGFp);
```

Parameters

<code>pGFpElementA</code>	Pointer to the context of the first multiplicand element of the $GF(p)$ field.
<code>pGFpElementB</code>	Pointer to the context of the second multiplicand element of the $GF(p)$ field.
<code>pGFpElementR</code>	Pointer to the context of the resulting element of the $GF(p)$ field.
<code>pGFp</code>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the product of the two elements of the $GF(p)$ field. The following pseudocode represents this operation: $R = A * B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFpElement</code> context parameters does not match the operation.

GFPExp

Raises an element of the $GF(p)$ field to a non-negative power.

Syntax

```
IppStatus ippsGFPExp(const IppsGFPElement *pGFpElementA, const IppsGFPElement *pGFpElementE,
IppsGFPElement *pGFpElementR, IppsGFpState *pGFp);
```

Parameters

<i>pGFpElementA</i>	Pointer to the context of the $GF(p)$ element that represents the base of the exponentiation.
<i>pGFpElementE</i>	Pointer to the context of the $GF(p)$ element that represents the exponent.
<i>pGFpElementR</i>	Pointer to the context of the resulting element of the $GF(p)$ field.
<i>pGFp</i>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function raises the element of the $GF(p)$ field to the given non-negative power. The following pseudocode represents this operation: $R = A^E$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPMontEncode

Converts an element of the $GF(p)$ field to a Montgomery residue number system.

Syntax

```
IppStatus ippsGFPMontEncode(const IppsGFPElement *pGFpElementA, IppsGFPElement
*pGFpElementR, IppsGFpState *pGFp);
```

Parameters

<i>pGFpElementA</i>	Pointer to the context of the given element of the $GF(p)$ field.
<i>pGFpElementR</i>	Pointer to the context of the resulting element of the $GF(p)$ field.
<i>pGFp</i>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function converts the given element of the $GF(p)$ field to the Montgomery residue number system. The following pseudocode represents this operation: $R = \text{MontMul}(A, M^2)$, where $M > p$ and $\text{gcd}(M, p) = 1$ (here *MontMul* denotes the Montgomery multiplication and *gcd* denotes the greatest common divisor).

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPMontDecode

Converts an element of the $GF(p)$ field represented in the Montgomery residue number system to the regular $GF(p)$ element.

Syntax

```
IppStatus ippGFPMontDecode(const IppsGFPElement *pGFpElementA, IppsGFPElement
*pGFpElementR, IppsGFpState *pGFp);
```

Parameters

<code>pGFpElementA</code>	Pointer to the context of the given element of the $GF(p)$ field.
<code>pGFpElementR</code>	Pointer to the context of the resulting element of the $GF(p)$ field.
<code>pGFp</code>	Pointer to the context of the $GF(p)$ field.

Description

This function is declared in the `ippcp.h` file. The function converts the element of the $GF(p)$ field represented in the Montgomery residue number system to the regular $GF(p)$ element. The following pseudocode represents this operation: $R = \text{MontMul}(A, 1)$ (here *MontMul* denotes the Montgomery 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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpState</code> or <code>IppsGFPElement</code> context parameters does not match the operation.

Functions for the $GF(p^d)$ Field

GFPXGetSize

Gets the size of the `IppsGFPXState` context of the finite field $GF(p^d)$.

Syntax

```
IppStatus ippsGFPXGetSize(const IppsGFPState *pGroundGFp, Ipp32u degree, Ipp32u
*stateSizeInBytes);
```

Parameters

<code>pGroundGFp</code>	Pointer to the ground finite field $GF(p)$.
<code>degree</code>	The degree of the extension.
<code>stateSizeInBytes</code>	Buffer size in bytes needed for the <code>IppsGFPXState</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPXState` context, suitable for storing data for the finite field $GF(p^d)$ determined by the extension degree of the finite field $GF(p)$ supplied in the `degree` parameter.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the <code>IppsGFPState</code> context parameter does not match the operation.
<code>ippStsSizeErr</code>	Indicates an error condition if the degree of the extension exceeds or equals nine.

GFPXInit

Initializes the `IppsGFPXState` context of the finite field $GF(p^d)$.

Syntax

```
IppStatus ippsGFPXInit(IppsGFPXState *pGFpx, IppsGFPState *pGroundGFp, Ipp32u degree,
const Ipp32u *pIrrPoly);
```

Parameters

<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field being initialized.
<code>pGroundGFp</code>	Pointer to the context of the ground field $GF(p)$.
<code>degree</code>	The degree of the extension.

pIrrPoly

Pointer to the array with coefficients of the irreducible polynomial.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory buffer **pGFpx* associated with the `IppsGFPXState` context and sets up the specific irreducible polynomial. The initialized context is used in the functions that create contexts of elements of the $GF(p^d)$ field and perform operations with the field elements.



NOTE. The function does not check primality of the polynomial.



Important. While you are calling the functions over the $GF(p^d)$ field, a properly initialized **pGroundGFp* context of the ground field $GF(p)$ is required.

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 the `IppsGFPState` context parameter does not match the operation.

GFPXGet

Extracts parameters of the finite field $GF(p^d)$ from the input structure.

Syntax

```
IppStatus ippGFPXGet(const IppsGFPXState *pGFpx, const Ipp32u **ppGroundGFp, Ipp32u
*pIrrPoly, Ipp32u *polyDegree, Ipp32u *polyLen, Ipp32u *elementLen);
```

Parameters

pGFpx

Pointer to the context of the $GF(p^d)$ field.

pGroundGFp

Pointer to the ground $GF(p)$ field.

pIrrPoly

Pointer to the array with coefficients of the irreducible polynomial.

polyDegree

The degree of the polynomial.

polyLen

Length of the array with the coefficients of the polynomial.

elementLen

Length of a $GF(p^d)$ element.

Description

This function is declared in the `ippcp.h` file. The function extracts parameters of the $GF(p^d)$ field from the `IppsGFPXState` context. You can get any combination of the following field parameters: a reference to the ground field $GF(p)$, a copy of the irreducible polynomial, its degree and length, as well as the length of an element of the $GF(p^d)$ field. To turn off extraction of a particular function parameter, set the appropriate function parameter to `NULL`.

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 <code>pGFpx</code> pointer is <code>NULL</code> .

GFPXElementGetSize

Gets the size of the `IppsGFPXElement` context of a $GF(p^d)$ element.

Syntax

```
ippStatus ippGFPXElementGetSize(const IppsGFPXState *pGFpx, Ipp32u *stateSizeInBytes);
```

Parameters

<code>pGFpx</code>	Pointer to the <code>IppsGFPXState</code> context of the $GF(p^d)$ field.
<code>stateSizeInBytes</code>	Buffer size in bytes needed for the <code>IppsGFPXElement</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPXElement` context, suitable for storing an element of the finite field $GF(p^d)$.

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 <code>NULL</code> .

GFPXElementInit

Initializes the `IppsGFPXElement` context of a $GF(p^d)$ element.

Syntax

```
ippStatus ippGFPXElementInit(IppsGFPXElement *pGFpxElement, const Ipp32u *pData, Ipp32u dataLen, IppsGFPXState *pGFpx);
```

Parameters

<code>pGFpxElement</code>	Pointer to the context of the $GF(p^d)$ element being initialized.
<code>pData</code>	Pointer to the data array storing the $GF(p^d)$ element.

<code>dataLen</code>	Length of the element.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory buffer `*pGFpxElement` associated with the `IppsGFpxElement` context and sets up the specific element of the $GF(p^d)$ field. The initialized `IppsGFpxElement` context is used in all the operations with this element of the $GF(p^d)$ 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 <code>pGFpxElement</code> or <code>pGFpx</code> pointers is <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if the <code>IppsGFpxState</code> context parameters does not match the operation.

GFPXSetElement

Assigns a value to an element of the $GF(p^d)$ field.

Syntax

```
IppStatus ippsgfpxSetElement(const Ipp32u *pData, Ipp32u dataLen, IppsGFpxElement
*pGFpxElement, IppsGFpxState *pGFpx);
```

Parameters

<code>pData</code>	Pointer to the data array storing the element of the $GF(p^d)$ field.
<code>dataLen</code>	Length of the $GF(p^d)$ element.
<code>pGFpxElement</code>	Pointer to the context of the $GF(p^d)$ element.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function copies (and converts if needed) the value from the user-defined `*pData` buffer to the `IppsGFpxElement` context of the $GF(p^d)$ element.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpxState</code> or <code>IppsGFpxElement</code> context parameters does not match the operation.

GFPXSetElementZero

Assigns the zero value to an element of the $GF(p^d)$ field.

Syntax

```
IppStatus ippsGFPXSetElementZero(IppsGFPXElement *pGFpxElement, IppsGFPXState *pGFpx);
```

Parameters

<i>pGFpxElement</i>	Pointer to the context of the $GF(p^d)$ element.
<i>pGFpx</i>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function assigns the zero value to the given element of the $GF(p^d)$ 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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.

GFPXSetElementPowerX

Assigns the value of a given power of x to an element of the $GF(p^d)$ field.

Syntax

```
IppStatus ippsGFPXSetElementPowerX(Ipp32u power, IppsGFPXElement *pGFpxElement, IppsGFPXState *pGFpx);
```

Parameters

<i>power</i>	The exponent.
<i>pGFpxElement</i>	Pointer to the context of the $GF(p^d)$ element.
<i>pGFpx</i>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function sets (and converts if needed) the value of the given element of the $GF(p^d)$ field to x^{power} .

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.

GFPXSetElementRandom

Assigns a random value to an element of the $GF(p^d)$ field.

Syntax

```
IppStatus ippGFPXSetElementRandom(IppsGFPXElement *pGFpxElement, IppsGFPXState *pGFpx,
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<code>pGFpxElement</code>	Pointer to the context of the $GF(p^d)$ element.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.
<code>rndFunc</code>	Function that generates random sequences.
<code>pRndParam</code>	Pointer to the context of a random sequence generator.

Description

This function is declared in the `ippcp.h` file. The function assigns a random value to the given element of the $GF(p^d)$ 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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.

See Also

- [Pseudorandom Number Generation Functions](#)

GFPXCmpElement

Compares two elements of the $GF(p^d)$ field.

Syntax

```
IppStatus ippGFPXCmpElement(const IppsGFPXElement *pGFpxElementA, const IppsGFPXElement
* pGFpxElementB, IppsElementCmpResult *cmpResult, const IppsGFPXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the first element of the $GF(p^d)$ field.
<code>pGFpxElementB</code>	Pointer to the context of the second element of the $GF(p^d)$ field.
<code>cmpResult</code>	Result of the comparison. For details, see comparison functions .
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function compares two elements of the $GF(p^d)$ field and returns the result in `*cmpResult`.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpXState</code> or <code>IppsGFpXElement</code> context parameters does not match the operation.

GFpXCpyElement

Copies one element of the $GF(p^d)$ field to another.

Syntax

```
IppStatus ippGFpXCpyElement(const IppsGFpXElement *pGFpxElementA, IppsGFpXElement
*pGFpxElementB, const IppsGFpXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the $GF(p^d)$ element being copied.
<code>pGFpxElementB</code>	Pointer to the context of the $GF(p^d)$ element being changed.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function copies one element of the $GF(p^d)$ field to another.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpXState</code> or <code>IppsGFpXElement</code> context parameters does not match the operation.

GFPXGetElement

Extracts the element of the $GF(p^d)$ field from an input IppsGFPXElement context.

Syntax

```
IppStatus ippsGFPXGetElement(const IppsGFPXElement *pGFpxElement, Ipp32u *pData, Ipp32u dataLen, const IppsGFPXState *pGFpx);
```

Parameters

<i>pGFpxElement</i>	Pointer to the context of the $GF(p^d)$ element.
<i>pData</i>	Pointer to the data array to copy the $GF(p^d)$ element from.
<i>dataLen</i>	Length of the data array.
<i>pGFpx</i>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function copies the element of the $GF(p^d)$ field from the IppsGFPXElement context to the user-defined *pData* buffer.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the IppsGFPXState or IppsGFPXElement context parameters does not match the operation.

GFPXNeg

Computes the additive inverse for an element of the $GF(p^d)$ field.

Syntax

```
IppStatus ippsGFPXNeg(const IppsGFPXElement *pGFpxElementA, IppsGFPXElement *pGFpxElementR, IppsGFPXState *pGFpx);
```

Parameters

<i>pGFpxElementA</i>	Pointer to the context of the given $GF(p^d)$ element.
<i>pGFpxElementR</i>	Pointer to the context of the resulting $GF(p^d)$ element.
<i>pGFpx</i>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the additive inverse for a given element of the $GF(p^d)$ field. The following pseudocode represents this operation: $R = 0 - A$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.

GFPXInv

Computes the multiplicative inverse for an element of the $GF(p^d)$ field.

Syntax

```
IppStatus ippGFPXInv(const IppsGFPXElement *pGFpxElementA, IppsGFPXElement *pGFpxElementR,
IppsGFPXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the given $GF(p^d)$ element.
<code>pGFpxElementR</code>	Pointer to the context of the resulting $GF(p^d)$ element.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the multiplicative inverse for a given element of the $GF(p^d)$ field. The following pseudocode represents this operation: $R = A^{(-1)}$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.
<code>ippStsDivByZeroErr</code>	Indicates an error condition if the zero element is attempted.

GFPXAdd

Adds two elements of the $GF(p^d)$ field.

Syntax

```
IppStatus ippGFPXAdd(const IppsGFPXElement *pGFpxElementA, const IppsGFPXElement
*pGFpxElementB, IppsGFPXElement *pGFpxElementR, IppsGFPXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the first summand element of the $GF(p^d)$ field.
<code>pGFpxElementB</code>	Pointer to the context of the second summand element of the $GF(p^d)$ field.
<code>pGFpxElementR</code>	Pointer to the context of the resulting element of the $GF(p^d)$ field.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the sum of the two elements of the $GF(p^d)$ field. The following pseudocode represents this operation: $R = A + B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpXState</code> or <code>IppsGFpXElement</code> context parameters does not match the operation.

GFPXAdd_GFP

Adds elements of the $GF(p^d)$ and $GF(p)$ fields.

Syntax

```
IppStatus ippsgfpxAdd_GFP(const IppsGFpXElement *pGFpxElementA, const IppsGFpElement *pGFpElementB, IppsGFpXElement *pGFpxElementR, IppsGFpXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the summand element of the $GF(p^d)$ field.
<code>pGFpElementB</code>	Pointer to the context of the summand element of the $GF(p)$ field.
<code>pGFpxElementR</code>	Pointer to the context of the resulting element of the $GF(p^d)$ field.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the sum of the elements of the $GF(p^d)$ and $GF(p)$ fields. The following pseudocode represents this operation: $R = A + B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> , <code>IppsGFPXElement</code> , or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPXSub

Subtracts one element of the $GF(p^d)$ field from another.

Syntax

```
IppStatus ippGFPXSub(const IppsGFPXElement *pGFpxElementA, const IppsGFPXElement
*pGFpxElementB, IppsGFPXElement *pGFpxElementR, IppsGFPXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the minuend element of the $GF(p^d)$ field.
<code>pGFpxElementB</code>	Pointer to the context of the subtrahend element of the $GF(p^d)$ field.
<code>pGFpxElementR</code>	Pointer to the context of the resulting element of the $GF(p^d)$ field.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the difference of the two elements of the $GF(p^d)$ field. The following pseudocode represents this operation: $R = A - B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.

GFPXSub_GFP

Subtracts an element of the $GF(p)$ field from an element of the $GF(p^d)$ field.

Syntax

```
IppStatus ippGFPXSub_GFP(const IppsGFPXElement *pGFpxElementA, const IppsGFPElement
*pGFpElementB, IppsGFPXElement *pGFpxElementR, IppsGFPXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the minuend element of the $GF(p^d)$ field.
<code>pGFpElementB</code>	Pointer to the context of the subtrahend element of the $GF(p)$ field.
<code>pGFpxElementR</code>	Pointer to the context of the resulting element of the $GF(p^d)$ field.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the difference of the elements of the $GF(p^d)$ and $GF(p)$ fields. The following pseudocode represents this operation: $R = A - B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFpXState</code> , <code>IppsGFpXElement</code> , or <code>IppsGFpElement</code> context parameters does not match the operation.

GFPXMul

Multiplies two elements of the $GF(p^d)$ field.

Syntax

```
IppStatus ippsgfpxmul(const IppsGFpXElement *pGFpxElementA, const IppsGFpXElement
*pGFpxElementB, IppsGFpXElement *pGFpxElementR, IppsGFpXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the first multiplicand element of the $GF(p^d)$ field.
<code>pGFpxElementB</code>	Pointer to the context of the second multiplicand element of the $GF(p^d)$ field.
<code>pGFpxElementR</code>	Pointer to the context of the resulting element of the $GF(p^d)$ field.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the product of the two elements of the $GF(p^d)$ field. The following pseudocode represents this operation: $R = A * B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.

GFPXMul_GFP

Multiplies an element of the $GF(p^d)$ field by an element of the $GF(p)$ field.

Syntax

```
IppStatus ippGFPXMul_GFP(const IppsGFPXElement *pGFpxElementA, const IppsGFPElement
*pGFpElementB, IppsGFPXElement *pGFpxElementR, IppsGFPXState *pGFpx);
```

Parameters

<code>pGFpxElementA</code>	Pointer to the context of the first multiplicand element of the $GF(p^d)$ field.
<code>pGFpElementB</code>	Pointer to the context of the second multiplicand element of the $GF(p)$ field.
<code>pGFpxElementR</code>	Pointer to the context of the resulting element of the $GF(p^d)$ field.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the product of the elements of the $GF(p^d)$ and $GF(p)$ fields. The following pseudocode represents this operation: $R = A * B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> , <code>IppsGFPXElement</code> , or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPXExp

Raises an element of the $GF(p^d)$ field to a non-negative power.

Syntax

```
IppStatus ippGFPXExp(const IppsGFPXElement *pGFpxElementA, const Ipp32u *pExp, Ipp32u
expLen, IppsGFPXElement *pGFpxElementR, IppsGFPXState *pGFpx);
```

Parameters

<i>pGFpxElementA</i>	Pointer to the context of the $GF(p^d)$ element that represents the base of the exponentiation.
<i>pExp</i>	Pointer to the data array that stores the exponent.
<i>expLen</i>	Length of the exponent.
<i>pGFpxElementR</i>	Pointer to the context of the resulting element of the $GF(p^d)$ field.
<i>pGFpx</i>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. The function raises the element of the $GF(p^d)$ field to the given non-negative power. The following pseudocode represents this operation: $R = A^E$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.

GFPXDiv

Performs the Euclidean division of two elements of the $GF(p^d)$ field.

Syntax

```
IppStatus ippGFPXDiv(const IppsGFPXElement *pGFpxElementA, const IppsGFPXElement
*pGFpxElementB, IppsGFPXElement *pGFpxElementQ, IppsGFPXElement *pGFpxElementR,
IppsGFPXState *pGFpx);
```

Parameters

<i>pGFpxElementA</i>	Pointer to the context of the dividend element of the $GF(p^d)$ field.
----------------------	--

<code>pGFpxElementB</code>	Pointer to the context of the divisor element of the $GF(p^d)$ field.
<code>pGFpxElementQ</code>	Pointer to the context of the resulting quotient element of the $GF(p^d)$ field.
<code>pGFpxElementR</code>	Pointer to the context of the resulting remainder element of the $GF(p^d)$ field.
<code>pGFpx</code>	Pointer to the context of the $GF(p^d)$ field.

Description

This function is declared in the `ippcp.h` file. For the two given elements of the $GF(p^d)$ field, the function computes the unique pair of the quotient and remainder. The following pseudocode represents this operation:
 $A = B * Q + R$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXState</code> or <code>IppsGFPXElement</code> context parameters does not match the operation.
<code>ippDivByZeroErr</code>	Indicates an error condition if the zero divisor is attempted.

Functions for the $GF(p^{d^2})$ Field

GFPXQGetSize

Gets the size of the `IppsGFPXQState` context of the finite field $GF(p^{d^2})$.

Syntax

```
IppStatus ippGFPXQGetSize(const IppsGFPXState *pGroundGFpx, Ipp32u *stateSizeInBytes);
```

Parameters

<code>pGroundGFpx</code>	Pointer to the ground finite field $GF(p^d)$.
<code>stateSizeInBytes</code>	Buffer size in bytes needed for the <code>IppsGFPXQState</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPXQState` context, suitable for storing data for the $GF(p^{d^2})$ field, that is, the quadratic extension of the finite field $GF(p^d)$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the <code>IppsGFPXState</code> context parameter does not match the operation.

GFPXQInit

Initializes the `IppsGFPXQState` context of the finite field $GF(p^{d^2})$.

Syntax

```
IppStatus ippGFPXQInit(IppsGFPXQState *pGFpxq, IppsGFPXState *pGroundGFpx, const Ipp32u *pIrrPoly);
```

Parameters

<code>pGFpxq</code>	Pointer to the context of the $GF(p^{d^2})$ field being initialized.
<code>pGroundGFpx</code>	Pointer to the context of the ground field $GF(p^d)$.
<code>pIrrPoly</code>	Pointer to the array with coefficients of the irreducible polynomial.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory buffer `*pGFpxq` associated with the `IppsGFPXQState` context and sets up the specific irreducible polynomial. The initialized context is used in the functions that create contexts of elements of the $GF(p^{d^2})$ field and perform operations with the field elements.



NOTE. The function does not check primality of the polynomial.



Important. While you are calling the functions over the $GF(p^{d^2})$ field, properly initialized `IppsGFPState` and `IppsGFPXState` contexts of the respective ground fields $GF(p)$ and $GF(p^d)$ are required.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if the <code>IppsGFPXState</code> context parameter does not match the operation.

GFPXQGet

Extracts parameters of the finite field $GF(p^{d^2})$ from the input structure.

Syntax

```
IppStatus ippGFPXQGet(const IppsGFPXQState *pGFpxq, const Ipp32u **ppGroundGFpx, Ipp32u *pIrrPoly, Ipp32u *polyDegree, Ipp32u *polyLen, Ipp32u *elementLen);
```

Parameters

<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.
<i>pGroundGFpx</i>	Pointer to the ground field $GF(p^d)$.
<i>pIrrPoly</i>	Pointer to the array with coefficients of the irreducible polynomial.
<i>polyDegree</i>	The degree of the polynomial.
<i>polyLen</i>	Length of the array with the coefficients of the polynomial.
<i>elementLen</i>	Length of a $GF(p^{d^2})$ element.

Description

This function is declared in the `ippcp.h` file. The function extracts parameters of the $GF(p^{d^2})$ field from the `IppsGFPXQState` context. You can get any combination of the following field parameters: a reference to the ground field $GF(p^d)$, a copy of the irreducible polynomial, its degree and length, as well as the length of an element of the $GF(p^{d^2})$ field. To turn off extraction of a particular field parameter, set the appropriate function parameter to `NULL`.

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 <i>pGFpxq</i> pointer is <code>NULL</code> .

GFPXQElementGetSize

Gets the size of the `IppsGFPXQElement` context of a $GF(p^{d^2})$ element.

Syntax

```
IppStatus ippGFPXQElementGetSize(const IppsGFPXQState *pGFpxq, Ipp32u *stateSizeInBytes);
```

Parameters

<i>pGFpxq</i>	Pointer to the <code>IppsGFPXQState</code> context of the $GF(p^{d^2})$ field.
<i>stateSizeInBytes</i>	Buffer size in bytes needed for the <code>IppsGFPXQElement</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPXQElement` context, suitable for storing an element of the finite field $GF(p^{d^2})$.

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 <code>NULL</code> .

GFPXQElementInit

Initializes the `IppsGFPXQElement` context of a $GF(p^{d^2})$ element.

Syntax

```
IppStatus ippGFPXQElementInit(IppsGFPXQElement *pGFpxqElement, const Ipp32u *pData, Ipp32u dataLen, IppsGFPXQState *pGFpxq);
```

Parameters

<code>pGFpxqElement</code>	Pointer to the context of the $GF(p^{d^2})$ element being initialized.
<code>pData</code>	Pointer to the data array storing the $GF(p^{d^2})$ element.
<code>dataLen</code>	Length of the element.
<code>pGFpxq</code>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory buffer `*pGFpxqElement` associated with the `IppsGFPXQElement` context and sets up the specific element of the $GF(p^{d^2})$ field. The initialized `IppsGFPXQElement` context is used in all the operations with this element of the $GF(p^{d^2})$ 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 <code>pGFpxqElement</code> or <code>pGFpxq</code> pointers is <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if the <code>IppsGFPXQState</code> context parameters does not match the operation.

GFPXQSetElement

Assigns a value to an element of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippGFPXQSetElement(const Ipp32u *pData, Ipp32u dataLen, IppsGFPXQElement *pGFpxqElement, IppsGFPXQState *pGFpxq);
```

Parameters

<code>pData</code>	Pointer to the data array storing the element of the $GF(p^{d^2})$ field.
<code>dataLen</code>	Length of the $GF(p^{d^2})$ element.
<code>pGFpxqElement</code>	Pointer to the context of the $GF(p^{d^2})$ element.
<code>pGFpxq</code>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function copies (and converts if needed) the value from the user-defined `*pData` buffer to the `IppsGFPXQElement` context of the $GF(p^{d^2})$ element.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQSetElementZero

Assigns the zero value to an element of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippGFPXQSetElementZero(IppsGFPXQElement *pGFpxqElement, IppsGFPXQState *pGFpxq);
```

Parameters

<code>pGFpxqElement</code>	Pointer to the context of the $GF(p^{d^2})$ element.
<code>pGFpxq</code>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function assigns the zero value to the given element of the $GF(p^{d^2})$ 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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQSetElementPowerX

Assigns the value of a given power of x to an element of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippGFPXQSetElementPowerX(Ipp32u power, IppsGFPXQElement *pGFpxqElement,
IppsGFPXQState *pGFpxq);
```

Parameters

<i>power</i>	The exponent.
<i>pGFpxqElement</i>	Pointer to the context of the $GF(p^{d^2})$ element.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function sets (and converts if needed) the value of the given element of the $GF(p^{d^2})$ field to x^{power} .

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQSetElementRandom

Assigns a random value to an element of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippGFPXQSetElementRandom(IppsGFPXQElement *pGFpxqElement, IppsGFPXQState *pGFpxq,
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>pGFpxqElement</i>	Pointer to the context of the $GF(p^{d^2})$ element.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.
<i>rndFunc</i>	Function that generates random sequences.
<i>pRndParam</i>	Pointer to the context of a random sequence generator.

Description

This function is declared in the `ippcp.h` file. The function assigns a random value to the given element of the $GF(p^{d^2})$ 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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

See Also

- [Pseudorandom Number Generation Functions](#)

GFPXQCmpElement

Compares two elements of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippsgFPXQCmpElement(const IppsGFPXQElement *pGFpxqElementA, const IppsGFPXQElement
*pGFpxqElementB, IppsElementCmpResult *cmpResult, const IppsGFPXQState *pGFpxq);
```

Parameters

<code>pGFpxqElementA</code>	Pointer to the context of the first element of the $GF(p^{d^2})$ field.
<code>pGFpxqElementB</code>	Pointer to the context of the second element of the $GF(p^{d^2})$ field.
<code>cmpResult</code>	Result of the comparison. For details, see comparison functions .
<code>pGFpx</code>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function compares two elements of the $GF(p^{d^2})$ field and returns the result in `*cmpResult`.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQCpyElement

Copies one element of the $GF(p^{d^2})$ field to another.

Syntax

```
IppStatus ippsgFPXQCpyElement(const IppsGFPXQElement *pGFpxqElementA, IppsGFPXQElement
*pGFpxqElementB, const IppsGFPXQState *pGFpxq);
```

Parameters

<i>pGFpxqElementA</i>	Pointer to the context of the $GF(p^{d^2})$ element being copied.
<i>pGFpxqElementB</i>	Pointer to the context of the $GF(p^{d^2})$ element being changed.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function copies one element of the $GF(p^{d^2})$ field to another.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQGetElement

Extracts the element of the $GF(p^{d^2})$ field from an input `IppsGFPXQElement` context.

Syntax

```
IppStatus ippGFPXQGetElement(const IppsGFPXQElement *pGFpxqElement, Ipp32u *pData, Ipp32u dataLen, const IppsGFPXQState *pGFpxq);
```

Parameters

<i>pGFpxqElement</i>	Pointer to the context of the $GF(p^{d^2})$ element.
<i>pData</i>	Pointer to the data array to copy the $GF(p^{d^2})$ element from.
<i>dataLen</i>	Length of the data array.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function copies the element of the $GF(p^{d^2})$ field from the `IppsGFPXQElement` context to the user-defined `*pData` buffer.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQNeg

Computes the additive inverse for an element of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippsGFPXQNeg(const IppsGFPXQElement *pGFpxqElementA, IppsGFPXQElement
*pGFpxqElementR, IppsGFPXQState *pGFpxq);
```

Parameters

<i>pGFpxqElementA</i>	Pointer to the context of the given $GF(p^{d^2})$ element.
<i>pGFpxqElementR</i>	Pointer to the context of the resulting $GF(p^{d^2})$ element.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the additive inverse for a given element of the $GF(p^{d^2})$ field. The following pseudocode represents this operation: $R = 0 - A$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQInv

Computes the multiplicative inverse for an element of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippsGFPXQInv(const IppsGFPXQElement *pGFpxqElementA, IppsGFPXQElement
*pGFpxqElementR, IppsGFPXQState *pGFpxq);
```

Parameters

<i>pGFpxqElementA</i>	Pointer to the context of the given $GF(p^{d^2})$ element.
<i>pGFpxqElementR</i>	Pointer to the context of the resulting $GF(p^{d^2})$ element.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the multiplicative inverse for a given element of the $GF(p^{d^2})$ field. The following pseudocode represents this operation: $R = A^{(-1)}$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.
<code>ippStsDivByZeroErr</code>	Indicates an error condition if the zero element is attempted.

GFPXQAdd

Adds two elements of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippGFPXQAdd(const IppsGFPXQElement *pGFpxqElementA, const IppsGFPXQElement
*pGFpxqElementB, IppsGFPXQElement *pGFpxqElementR, IppsGFPXQState *pGFpxq);
```

Parameters

<code>pGFpxqElementA</code>	Pointer to the context of the first summand element of the $GF(p^{d^2})$ field.
<code>pGFpxqElementB</code>	Pointer to the context of the second summand element of the $GF(p^{d^2})$ field.
<code>pGFpxqElementR</code>	Pointer to the context of the resulting element of the $GF(p^{d^2})$ field.
<code>pGFpxq</code>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the sum of the two elements of the $GF(p^{d^2})$ field. The following pseudocode represents this operation: $R = A + B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQSub

Subtracts one element of the $GF(p^{d^2})$ field from another.

Syntax

```
IppStatus ippsGFPXQSub(const IppsGFPXQElement *pGFpxqElementA, const IppsGFPXQElement
*pGFpxqElementB, IppsGFPXQElement *pGFpxqElementR, IppsGFPXQState *pGFpxq);
```

Parameters

<i>pGFpxqElementA</i>	Pointer to the context of the minuend element of the $GF(p^{d^2})$ field.
<i>pGFpxqElementB</i>	Pointer to the context of the subtrahend element of the $GF(p^{d^2})$ field.
<i>pGFpxqElementR</i>	Pointer to the context of the resulting element of the $GF(p^{d^2})$ field.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the difference of the two elements of the $GF(p^{d^2})$ field. The following pseudocode represents this operation: $R = A - B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQMul

Multiplies two elements of the $GF(p^{d^2})$ field.

Syntax

```
IppStatus ippsGFPXQMul(const IppsGFPXQElement *pGFpxqElementA, const IppsGFPXQElement
*pGFpxqElementB, IppsGFPXQElement *pGFpxqElementR, IppsGFPXQState *pGFpxq);
```

Parameters

<i>pGFpxqElementA</i>	Pointer to the context of the first multiplicand element of the $GF(p^{d^2})$ field.
<i>pGFpxqElementB</i>	Pointer to the context of the second multiplicand element of the $GF(p^{d^2})$ field.

<i>pGFpxqElementR</i>	Pointer to the context of the resulting element of the $GF(p^{d^2})$ field.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the product of the two elements of the $GF(p^{d^2})$ field. The following pseudocode represents this operation: $R = A * B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

GFPXQMul_GFP

Multiplies an element of the $GF(p^{d^2})$ field by an element of the $GF(p)$ field.

Syntax

```
IppStatus ippGFPXQMul_GFP(const IppsGFPXQElement *pGFpxqElementA, const IppsGFPElement *pGFpElementB, IppsGFPXQElement *pGFpxqElementR, IppsGFPXQState *pGFpxq);
```

Parameters

<i>pGFpxqElementA</i>	Pointer to the context of the first multiplicand element of the $GF(p^{d^2})$ field.
<i>pGFpElementB</i>	Pointer to the context of the second multiplicand element of the $GF(p)$ field.
<i>pGFpxqElementR</i>	Pointer to the context of the resulting element of the $GF(p^{d^2})$ field.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function computes the product of the elements of the $GF(p^{d^2})$ and $GF(p)$ fields. The following pseudocode represents this operation: $R = A * B$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> , <code>IppsGFPXQElement</code> , or <code>IppsGFPElement</code> context parameters does not match the operation.

GFPXQExp

Raises an element of the $GF(p^{d^2})$ field to a non-negative power.

Syntax

```
IppStatus ippsGFPXQExp(const IppsGFPXQElement *pGFpxqElementA, const Ipp32u *pExp, Ipp32u expLen, IppsGFPXQElement *pGFpxqElementR, IppsGFPXQState *pGFpxq);
```

Parameters

<i>pGFpxqElementA</i>	Pointer to the context of the $GF(p^{d^2})$ element that represents the base of the exponentiation.
<i>pExp</i>	Pointer to the data array that stores the exponent.
<i>expLen</i>	Length of the exponent.
<i>pGFpxqElementR</i>	Pointer to the context of the resulting element of the $GF(p^{d^2})$ field.
<i>pGFpxq</i>	Pointer to the context of the $GF(p^{d^2})$ field.

Description

This function is declared in the `ippcp.h` file. The function raises the element of the $GF(p^{d^2})$ field to the given non-negative power. The following pseudocode represents this operation: $R = A^E$.

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 <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the <code>IppsGFPXQState</code> or <code>IppsGFPXQElement</code> context parameters does not match the operation.

Arithmetic of the Group of Elliptic Curve Points

This section describes the Intel IPP primitives that implement arithmetic operations with points of elliptic curves [EC]. Arithmetic of elliptic curves over the following finite fields is implemented:

$GF(p)$	The elliptic curve is defined by the equation: $y^2 = x^3 + Ax + B \pmod{p},$ where the odd prime number p is the modulus of the $GF(p)$ field.
$GF(p^d)$	The elliptic curve is defined by the equation: $y^2 = x^3 + Ax + B \pmod{g(t)},$ where $g(t)$ is the irreducible polynomial of degree d .

Here A , B , x , and y belong to the respective field $GF(p)$ or $GF(p^d)$,

A and B are the coefficients (parameters) of the curve,

x and y are coordinates of a point on the curve.

Table “Intel IPP Arithmetic Functions for the Group of Elliptic Curve Points” lists all the functions for the elliptic curve point arithmetic.

Intel IPP Arithmetic Functions for the Group of Elliptic Curve Points

Function Base Name	Operation
Arithmetic of Elliptic Curves over GF(p)	
GFPECGetSize	Gets the size of the context of an elliptic curve over the GF(p) field.
GFPECInit	Initializes the context of an elliptic curve over the GF(p) field.
GFPECSets	Sets up parameters of an elliptic curve over the GF(p) field.
GFPECGet	Extracts parameters of the elliptic curve over the GF(p) field from the context.
GFPECVerify	Verifies parameters of an elliptic curve over the GF(p) field.
GFPECPointGetSize	Gets the size of the context of a point on the elliptic curve over the GF(p) field.
GFPECPointInit	Initializes the context of a point on the elliptic curve over the GF(p) field.
GFPECSetsPoint	Sets up the coordinates of a point on the elliptic curve over the GF(p) field.
GFPECSetsPointAtInfinity	Sets the coordinates of a point on the elliptic curve over the GF(p) field to those of the point at infinity.
GFPECSetsPointRandom	Sets random coordinates of a point on the elliptic curve over the GF(p) field.
GFPECCpyPoint	Copies one point of the elliptic curve over the GF(p) field to another.
GFPECGetPoint	Extracts coordinates of the point on the elliptic curve over the GF(p) field from the context.
GFPECVerifyPoint	Verifies a point of the elliptic curve over the GF(p) field.
GFPECCmpPoint	Compares points of the elliptic curve over the GF(p) field.
GFPECNegPoint	Computes the inverse for a point of the elliptic curve over the GF(p) field.
GFPECAAddPoint	Adds points on the elliptic curve over the GF(p) field.
GFPECMulPointScalar	Multiplies a point on the elliptic curve over the GF(p) field by a scalar.
Arithmetic of Elliptic Curves over GF(p^d)	
GFPXECGetSize	Gets the size of the context of an elliptic curve over the GF(p^d) field.
GFPXECInit	Initializes the context of an elliptic curve over the GF(p^d) field.
GFPXECSet	Sets up parameters of an elliptic curve over the GF(p^d) field.
GFPXECGet	Extracts parameters of the elliptic curve over the GF(p^d) field from the context.
GFPXECVerify	Verifies parameters of an elliptic curve over the GF(p^d) field.
GFPXECPointGetSize	Gets the size of the context of a point on the elliptic curve over the GF(p^d) field.
GFPXECPointInit	Initializes the context of a point on the elliptic curve over the GF(p^d) field.
GFPXECSetPoint	Sets up the coordinates of a point on the elliptic curve over the GF(p^d) field.
GFPXECSetPointAtInfinity	Sets the coordinates of a point on the elliptic curve over the GF(p^d) field to those of the point at infinity.

Function Base Name	Operation
GFPXECSetPointRandom	Sets random coordinates of a point on the elliptic curve over the $GF(p^d)$ field.
GFPXECCpyPoint	Copies one point of the elliptic curve over the $GF(p^d)$ field to another.
GFPXECGetPoint	Extracts coordinates of the point on the elliptic curve over the $GF(p^d)$ field from the context.
GFPXECVerifyPoint	Verifies a point of the elliptic curve over the $GF(p^d)$ field.
GFPXECCmpPoint	Compares points of the elliptic curve over the $GF(p^d)$ field.
GFPXECNegPoint	Computes the inverse for a point of the elliptic curve over the $GF(p^d)$ field.
GFPXECAddPoint	Adds points on the elliptic curve over the $GF(p^d)$ field.
GFPXECMulPointScalar	Multiplies a point on the elliptic curve over the $GF(p^d)$ field by a scalar.

The functions described in this section use context structures of the following types to carry data of the elliptic curve and elliptic curve point:

Elliptic curve over $GF(p)$	<code>IppsGFPECState</code> and <code>IppsGFPECPPoint</code> , respectively.
Elliptic curve over $GF(p^d)$	<code>IppsGFPXECState</code> and <code>IppsGFPXECPoint</code> , respectively.

See Also

- [Finite Field Arithmetic](#)

Functions for the Elliptic Curve over $GF(p)$

GFPECGetSize

Gets the size of the `IppsGFPECState` context of the elliptic curve over the $GF(p)$ field.

Syntax

```
IppStatus ippsGFPECGetSize(const IppsGFpState *pGFp, Ipp32u *stateSizeInBytes);
```

Parameters

<code>pGFp</code>	Pointer to the <code>IppsGFpState</code> context of the finite field $GF(p)$.
<code>stateSizeInBytes</code>	Buffer size in bytes needed for the <code>IppsGFPECState</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPECState` context, suitable for storing data for the elliptic curve over the $GF(p)$ 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 <code>NULL</code> .

`ippStsContextMatchErr`

Indicates an error condition if any of the context parameters does not match the operation.

GFPECEInit

Initializes the `IppsGFPECEState` context of the elliptic curve over the $GF(p)$ field.

Syntax

```
IppStatus ippGFPECEInit(IppsGFPECEState *pEC, const IppsGFPECElement *pA, const IppsGFPECElement *pB, const IppsGFPECElement *pX, const IppsGFPECElement *pY, const Ipp32u *pOrder, Ipp32u orderLen, Ipp32u cofactor, IppsGFPEState *pGFp);
```

Parameters

<code>pEC</code>	Pointer to the context of the elliptic curve being initialized.
<code>pA</code>	Pointer to the A parameter of the elliptic curve.
<code>pB</code>	Pointer to the B parameter of the elliptic curve.
<code>pX, pY</code>	Pointers to the x and y coordinates of the base point of the elliptic curve.
<code>pOrder</code>	Pointer to the array storing the order of the base point.
<code>orderLen</code>	Length of the array storing the order of the base point.
<code>cofactor</code>	The value of the cofactor.
<code>pGFp</code>	Pointer to the context of the elliptic curve definition field $GF(p)$.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory buffer `*pEC` associated with the `IppsGFPECEState` context and sets up the specific parameters of the elliptic curve, if they are supplied. The initialized context is used in the functions that create contexts of points on the curve (elements of the group of points) and perform operations with the points.



NOTE. Only the `pEC` and `pGFp` parameters are required. You can omit the other parameters by setting their values to `NULL` or zero and set up the missing parameters of the elliptic curve later on by calling `GFPECESet`.



Important. While you are calling the arithmetic functions for the elliptic curve `*pEC`, a properly initialized `*pGFp` context of the definition field $GF(p)$ is required.

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 <code>pEC</code> or <code>pGFp</code> pointers is <code>NULL</code> .

`ippStsContextMatchErr` Indicates an error condition if any of the context parameters does not match the operation.

GFPECCSet

Sets up parameters of the elliptic curve in the `IppsGFPECCState` context.

Syntax

```
IppStatus ippGFPECCSet(const IppsGFPElement *pA, const IppsGFPElement *pB, const
IppsGFPElement *pX, const IppsGFPElement *pY, const Ipp32u *pOrder, Ipp32u orderLen, Ipp32u
cofactor, IppsGFPECCState *pEC);
```

Parameters

<code>pA</code>	Pointer to the A parameter of the elliptic curve.
<code>pB</code>	Pointer to the B parameter of the elliptic curve.
<code>pX, pY</code>	Pointers to the x and y coordinates of the base point of the elliptic curve.
<code>pOrder</code>	Pointer to the array storing the order of the base point.
<code>orderLen</code>	Length of the array storing the order of the base point.
<code>cofactor</code>	The value of the cofactor.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function assigns values to the parameters of the elliptic curve in the `IppsGFPECCState` context.

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 <code>pEC</code> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECCGet

Extracts elliptic curve parameters from the input `IppsGFPECCState` context.

Syntax

```
IppStatus ippGFPECCGet(const IppsGFPECCState *pEC, const IppsGFPEState **pGFp, Ipp32u
*pElementLen, IppsGFPElement *pA, IppsGFPElement *pB, IppsGFPElement *pX, IppsGFPElement
*pY, const Ipp32u **pOrder, Ipp32u *orderLen, Ipp32u *cofactor);
```

Parameters

<i>pEC</i>	Pointer to the context of the elliptic curve.
<i>pGFp</i>	Pointer to the context of the elliptic curve definition field $GF(p)$.
<i>pA</i>	Pointer to a copy of the <i>A</i> parameter of the elliptic curve.
<i>pB</i>	Pointer to a copy of the <i>B</i> parameter of the elliptic curve.
<i>pX</i> , <i>pY</i>	Pointers to copies of the <i>x</i> and <i>y</i> coordinates of the base point of the elliptic curve.
<i>pOrder</i>	Address of the pointer to the array storing the order of the base point.
<i>orderLen</i>	Length of the array storing the order of the base point.
<i>cofactor</i>	The value of the cofactor.

Description

This function is declared in the `ippcp.h` file. The function extracts parameters of the elliptic curve from the input `IppsGFPECState` context. You can get any combination of the following parameters: a reference to the definition field, copies of the *A* and *B* coefficients and *x* and *y* coordinates, a reference to the order of the base point, the length of the order, and the value of the cofactor. To turn off extraction of a particular parameter of the elliptic curve, set the appropriate function parameter to `NULL`.

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 <i>pEC</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECVerify

Verifies parameters of the elliptic curve over the $GF(p)$ field.

Syntax

```
IppStatus ippGFPECVerify(IppsGFPECState *pEC, IppsECResult *result);
```

Parameters

<i>pEC</i>	Pointer to the context of the elliptic curve.
<i>result</i>	The result of the parameter verification.

Description

This function is declared in the `ippcp.h` file. The function verifies parameters of the elliptic curve from the input `IppsGFPECState` context and returns the result in **result*. The result of the verification may have the following values:

<code>ippECValid</code>	Parameters are valid.
-------------------------	-----------------------

<code>ippECIsZeroDiscriminant</code>	$4*A^3+3*B^2=0 \bmod p$.
<code>ippECPointIsAtInfinity</code>	Base point $G=(x,y)$ is the point at infinity.
<code>ippECPointIsNotValid</code>	Base point $G=(x,y)$ is not on the curve.
<code>ippECInvalidOrder</code>	The order of the base point $G=(x,y)$ is invalid.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECPPointGetSize

Gets the size of the `IppsGFPECPPoint` context of a point on the elliptic curve.

Syntax

```
IppStatus ippGFPECPPointGetSize(const IppsGFPECPState *pEC, Ipp32u *stateSizeInBytes);
```

Parameters

<code>pEC</code>	Pointer to the context of the elliptic curve.
<code>stateSizeInBytes</code>	Buffer size in bytes needed for the <code>IppsGFPECPPoint</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPECPPoint` context, suitable for storing data for a point on the elliptic curve over the $GF(p)$ 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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECPPointInit

Initializes the `IppsGFPECPPoint` context of a point on the elliptic curve.

Syntax

```
IppStatus ippGFPECPPointInit(IppsGFPECPPoint *pPoint, const IppsGFPElement *pX, const IppsGFPElement *pY, IppsGFPECPState *pEC);
```

Parameters

<i>pPoint</i>	Pointer to the <code>IppsGFPECPPoint</code> context being initialized.
<i>pX</i> , <i>pY</i>	Pointers to the <i>x</i> and <i>y</i> coordinates of the point on the elliptic curve.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function initializes the `IppsGFPECPPoint` context and sets up the specific coordinates of the elliptic curve point.



NOTE. Setting the *pX* and *pY* pointers to `NULL` initializes the context for the point at infinity.

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 <i>pEC</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECSetPoint

Sets up the coordinates of a point on the elliptic curve over the GF(p) field.

Syntax

```
IppStatus ippGFPECSetPoint(const IppsGFPElement *pX, const IppsGFPElement *pY,
IppsGFPECPPoint *pPoint, IppsGFPECPState *pEC);
```

Parameters

<i>pX</i> , <i>pY</i>	Pointers to the <i>x</i> and <i>y</i> coordinates of the point on the elliptic curve.
<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function assigns the given values to the coordinates of the elliptic curve point in the `IppsGFPECPPoint` context.

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 <i>pEC</i> pointer is <code>NULL</code> .

`ippStsContextMatchErr` Indicates an error condition if any of the context parameters does not match the operation.

GFPECSetsPointAtInfinity

Sets the coordinates of the elliptic curve point in the `IppsGFPECPPoint` context to those of the point at infinity.

Syntax

```
IppStatus ippGFPECSetsPointAtInfinity(IppsGFPECPPoint *pPoint, IppsGFPECSState *pEC);
```

Parameters

`pPoint` Pointer to the context of the elliptic curve point.
`pEC` Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function sets the coordinates of the elliptic curve point in the `IppsGFPECPPoint` context to the coordinates of the point at infinity.

Return Values

`ippStsNoErr` Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr` Indicates an error condition if the `pEC` pointer is `NULL`.
`ippStsContextMatchErr` Indicates an error condition if any of the context parameters does not match the operation.

GFPECSetsPointRandom

Sets random coordinates of a point on the elliptic curve over the $GF(p)$ field.

Syntax

```
IppStatus ippGFPECSetsPointRandom(IppsGFPECPPoint *pPoint, IppsGFPECSState *pEC,
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

`pPoint` Pointer to the context of the elliptic curve point.
`pEC` Pointer to the context of the elliptic curve.
`rndFunc` Function that generates random sequences.
`pRndParam` Pointer to the context of a random sequence generator.

Description

This function is declared in the `ippcp.h` file. The function assigns random values to the coordinates of the elliptic curve point in the `IppsGFPECPPoint` context.

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 <code>pEC</code> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

See Also

- [Pseudorandom Number Generation Functions](#)

GFPECCpyPoint

Copies one point of the elliptic curve over the GF(p) field to another.

Syntax

```
IppStatus ippGFPECCpyPoint(const IppsGFPECPPoint *pA, IppsGFPECPPoint *pR, IppsGFPECState *pEC);
```

Parameters

<code>pA</code>	Pointer to the context of the elliptic curve point being copied.
<code>pR</code>	Pointer to the context of the elliptic curve point being changed.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function copies one point of the elliptic curve over the GF(p) field to another.

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 <code>pEC</code> pointer is <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECCGetPoint

Extracts coordinates of the point on the elliptic curve from the IppsGFPECPPoint context.

Syntax

```
IppStatus ippGFPECCGetPoint(const IppsGFPECPPoint *pPoint, IppsGFPEElement *pX, IppsGFPEElement *pY, IppsGFPECState *pEC);
```


Parameters

<code>pPoint</code>	Pointer to the context of the elliptic curve point.
<code>pX, pY</code>	Pointers to the x and y coordinates of the point on the elliptic curve.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function exports the coordinates of the elliptic curve point from the `IppsGFPECPPoint` context to the user-defined elements of the definition field. To turn off extraction of a particular coordinate, set the appropriate function parameter to `NULL`.

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 <code>pEC</code> or <code>pPoint</code> pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECVVerifyPoint

Verifies whether the point belongs to the elliptic curve over the $GF(p)$ field.

Syntax

```
IppStatus ippGFPECVVerifyPoint(const IppsGFPECPPoint *pPoint, IppsECResult *result,
IppsGFPECPState *pEC);
```

Parameters

<code>pPoint</code>	Pointer to the context of the elliptic curve point.
<code>result</code>	The result of the verification.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function verifies whether the given point belongs to the elliptic curve over the $GF(p)$ field. The result of the verification is returned in `*result` and may have the following values:

<code>ippECValid</code>	The point belongs to the curve.
<code>ippECPointIsValid</code>	The point does not belong to the curve.

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 <code>NULL</code> .

`ippStsContextMatchErr`

Indicates an error condition if any of the context parameters does not match the operation.

GFPECCmpPoint

Compares two points on the elliptic curve over the GF(p) field.

Syntax

```
IppStatus ippGFPECCmpPoint(const IppsGFPECCPoint *pA, const IppsGFPECCPoint *pB,
IppsElementCmpResult *result, IppsGFPECCState *pEC);
```

Parameters

<code>pA</code>	Pointer to the context of the first point on the elliptic curve.
<code>pB</code>	Pointer to the context of the second point on the elliptic curve.
<code>result</code>	The result of the parameter verification.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippccp.h` file. The function compares coordinates of two points on the elliptic curve over the GF(p) field and returns the result in `*result`. The result of the comparison may have the following values:

<code>ippElementEQ</code>	The points are equal.
<code>ippElementNE</code>	The points are not equal.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECNegPoint

Computes the inverse point for a given point on the elliptic curve over the GF(p) field.

Syntax

```
IppStatus ippGFPECNegPoint(const IppsGFPECPoint *pA, IppsGFPECPoint *pR, IppsGFPECState
*pEC);
```

Parameters

<code>pA</code>	Pointer to the context of the given point on the elliptic curve.
<code>pR</code>	Pointer to the context of the resulting point on the elliptic curve.

pEC Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. For a given point of the elliptic curve over the $GF(p)$ field, the function computes the coordinates of the inverse point. The following pseudocode represents this operation: $R = 0 - A$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECAAddPoint

Adds two points on the elliptic curve over the $GF(p)$ field.

Syntax

```
IppStatus ippSGFPECAAddPoint(const IppsGFPECPPoint *pA, const IppsGFPECPPoint *pB,
IppsGFPECPPoint *pR, IppsGFPECState *pEC);
```

Parameters

<i>pA</i>	Pointer to the context of the first summand point on the elliptic curve.
<i>pB</i>	Pointer to the context of the second summand point on the elliptic curve.
<i>pR</i>	Pointer to the context of the resulting point on the elliptic curve.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function computes the coordinates of the elliptic curve point that equals the sum of the two given points. The following pseudocode represents this operation: $R = A + B$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPECMulPointScalar

Adds a point on the elliptic curve over the GF(p) field to itself multiple times.

Syntax

```
IppStatus ippsGFPECMulPointScalar(const IppsGFPECPPoint *pA, const Ipp32u *pScalar, Ipp32u scalarLen, IppsGFPECPPoint *pR, IppsGFPECState *pEC);
```

Parameters

<i>pA</i>	Pointer to the context of the given point on the elliptic curve.
<i>pScalar</i>	Pointer to the data array that stores the scalar.
<i>scalarLen</i>	Length of the scalar.
<i>pR</i>	Pointer to the context of the resulting point on the elliptic curve.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function computes the coordinates of the elliptic curve point that equals the product of the given point and the scalar. The following pseudocode represents this operation:
 $R = scalar * A.$

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

Functions for the Elliptic Curve over GF(p^d)

GFPXECGetSize

Gets the size of the IppsGFPXECState context of the elliptic curve over the GF(p) field.

Syntax

```
IppStatus ippsGFPXECGetSize(const IppsGFPXState *pGFpx, Ipp32u *stateSizeInBytes);
```

Parameters

<i>pGFpx</i>	Pointer to the <code>IppsGFPXState</code> context of the finite field GF(p^d).
<i>stateSizeInBytes</i>	Buffer size in bytes needed for the <code>IppsGFPXECState</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPXECState` context, suitable for storing data for the elliptic curve over the $GF(p^d)$ 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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECInit

Initializes the `IppsGFPXECState` context of the elliptic curve over the $GF(p^d)$ field.

Syntax

```
IppStatus ippsgfpxecinit(IppsGFPXECState *pEC, const IppsGFPXElement *pA, const
IppsGFPXElement *pB, const IppsGFPXElement *pX, const IppsGFPXElement *pY, const Ipp32u
*pOrder, Ipp32u orderLen, Ipp32u cofactor, IppsGFPXState *pGFpx);
```

Parameters

<code>pEC</code>	Pointer to the context of the elliptic curve being initialized.
<code>pA</code>	Pointer to the A parameter of the elliptic curve.
<code>pB</code>	Pointer to the B parameter of the elliptic curve.
<code>pX, pY</code>	Pointers to the x and y coordinates of the base point of the elliptic curve.
<code>pOrder</code>	Pointer to the array storing the order of the base point.
<code>orderLen</code>	Length of the array storing the order of the base point.
<code>cofactor</code>	The value of the cofactor.
<code>pGFpx</code>	Pointer to the context of the elliptic curve definition field $GF(p^d)$.

Description

This function is declared in the `ippcp.h` file. The function initializes the memory buffer `*pEC` associated with the `IppsGFPXECState` context and sets up the specific parameters of the elliptic curve, if they are supplied. The initialized context is used in the functions that create contexts of points on the curve (elements of the group of points) and perform operations with the points.



NOTE. Only the `pEC` and `pGFpx` parameters are required. You can omit the other parameters by setting their values to `NULL` or zero and set up the missing parameters of the elliptic curve later on by calling `GFPXECSet`.



Important. While you are calling the arithmetic functions for the elliptic curve **pEC*, properly initialized *IppsGFPXState* and *IppsGFPState* contexts of the definition field $GF(p^d)$ and ground field $GF(p)$ are required.

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 <i>pEC</i> or <i>pGFPx</i> pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECSet

Sets up parameters of the elliptic curve in the IppsGFPXECState context.

Syntax

```
IppStatus ippsgfpxecset(const IppsGFPXElement *pA, const IppsGFPXElement *pB, const
IppsGFPXElement *pX, const IppsGFPXElement *pY, const Ipp32u *pOrder, Ipp32u orderLen,
Ipp32u cofactor, IppsGFPXECState *pEC);
```

Parameters

<i>pA</i>	Pointer to the <i>A</i> parameter of the elliptic curve.
<i>pB</i>	Pointer to the <i>B</i> parameter of the elliptic curve.
<i>pX</i> , <i>pY</i>	Pointers to the <i>x</i> and <i>y</i> coordinates of the base point of the elliptic curve.
<i>pOrder</i>	Pointer to the array storing the order of the base point.
<i>orderLen</i>	Length of the array storing the order of the base point.
<i>cofactor</i>	The value of the cofactor.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function assigns values to the parameters of the elliptic curve in the *IppsGFPXECState* context.

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 <i>pEC</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECGet

extracts elliptic curve parameters from the input IppsGFPXECState context.

Syntax

```
IppStatus ippsGFPXECGet(const IppsGFPXECState *pEC, const IppsGFPXState **pGFpx, Ipp32u *pElementLen, IppsGFPXElement *pA, IppsGFPXElement *pB, IppsGFPXElement *pX, IppsGFPXElement *pY, const Ipp32u **pOrder, Ipp32u *orderLen, Ipp32u *cofactor);
```

Parameters

<i>pEC</i>	Pointer to the context of the elliptic curve.
<i>pGFpx</i>	Pointer to the context of the elliptic curve definition field $GF(p^d)$.
<i>pA</i>	Pointer to a copy of the <i>A</i> parameter of the elliptic curve.
<i>pB</i>	Pointer to a copy of the <i>B</i> parameter of the elliptic curve.
<i>pX</i> , <i>pY</i>	Pointers to copies of the <i>x</i> and <i>y</i> coordinates of the base point of the elliptic curve.
<i>pOrder</i>	Address of the pointer to the array storing the order of the base point.
<i>orderLen</i>	Length of the array storing the order of the base point.
<i>cofactor</i>	The value of the cofactor.

Description

This function is declared in the `ippcp.h` file. The function extracts parameters of the elliptic curve from the input `IppsGFPXECState` context. You can get any combination of the following parameters: a reference to the definition field, copies of the *A* and *B* coefficients and the *x* and *y* coordinates, a reference to the order of the base point, the length of the order, and the value of the cofactor. To turn off extraction of a particular parameter of the elliptic curve, set the appropriate function parameter to `NULL`.

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 <i>pEC</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECVerify

Verifies parameters of the elliptic curve over the $GF(p^d)$ field.

Syntax

```
IppStatus ippsGFPXECVerify(IppsGFPXECState *pEC, IppsECResult *result);
```

Parameters

<i>pEC</i>	Pointer to the context of the elliptic curve.
<i>result</i>	The result of the parameter verification.

Description

This function is declared in the `ippcp.h` file. The function verifies parameters of the elliptic curve from the input `IppsGFPXECState` context and returns the result in **result*. The result of the verification may have the following values:

<code>ippECValid</code>	Parameters are valid.
<code>ippECIsZeroDiscriminant</code>	$4*A^3+3*B^2=0 \bmod g(t)$.
<code>ippECPointIsAtInfinity</code>	Base point $G=(x,y)$ is the point at infinity.
<code>ippECPointIsNotValid</code>	Base point $G=(x,y)$ is not on the curve.
<code>ippECInvalidOrder</code>	The order of the base point $G=(x,y)$ is invalid.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECPointGetSize

Gets the size of the IppsGFPXECPoint context of a point on the elliptic curve.

Syntax

```
ippStatus ippGFPXECPointGetSize(const IppsGFPXECState *pEC, Ipp32u *stateSizeInBytes);
```

Parameters

<i>pEC</i>	Pointer to the context of the elliptic curve.
<i>stateSizeInBytes</i>	Buffer size in bytes needed for the <code>IppsGFPXECPoint</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsGFPXECPoint` context, suitable for storing data for a point on the elliptic curve over the $GF(p^d)$ 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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECPointInit

Initializes the `IppsGFPXECPoint` context of a point on the elliptic curve.

Syntax

```
IppStatus ippsGFPXECPointInit(IppsGFPXECPoint *pPoint, const IppsGFPXElement *pX, const IppsGFPXElement *pY, IppsGFPXECState *pEC);
```

Parameters

<code>pPoint</code>	Pointer to the <code>IppsGFPXECPoint</code> context being initialized.
<code>pX</code> , <code>pY</code>	Pointers to the x and y coordinates of the point on the elliptic curve.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function initializes the `IppsGFPXECPoint` context and sets up the specific coordinates of the elliptic curve point.



NOTE. Setting the `pX` and `pY` pointers to `NULL` initializes the context for the point at infinity.

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 <code>pEC</code> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECSetPoint

Sets up the coordinates of a point on the elliptic curve over the $GF(p^d)$ field.

Syntax

```
IppStatus ippsGFPXECSetPoint(const IppsGFPXElement *pX, const IppsGFPXElement *pY, IppsGFPXECPoint *pPoint, IppsGFPXECState *pEC);
```

Parameters

<code>pX</code> , <code>pY</code>	Pointers to the x and y coordinates of the point on the elliptic curve.
<code>pPoint</code>	Pointer to the context of the elliptic curve point.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function assigns the given values to the coordinates of the elliptic curve point in the `IppsGFPXECPoint` context.

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 <code>pEC</code> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECSetPointAtInfinity

Sets the coordinates of the elliptic curve point in the `IppsGFPXECPoint` context to those of the point at infinity.

Syntax

```
IppStatus ippsgFPXECSetPointAtInfinity(IppsGFPXECPoint *pPoint, IppsGFPXECState *pEC);
```

Parameters

<code>pPoint</code>	Pointer to the context of the elliptic curve point.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function sets the coordinates of the elliptic curve point in the `IppsGFPXECPoint` context to the coordinates of the point at infinity.

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 <code>pEC</code> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECSetPointRandom

Sets random coordinates of a point on the elliptic curve over the $GF(p^d)$ field.

Syntax

```
IppStatus ippsgFPXECSetPointRandom(IppsGFPXECPoint *pPoint, IppsGFPXECState *pEC,  
IppBitSupplier rndFunc, void* pRndParam);
```

Parameters

<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>pEC</i>	Pointer to the context of the elliptic curve.
<i>rndFunc</i>	Function that generates random sequences.
<i>pRndParam</i>	Pointer to the context of a random sequence generator.

Description

This function is declared in the `ippcp.h` file. The function assigns random values to the coordinates of the elliptic curve point in the `IppsGFPXECPoint` context.

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 <i>pEC</i> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

See Also

- [Pseudorandom Number Generation Functions](#)

GFPXECpyPoint

Copies one point of the elliptic curve over the $GF(p^d)$ field to another.

Syntax

```
IppStatus ippGFPXECpyPoint(const IppsGFPXECPoint *pA, IppsGFPXECPoint *pR, IppsGFPXECState *pEC);
```

Parameters

<i>pA</i>	Pointer to the context of the elliptic curve point being copied.
<i>pR</i>	Pointer to the context of the elliptic curve point being changed.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function copies one point of the elliptic curve over the $GF(p^d)$ field to another.

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 <i>pEC</i> pointer is <code>NULL</code> .
<code>ippContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECGetPoint

Extracts coordinates of the point on the elliptic curve from the IppsGFPXECPoint context.

Syntax

```
IppStatus ippsGFPXECGetPoint(const IppsGFPXECPoint *pPoint, IppsGFPXElement *pX,
IppsGFPXElement *pY, IppsGFPXECState *pEC);
```

Parameters

<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>pX</i> , <i>pY</i>	Pointers to the <i>x</i> and <i>y</i> coordinates of the point on the elliptic curve.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function exports the coordinates of the elliptic curve point from the `IppsGFPXECPoint` context to the user-defined elements of the definition field. To turn off extraction of a particular coordinate, set the appropriate function parameter to `NULL`.

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 <i>pEC</i> or <i>pPoint</i> pointers is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECVerifyPoint

Verifies whether the point belongs to the elliptic curve over the $GF(p^d)$ field.

Syntax

```
IppStatus ippsGFPXECVerifyPoint(const IppsGFPXECPoint *pPoint, IppsECResult *result,
IppsGFPXECState *pEC);
```

Parameters

<i>pPoint</i>	Pointer to the context of the elliptic curve point.
<i>result</i>	The result of the verification.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function verifies whether the given point belongs to the elliptic curve over the $GF(p^d)$ field. The result of the verification is returned in `*result` and can have the following values:

<code>ippECValid</code>	The point belongs to the curve.
<code>ippECPointIsNotValid</code>	The point does not belong to the curve.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECmpPoint

Compares two points on the elliptic curve over the $GF(p^d)$ field.

Syntax

```
IppStatus ippGFPXECmpPoint(const IppsGFPXECPoint *pA, const IppsGFPXECPoint *pB,
IppsElementCmpResult *result, IppsGFPXECState *pEC);
```

Parameters

<code>pA</code>	Pointer to the context of the first point on the elliptic curve.
<code>pB</code>	Pointer to the context of the second point on the elliptic curve.
<code>result</code>	The result of the parameter verification.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function compares coordinates of two points on the elliptic curve over the $GF(p^d)$ field and returns the result in `*result`. The result of the comparison may have the following values:

<code>ippElementEQ</code>	The points are equal.
<code>ippElementNE</code>	The points are not equal.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECNegPoint

Computes the inverse point for a given point on the elliptic curve over the $GF(p^d)$ field.

Syntax

```
IppStatus ippsgFPXECNegPoint(const IppsgFPXECPoint *pA, IppsgFPXECPoint *pR, IppsgFPXECState *pEC);
```

Parameters

<i>pA</i>	Pointer to the context of the given point on the elliptic curve.
<i>pR</i>	Pointer to the context of the resulting point on the elliptic curve.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. For a given point of the elliptic curve over the $GF(p^d)$ field, the function computes the coordinates of the inverse point. The following pseudocode represents this operation: $R = 0 - A$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECAddPoint

Adds two points on the elliptic curve over the $GF(p^d)$ field.

Syntax

```
IppStatus ippsgFPXECAddPoint(const IppsgFPXECPoint *pA, const IppsgFPXECPoint *pB, IppsgFPXECPoint *pR, IppsgFPXECState *pEC);
```

Parameters

<i>pA</i>	Pointer to the context of the first summand point on the elliptic curve.
<i>pB</i>	Pointer to the context of the second summand point on the elliptic curve.
<i>pR</i>	Pointer to the context of the resulting point on the elliptic curve.
<i>pEC</i>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function computes the coordinates of the elliptic curve point that equals the sum of the two given points. The following pseudocode represents this operation: $R = A + B$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

GFPXECMulPointScalar

Adds a point on the elliptic curve over the $GF(p^d)$ field to itself multiple times.

Syntax

```
IppStatus ippsgFPXECMulPointScalar(const IppsGFPXECPoint *pA, const Ipp32u *pScalar, Ipp32u scalarLen, IppsGFPXECPoint *pR, IppsGFPXECState *pEC);
```

Parameters

<code>pA</code>	Pointer to the context of the given point on the elliptic curve.
<code>pScalar</code>	Pointer to the data array that stores the scalar.
<code>scalarLen</code>	Length of the scalar.
<code>pR</code>	Pointer to the context of the resulting point on the elliptic curve.
<code>pEC</code>	Pointer to the context of the elliptic curve.

Description

This function is declared in the `ippcp.h` file. The function computes the coordinates of the elliptic curve point that equals the product of the given point and the scalar. The following pseudocode represents this operation: $R = scalar * A$.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

Tate Pairing

This section describes the Intel IPP primitives for Tate pairing [EHCC].

Table “Intel IPP Tate Pairing Functions” lists all the functions for Tate pairing.

Intel IPP Tate Pairing Functions

Function Base Name	Operation
TatePairingDE3GetSize	Gets the size of the context of the pairing operation.
TatePairingDE3Init	Initializes the context of the pairing operation.
TatePairingDE3Get	Extracts parameters of the pairing operation from the context.
TatePairingDE3Apply	Performs Tate pairing.

TatePairingDE3GetSize

Gets the size of the `IppsTatePairingDE3State` context of the pairing operation.

Syntax

```
IppStatus ippsTatePairingDE3GetSize(const IppsGFPECCState *pECp, const IppsGFPXECState *pECpx, Ipp32u *stateSizeInBytes);
```

Parameters

<code>pECp</code>	Pointer to the context of the elliptic curve over the $GF(p)$ field.
<code>pECpx</code>	Pointer to the context of the elliptic curve over the $GF(p^d)$ field.
<code>stateSizeInBytes</code>	Buffer size in bytes needed for the <code>IppsTatePairingDE3State</code> context.

Description

This function is declared in the `ippcp.h` file. The function returns the size of the buffer associated with the `IppsTatePairingDE3State` context, suitable for storing data and buffers for the pairing operation.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

TatePairingDE3Init

Initializes the `IppsTatePairingDE3State` context of the pairing operation.

Syntax

```
IppStatus ippsTatePairingDE3Init(IppsTatePairingDE3State *pPairing, const IppsGFPECCState *pECp, const IppsGFPXECState *pECpx, const IppsGFPXQState *pGFpxq);
```

Parameters

<code>pPairing</code>	Pointer to the context being initialized.
<code>pECp</code>	Pointer to the context of the elliptic curve over the $GF(p)$ field.

<i>pECpx</i>	Pointer to the context of the elliptic curve over the $\text{GF}(p^d)$ field.
<i>pGFpxq</i>	Pointer to the context of the finite field $\text{GF}(p^{d^2})$.

Description

This function is declared in the `ippcp.h` file. The function initializes the *pPairing* buffer associated with the `IppsTatePairingDE3State` context, sets up and precomputes the parameters needed for the pairing operation.



Important. While the *pPairing* context exists, properly initialized contexts of the elliptic curves and the $\text{GF}(p^{d^2})$ field are required.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

TatePairingDE3Get

Extracts parameters of the pairing operation from the IppsTatePairingDE3State context.

Syntax

```
IppStatus ippstTatePairingDE3Get(const IppsTatePairingDE3State *pPairing, const
IppsGFPECEState **pECp, const IppsGFPXECState **pECpx, const IppsGFPXQState **pGFpxq);
```

Parameters

<i>pPairing</i>	Pointer to the context of the pairing operation.
<i>pECp</i>	Address of the pointer to the context of the elliptic curve over the $\text{GF}(p)$ field.
<i>pECpx</i>	Address of the pointer to the context of the elliptic curve over the $\text{GF}(p^d)$ field.
<i>pGFpxq</i>	Address of the pointer to the context of the finite field $\text{GF}(p^{d^2})$.

Description

This function is declared in the `ippcp.h` file. The function extracts parameters of the pairing operation from the input `IppsTatePairingDE3State` context. You can get any combination of the following parameters: references to the `IppsGFPECEState` and `IppsGFPXECState` contexts of the elliptic curves and a reference to the `IppsGFPXQState` context of the $\text{GF}(p^{d^2})$ field. To turn off extraction of a particular parameter of the pairing operation, set the appropriate function parameter to `NULL`.

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 <code>pPairing</code> pointer is <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the <code>IppsTatePairingDE3State</code> context parameters does not match the operation.

TatePairingDE3Apply

Performs the pairing operation.

Syntax

```
IppStatus ippstTatePairingDE3Apply(const IppsGFPECPPoint *pPointA, const IppsGFPXECPoint
*pPointB, IppsGFPXQElement *pGFpxqElementR, IppsTatePairingDE3State *pPairing);
```

Parameters

<code>pPointA</code>	Pointer to the context of the elliptic curve over the $GF(p)$ field.
<code>pPointB</code>	Pointer to the context of the elliptic curve over the $GF(p^d)$ field.
<code>pGFpxqElementR</code>	Pointer to the context of the resulting element of the finite field $GF(p^{d^2})$.
<code>pPairing</code>	Pointer to the context of the pairing operation.

Description

This function is declared in the `ippcp.h` file. The function computes the $GF(p^{d^2})$ field element that equals the pairing of elliptic curve points `*pPointA` and `*pPointB`.

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 <code>NULL</code> .
<code>ippStsContextMatchErr</code>	Indicates an error condition if any of the context parameters does not match the operation.

Support Functions and Classes

This appendix contains miscellaneous information on support functions and classes that may be helpful to users of the Intel® Integrated Performance Primitives (Intel® IPP) for cryptography.

The [Version Information Function](#) section describes an Intel IPP function that provides version information for cryptography software.

The [Classes and Functions Used in Examples](#) section presents source code of classes and functions needed for examples given in the manual chapters.

Version Information Function

GetLibVersion

Returns information about the active version of the Intel IPP software for cryptography.

Syntax

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

Description

The function `ippcpGetLibVersion` is declared in the `ippcp.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 cryptography. There is no need for you to release memory referenced by the returned pointer because it points to a static variable. The following fields of the `IppLibraryVersion` structure are available:

<code>major</code>	is the major number of the current library version.
<code>minor</code>	is the minor number of the current library version.
<code>majorBuild</code>	is the number of builds for the (<code>major.minor</code>) version.
<code>build</code>	is the total number of Intel IPP builds.
<code>Name</code>	is the name of the current library version.
<code>Version</code>	is the version string.
<code>BuildDate</code>	is the actual build date

For example, if the library version is "7.0", library name is "ippcp.lib", and build date is "Jul 20 2011", then the fields in this structure are set as follows:

```
major = 7, minor = 0, Name = "ippcp_l.lib", Version = "7.0 build 205.68", BuildDate = "Jul 20 2011".
```

Using the `ippcpGetLibVersion` Function

The code example below shows how to use the function `ippcpGetLibVersion`.

```
void libinfo(void) { const IppLibraryVersion* lib = ippcpGetLibVersion();  
printf("%s %s %d.%d.%d.%d\n", lib->Name, lib->Version, lib->major, lib->minor, lib->majorBuild, lib->build);  
}
```

Output:

```
ippcp_1.lib 7.0 build 205.68
```

Classes and Functions Used in Examples

This section presents source code of functions and classes used in [Example “The Use of RSA Primitives”](#) and [Example “Use of DLPSignDSA and DLPVerifyDSA”](#), provided in the “Public Key Cryptography Functions” chapter.

BigNumber Class

The section presents source code of the `BigNumber` class.

Declarations

Contents of the header file (`bignum.h`) declaring the `BigNumber` class are presented below:

```
#if !defined _BIGNUMBER_H_  
#define _BIGNUMBER_H_  
  
#include "ippcp.h"  
  
#include <iostream>  
#include <vector>  
#include <iterator>  
using namespace std;  
  
class BigNumber  
{  
public:  
    BigNumber(Ipp32u value=0);  
    BigNumber(Ipp32s value);  
    BigNumber(const IppsBigNumState* pBN);  
    BigNumber(const Ipp32u* pData, int length=1, IppsBigNumSGN sgn=IppsBigNumPOS);  
    BigNumber(const BigNumber& bn);  
    BigNumber(const char *s);  
    virtual ~BigNumber();  
  
    // set value  
    void Set(const Ipp32u* pData, int length=1, IppsBigNumSGN sgn=IppsBigNumPOS);  
  
    // conversion to IppsBigNumState  
    friend IppsBigNumState* BN(const BigNumber& bn) {return bn.m_pBN;}  
    operator IppsBigNumState* () const { return m_pBN; }  
  
    // some useful constatns  
    static const BigNumber& Zero();
```

```

static const BigNumber& One();
static const BigNumber& Two();

// arithmetic operators probably need
BiggerNumber& operator = (const BigNumber& bn);
BiggerNumber& operator += (const BigNumber& bn);
BiggerNumber& operator -= (const BigNumber& bn);
BiggerNumber& operator *= (Ipp32u n);
BiggerNumber& operator *= (const BigNumber& bn);
BiggerNumber& operator /= (const BigNumber& bn);
BiggerNumber& operator % = (const BigNumber& bn);
friend BigNumber operator + (const BigNumber& a, const BigNumber& b);
friend BigNumber operator - (const BigNumber& a, const BigNumber& b);
friend BigNumber operator * (const BigNumber& a, const BigNumber& b);
friend BigNumber operator * (const BigNumber& a, Ipp32u);
friend BigNumber operator % (const BigNumber& a, const BigNumber& b);
friend BigNumber operator / (const BigNumber& a, const BigNumber& b);

// modulo arithmetic
BiggerNumber Modulo(const BigNumber& a) const;
BiggerNumber ModAdd(const BigNumber& a, const BigNumber& b) const;
BiggerNumber ModSub(const BigNumber& a, const BigNumber& b) const;
BiggerNumber ModMul(const BigNumber& a, const BigNumber& b) const;
BiggerNumber InverseAdd(const BigNumber& a) const;
BiggerNumber InverseMul(const BigNumber& a) const;

// comparisons
friend bool operator < (const BigNumber& a, const BigNumber& b);
friend bool operator > (const BigNumber& a, const BigNumber& b);
friend bool operator == (const BigNumber& a, const BigNumber& b);
friend bool operator != (const BigNumber& a, const BigNumber& b);
friend bool operator <= (const BigNumber& a, const BigNumber& b) {return !(a>b);}
friend bool operator >= (const BigNumber& a, const BigNumber& b) {return !(a<b);}

// easy tests
bool IsOdd() const;
bool IsEven() const { return !IsOdd(); }

// size of BigNumber
int MSB() const;
int LSB() const;
int BitSize() const { return MSB()+1; }
int DwordSize() const { return (BitSize()+31)>>5;}
friend int Bit(const vector<Ipp32u>& v, int n);

// conversion and output
void num2hex( string& s ) const; // convert to hex string
void num2vec( vector<Ipp32u>& v ) const; // convert to 32-bit word vector

friend ostream& operator << (ostream& os, const BigNumber& a);

protected:
    bool create(const Ipp32u* pData, int length, IppsBigNumSGN sgn=IppsBigNumPOS);
    int compare(const BigNumber& ) const;

    IppsBigNumState* m_pBN;
};

// convert bit size into 32-bit words
#define BITSIZE_WORD(n) (((n)+31)>>5)

```

```
#endif // _BIGNUMBER_H_
```

Definitions

C++ definitions for the `BigNumber` class methods are given below. For the declarations to be included, see the preceding [Declarations](#) section.

```
#include "bignum.h"

////////////////////////////////////////
//
// BigNumber
//
////////////////////////////////////////

BigNumber::~BigNumber()
{
    delete [] (Ipp8u*)m_pBN;
}

bool BigNumber::create(const Ipp32u* pData, int length, IppsBigNumSGN
sgn)
{
    int size;
    ippsBigNumGetSize(length, &size);
    m_pBN = (IppsBigNumState*)( new Ipp8u[size] );
    if(!m_pBN)
        return false;
    ippsBigNumInit(length, m_pBN);
    if(pData)
        ippsSet_BN(sgn, length, pData, m_pBN);
    return true;
}

// constructors
//
BigNumber::BigNumber(Ipp32u value)
{
    create(&value, 1, IppsBigNumPOS);
}

BigNumber::BigNumber(Ipp32s value)
{
    Ipp32s avalue = abs(value);
    create((Ipp32u*)&avalue, 1, (value<0)? IppsBigNumNEG :
IppsBigNumPOS);
}

BigNumber::BigNumber(const IppsBigNumState* pBN)
{
    int bnLen;
    ippsGetSize_BN(pBN, &bnLen);
    Ipp32u* bnData = new Ipp32u[bnLen];
    IppsBigNumSGN sgn;
    ippsGet_BN(&sgn, &bnLen, bnData, pBN);
    //
    create(bnData, bnLen, sgn);
    //
    delete bnData;
}
```

```

}

BigNumber::BigNumber(const Ipp32u* pData, int length, IppsBigNumSGN
sgn)
{
    create(pData, length, sgn);
}

static
char HexDigitList[] = "0123456789ABCDEF";

BigNumber::BigNumber(const char* s)
{
    bool neg = '-' == s[0];
    if(neg) s++;
    bool hex = ('0'==s[0]) && (('x'==s[1]) || ('X'==s[1]));

    int dataLen;
    Ipp32u base;
    if(hex) {
        s += 2;
        base = 0x10;
        dataLen = (strlen(s) + 7)/8;
    }
    else {
        base = 10;
        dataLen = (strlen(s) + 9)/10;
    }

    create(0, dataLen);
    *(this) = Zero();
    while(*s) {
        char tmp[2] = {s[0],0};
        Ipp32u digit = strchrn(HexDigitList, tmp);
        *this = (*this) * base + BigNumber( digit );
        s++;
    }

    if(neg)
        (*this) = Zero() - (*this);
}

BigNumber::BigNumber(const BigNumber& bn)
{
    IppsBigNumSGN sgn;
    int length;
    ippsGetSize_BN(bn.m_pBN, &length);
    Ipp32u* pData = new Ipp32u[length];
    ippsGet_BN(&sgn, &length, pData, bn.m_pBN);
    //
    create(pData, length, sgn);
    //
    delete [] pData;
}

// set value
//
void BigNumber::Set(const Ipp32u* pData, int length, IppsBigNumSGN
sgn)
{
    ippsSet_BN(sgn, length, pData, BN(*this));
}

```

```

}

// constants
//
const BigNumber& BigNumber::Zero()
{
    static const BigNumber zero(0);
    return zero;
}

const BigNumber& BigNumber::One()
{
    static const BigNumber one(1);
    return one;
}

const BigNumber& BigNumber::Two()
{
    static const BigNumber two(2);
    return two;
}

// arithmetic operators
//
BigNumber& BigNumber::operator =(const BigNumber& bn)
{
    if(this != &bn) { // prevent self copy
        int length;
        ippsGetSize_BN(bn.m_pBN, &length);
        Ipp32u* pData = new Ipp32u[length];
        IppsBigNumSGN sgn;
        ippsGet_BN(&sgn, &length, pData, bn.m_pBN);
        //
        delete (Ipp8u*)m_pBN;
        create(pData, length, sgn);
        //
        delete pData;
    }
    return *this;
}

BigNumber& BigNumber::operator += (const BigNumber& bn)
{
    int aLength;
    ippsGetSize_BN(BN(*this), &aLength);
    int bLength;
    ippsGetSize_BN(BN(bn), &bLength);
    int rLength = IPP_MAX(aLength, bLength) + 1;

    BigNumber result(0, rLength);
    ippsAdd_BN(BN(*this), BN(bn), BN(result));
    *this = result;
    return *this;
}

BigNumber& BigNumber::operator -= (const BigNumber& bn)
{
    int aLength;
    ippsGetSize_BN(BN(*this), &aLength);
    int bLength;

```



```

    ippsGetSize_BN(BN(bn), &bLength);
    int rLength = IPP_MAX(aLength,bLength);

    BigNumber result(0, rLength);
    ippsSub_BN(BN(*this), BN(bn), BN(result));
    *this = result;
    return *this;
}

BigNumber& BigNumber::operator *= (const BigNumber& bn)
{
    int aLength;
    ippsGetSize_BN(BN(*this), &aLength);
    int bLength;
    ippsGetSize_BN(BN(bn), &bLength);
    int rLength = aLength+bLength;

    BigNumber result(0, rLength);
    ippsMul_BN(BN(*this), BN(bn), BN(result));
    *this = result;
    return *this;
}

BigNumber& BigNumber::operator *= (Ipp32u n)
{
    int aLength;
    ippsGetSize_BN(BN(*this), &aLength);
    BigNumber bn(n);

    BigNumber result(0, aLength+1);
    ippsMul_BN(BN(*this), BN(bn), BN(result));
    *this = result;
    return *this;
}

BigNumber& BigNumber::operator %= (const BigNumber& bn)
{
    BigNumber remainder(bn);
    ippsMod_BN(BN(*this), BN(bn), BN(remainder));
    *this = remainder;
    return *this;
}

BigNumber& BigNumber::operator /= (const BigNumber& bn)
{
    BigNumber quotient(*this);
    BigNumber remainder(bn);
    ippsDiv_BN(BN(*this), BN(bn), BN(quotient), BN(remainder));
    *this = quotient;
    return *this;
}

BigNumber operator + (const BigNumber& a, const BigNumber&
b )
{
    BigNumber r(a);
    return r += b;
}

BigNumber operator - (const BigNumber& a, const BigNumber&
b )
{

```

```

    BigNumber r(a);
    return r -= b;
}

BigNumber operator * (const BigNumber& a, const BigNumber&
b )
{
    BigNumber r(a);
    return r *= b;
}

BigNumber operator * (const BigNumber& a, Ipp32u n)
{
    BigNumber r(a);
    return r *= n;
}

BigNumber operator / (const BigNumber& a, const BigNumber&
b )
{
    BigNumber q(a);
    return q /= b;
}

BigNumber operator % (const BigNumber& a, const BigNumber&
b )
{
    BigNumber r(b);
    ippsMod_BN(BN(a), BN(b), BN(r));
    return r;
}

// modulo arithmetic
//
BigNumber BigNumber::Modulo(const BigNumber& a) const
{
    return a % *this;
}

BigNumber BigNumber::InverseAdd(const BigNumber& a) const
{
    BigNumber t = Modulo(a);
    if(t==BigNumber::Zero())
        return t;
    else
        return *this - t;
}

BigNumber BigNumber::InverseMul(const BigNumber& a) const
{
    BigNumber r(*this);
    ippsModInv_BN(BN(a), BN(*this), BN(r));
    return r;
}

BigNumber BigNumber::ModAdd(const BigNumber& a, const BigNumber&
b) const
{
    BigNumber r = this->Modulo(a+b);
    return r;
}

```

```

BigNumber BigNumber::ModSub(const BigNumber& a, const BigNumber&
b) const
{
    BigNumber r = this->Modulo(a + this->InverseAdd(b));
    return r;
}

BigNumber BigNumber::ModMul(const BigNumber& a, const BigNumber&
b) const
{
    BigNumber r = this->Modulo(a*b);
    return r;
}

// comparison
//
int BigNumber::compare(const BigNumber &bn) const
{
    Ipp32u result;
    BigNumber tmp = *this - bn;
    ippsCmpZero_BN(BN(tmp), &result);
    return (result==IS_ZERO)? 0 : (result==GREATER_THAN_ZERO)? 1
: -1;
}

bool operator < (const BigNumber &a, const BigNumber &b)
{ return a.compare(b) < 0; }
bool operator > (const BigNumber &a, const BigNumber &b)
{ return a.compare(b) > 0; }
bool operator == (const BigNumber &a, const BigNumber &b)
{ return 0 == a.compare(b); }
bool operator != (const BigNumber &a, const BigNumber &b)
{ return 0 != a.compare(b); }

// easy tests
//
bool BigNumber::IsOdd() const
{
    vector<Ipp32u> v;
    num2vec(v);
    return v[0]&1;
}

// size of BigNumber
//
int BigNumber::LSB() const
{
    if( *this == BigNumber::Zero() )
        return 0;

    vector<Ipp32u> v;
    num2vec(v);

    int lsb = 0;
    vector<Ipp32u>::iterator i;
    for(i=v.begin(); i!=v.end(); i++) {
        Ipp32u x = *i;
        if(0==x)
            lsb += 32;
    }
}

```

```

        else {
            while(0==(x&1)) {
                lsb++;
                x >>= 1;
            }
            break;
        }
    }
    return lsb;
}

int BigNumber::MSB() const
{
    if( *this == BigNumber::Zero() )
        return 0;

    vector<Ipp32u> v;
    num2vec(v);

    int msb = v.size()*32 -1;
    vector<Ipp32u>::reverse_iterator i;
    for(i=v.rbegin(); i!=v.rend(); i++) {
        Ipp32u x = *i;
        if(0==x)
            msb -=32;
        else {
            while(!(x&0x80000000)) {
                msb--;
                x <<= 1;
            }
            break;
        }
    }
    return msb;
}

int Bit(const vector<Ipp32u>& v, int n)
{
    return 0 != ( v[n>>5] & (1<<(n&0x1F)) );
}

// conversions and output
//
void BigNumber::num2vec( vector<Ipp32u>& v ) const
{
    int length;
    ippsGetSize_BN(BN(*this), &length);
    Ipp32u* pData = new Ipp32u[length];
    IppsBigNumSGN sgn;
    ippsGet_BN(&sgn, &length, pData, BN(*this));
    //
    for(int n=0; n<length; n++)
        v.push_back( pData[n] );
    //
    delete pData;
}

void BigNumber::num2hex( string& s ) const
{
    int length;

```

```

    ippsGetSize_BN(BN(*this), &length);
    Ipp32u* pData = new Ipp32u[length];
    IppsBigNumSGN sgn;
    ippsGet_BN(&sgn, &length, pData, BN(*this));

    s.append(1, (sgn==IppsBigNumNEG)? '-' : ' ');
    s.append(1, '0');
    s.append(1, 'x');
    for(int n=length; n>0; n--) {
        Ipp32u x = pData[n-1];
        for(int nd=8; nd>0; nd--) {
            char c = HexDigitList[(x>>(nd-1)*4)&0xF];
            s.append(1, c);
        }
    }
    delete pData;
}

ostream& operator << ( ostream &os, const BigNumber& a ) {
    string s;
    a.num2hex(s);
    os << s.c_str();
    return os;
}

```

Functions for Creation of Cryptographic Contexts

The section presents source code for creation of some cryptographic contexts.

Declarations

Contents of the header file (`cpobjs.h`) declaring functions for creation of some cryptographic contexts are presented below:

```

#ifdef _CPOBJS_H_
#define _CPOBJS_H_
//
// create new of some ippsCP 'objects'
//
#include "ippcp.h"
#include <stdlib.h>

#define BITS_2_WORDS(n) (((n)+31)>>5)
int Bitsize2Wordsize(int nBits);

Ipp32u* rand32(Ipp32u* pX, int size);

IppsBigNumState* newBN(int len, const Ipp32u* pData=0);
IppsBigNumState* newRandBN(int len);
void deleteBN(IppsBigNumState* pBN);

IppsPRNGState* newPRNG(int seedBitsize=160);
void deletePRNG(IppsPRNGState* pPRNG);

IppsRSAState* newRSA(int lenN, int lenP, IppsRSAKeyType type);
void deleteRSA(IppsRSAState* pRSA);

IppsDLPState* newDLP(int lenM, int lenL);

```

```
void deleteDLP(IppsDLPState* pDLP);

#endif // _CPOBJS_H_
```

Definitions

C++ definitions of functions creating cryptographic contexts are given below. For the declarations to be included, see the preceding [Declarations](#) section.

```
#include "cpobj.h"

// convert bitsize into 32-bit wordsize
int Bitsize2Wordsize(int nBits)
{ return (nBits+31)>>5; }

// new BN number
IppsBigNumState* newBN(int len, const Ipp32u* pData)
{
    int size;
    ippsBigNumGetSize(len, &size);
    IppsBigNumState* pBN = (IppsBigNumState*)( new Ipp8u [size] );
    ippsBigNumInit(len, pBN);
    if(pData)
        ippsSet_BN(IppsBigNumPOS, len, pData, pBN);
    return pBN;
}

// desrtoy BN
void deleteBN(IppsBigNumState* pBN)
{ delete [] (Ipp8u*)pBN; }

// set up array of 32-bit items random
Ipp32u* rand32(Ipp32u* pX, int size)
{
    for(int n=0; n<size; n++)
        pX[n] = (rand()<<16) + rand();
    return pX;
}

IppsBigNumState* newRandBN(int len)
{
    Ipp32u* pBuffer = new Ipp32u [len];
    IppsBigNumState* pBN = newBN(len, rand32(pBuffer,len));
    delete [] pBuffer;
    return pBN;
}

//
// 'external' PRNG
//
IppsPRNGState* newPRNG(int seedBitsize)
{
    int seedSize = Bitsize2Wordsize(seedBitsize);
    Ipp32u* seed = new Ipp32u [seedSize];
    Ipp32u* augm = new Ipp32u [seedSize];
```

```

    int size;
    IppsBigNumState* pTmp;
    ippsPRNGGetSize(&size);
    IppsPRNGState* pCtx = (IppsPRNGState*)( new Ipp8u [size] );
    ippsPRNGInit(seedBitsize, pCtx);

    ippsPRNGSetSeed(pTmp=newBN(seedSize,rand32(seed,seedSize)), pCtx);
    delete [] (Ipp8u*)pTmp;
    ippsPRNGSetAugment(pTmp=newBN(seedSize,rand32(augm,seedSize)),
pCtx);
    delete [] (Ipp8u*)pTmp;

    delete [] seed;
    delete [] augm;
    return pCtx;
}
void deletePRNG(IppsPRNGState* pPRNG)
{ delete [] (Ipp8u*)pPRNG; }

//
// RSA context
//
IppsRSAState* newRSA(int lenN, int lenP, IppRSAKeyType type)
{
    int size;
    ippsRSAGetSize(lenN,lenP, type, &size);
    IppsRSAState* pCtx = (IppsRSAState*)( new Ipp8u [size] );
    ippsRSAInit(lenN,lenP, type, pCtx);

    return pCtx;
}
void deleteRSA(IppsRSAState* pRSA)
{ delete [] (Ipp8u*)pRSA; }

//
// DLP context
//
IppsDLPState* newDLP(int lenM, int lenL)
{
    int size;
    ippsDLPGetSize(lenM, lenL, &size);
    IppsDLPState *pCtx= (IppsDLPState *)new Ipp8u[ size ];
    ippsDLPInit(lenM, lenL, pCtx);

    return pCtx;
}
void deleteDLP(IppsDLPState* pDLP)
{ delete [] (Ipp8u*)pDLP; }

```


Calling the Cryptography Functions from Fortran-90



The Intel® Integrated Performance Primitives (Intel® IPP) for cryptography basically provide C interface. However, you can invoke Intel IPP cryptography functions directly from other languages if you are familiar with the inter-language calling conventions of your platforms. To promote portability, relieve you of having to deal with the calling convention specifics, and provide an interface to the functions that looks natural in Fortran-90, the Fortran-specific header file has been implemented.

The header file is available with the Intel IPP Samples, an extensive library of code samples and codecs implemented using the Intel IPP functions to help demonstrate the use of Intel IPP and accelerate the development of your application, components, and codecs. The samples can be downloaded from <http://www.intel.com/cd/software/products/asmo-na/eng/220046.htm>. The header file is a part of the Fortran-90 Interface to Intel IPP for Cryptography sample, which can be found in the `\ipp_samples\language-interface\f90-crypto` directory after downloading the code samples.

The interface header file `.\include\ippcp.f90` contains interfaces for direct calling Intel IPP cryptography functions from Fortran-90 applications.

Along with the header file, the sample, in particular, provides Fortran-90 examples for all Intel IPP cryptography functions.

The Fortran-90 Interface to Intel IPP for Cryptography sample demonstrates how to call Intel IPP cryptography functions using Fortran-90 API. The structure of the sample is designed so that each specific cryptographic algorithm, for example, an encryption in the OFB mode according to DES algorithm, is represented by a module. Each module contains the sequence of operations that an application code must perform to complete the algorithm. So, from the module you can get to know how to carry out the algorithmic chain, define parameters, set data, and call the cryptography functions in a Fortran-90 application. Modules related to a certain cryptographic algorithm, for example, TDES block cipher, make up a file with the corresponding name, `crypto_tdes_samples.f90`, in this case. Names of modules inside the files are also self-explanatory. For example, in the above file, the module `TDES_CBC_SAMPLE` implements a typical TDES encryption in the CBC mode.

The file `\ipp_samples\language-interface\f90-crypto\doc\crypto-f90.pdf` contains the detailed description of the Fortran-90 cryptography API and the sample usage.

In the code example below, modules are fragments of the definition module `IPPCP_DEFS` and interface module `IPPCP_F90` from the header file `ippcp.f90` and the `DES_ECB_SAMPLE` program demonstrates a Fortran-90 implementation of [Example "DES/TDES Encryption and Decryption"](#).

Fortran-90 Implementation of DES Encryption and Decryption

```
MODULE IPPCP_DEFS
```

```
! Codes for parameter PADDING
INTEGER(4), PARAMETER :: IppsCPPaddingNONE = 0

! DES/TDES DEFINITIONS
INTEGER(4), PARAMETER :: DES_BLOCKSIZE = 8 ! cipher blocksize (bytes)
INTEGER(4), PARAMETER :: DES_KEYSIZE = 8 ! cipher keysize (bytes)
```

```

END MODULE IPPCP_DEFS

MODULE IPPCP_F90

USE IPPCP_DEFS
INTERFACE
  !IppStatus ippsDESGetSize(int* pSize)
  INTEGER(4) FUNCTION ippsDESGetSize(size)

      INTEGER(4), INTENT(OUT) :: size
      !DEC$ATTRIBUTES STDCALL, DECORATE, ALIAS : 'ippsDESGetSize':: ippsDESGetSize
      !MS$ATTRIBUTES REFERENCE :: size
  END FUNCTION ippsDESGetSize
END INTERFACE

INTERFACE
  !IppStatus ippsDESInit(const Ipp8u* pKey, IppsDESSpec* pCtx);
  INTEGER(4) FUNCTION ippsDESInit(Key, SPEC_CONTEXT)
      USE IPPCP_DEFS
      CHARACTER(LEN=DES_KEYSIZE), INTENT(IN) :: Key
      CHARACTER(LEN=1), INTENT(INOUT) :: SPEC_CONTEXT(*)

      !DEC$ATTRIBUTES STDCALL, DECORATE, ALIAS : 'ippsDESInit'::ippsDESInit
      !MS$ATTRIBUTES REFERENCE :: Key
      !MS$ATTRIBUTES REFERENCE :: SPEC_CONTEXT
  END FUNCTION ippsDESInit
END INTERFACE

INTERFACE
  !IppStatus ippsDESEncryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int len,
  !                             const IppsDESSpec* pCtx, IppsCPPadding padding);
  INTEGER(4) FUNCTION ippsDESEncryptECB(Src, Dst, len, SPEC_CONTEXT, PADDING)

      CHARACTER(LEN=*), INTENT(IN) :: Src
      CHARACTER(LEN=*), INTENT(OUT) :: Dst
      INTEGER(4), INTENT(IN) :: len
      CHARACTER(LEN=1), INTENT(IN) :: SPEC_CONTEXT(*)
      INTEGER(4), INTENT(IN) :: PADDING

      !DEC$ATTRIBUTES STDCALL, DECORATE, ALIAS : 'ippsDESEncryptECB'::ippsDESEncryptECB
      !MS$ATTRIBUTES REFERENCE :: Src
      !MS$ATTRIBUTES REFERENCE :: Dst
      !MS$ATTRIBUTES REFERENCE :: SPEC_CONTEXT
  END FUNCTION ippsDESEncryptECB
END INTERFACE

INTERFACE
  !IppStatus ippsDESDecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int len,
  !                             const IppsDESSpec* pCtx, IppsCPPadding padding);
  INTEGER(4) FUNCTION ippsDESDecryptECB(Src, Dst, len, SPEC_CONTEXT, PADDING)

      CHARACTER(LEN=*), INTENT(IN) :: Src
      CHARACTER(LEN=*), INTENT(OUT) :: Dst
      INTEGER(4), INTENT(IN) :: len
      CHARACTER(LEN=1), INTENT(IN) :: SPEC_CONTEXT(*)
      INTEGER(4), INTENT(IN) :: PADDING

      !DEC$ATTRIBUTES STDCALL, DECORATE, ALIAS : 'ippsDESDecryptECB'::ippsDESDecryptECB
      !MS$ATTRIBUTES REFERENCE :: Src
      !MS$ATTRIBUTES REFERENCE :: Dst
      !MS$ATTRIBUTES REFERENCE :: SPEC_CONTEXT

```

```

        END FUNCTION ippsDESDecryptECB
    END INTERFACE

END MODULE IPPCP_F90

PROGRAM DES_ECB_SAMPLE
    USE IPPCP_F90

    INTEGER(4) :: size
    INTEGER(4) status

    CHARACTER(LEN=DES_KEYSIZE) Key
    INTEGER(1), DIMENSION(DES_KEYSIZE) :: Key_int
    INTEGER(4) textSize

! context description
    CHARACTER(LEN=1), ALLOCATABLE :: SPEC_CONTEXT(:)

    CHARACTER(LEN=40) ptext
    CHARACTER(LEN=40) rtext
    CHARACTER(LEN=40) ctext

! the size of text is always multiple of cipher block size
! (DES_DES_BLOCKSIZE =8 bytes)
    textSize = 40

! define the message to be encrypted
    ptext = 'quick brown fox jum over lazy dog      '

! define the Key
    Key_int = (/Z'01',Z'02',Z'03',Z'04',Z'05',Z'06',Z'07',Z'08'/)
    Key = TRANSFER(Key_int, Key)

! get size of the context needed for the encryption/decryption operation
    status = ippsDESGetSize(size)

! allocate context
    ALLOCATE (SPEC_CONTEXT(size))

! prepare the context for the DES usage
    status = ippsDESInit(Key, SPEC_CONTEXT)

! encrypt (ECB mode) ptext message
    status = ippsDESEncryptECB(ptext, ctext, textSize, SPEC_CONTEXT, IppsCPPaddingNONE)

! decrypt (ECB mode) ctext message
    status = ippsDESDecryptECB(ctext, rtext, textSize, SPEC_CONTEXT, IppsCPPaddingNONE)

! deallocate context
    DEALLOCATE (SPEC_CONTEXT)

END PROGRAM

```


Bibliography

This bibliography provides a list of publications that might be helpful to you in using cryptography functions of Intel IPP.

- [3GPP 35.202] *3GPP TS 35.202 V3.1.1 (2001-07). 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; Specification of the 3GPP Confidentiality and Integrity Algorithms; 3G Security; Document 2: KASUMI Specification (Release 1999).* Available from <http://isearch.etsi.org/3GPPSearch/isysquery/403fe057-469e-46a4-b298-f80b78bf4343/3/doc/35202-311.pdf>.
- [3GPP 2006] *Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification.* September 2006. Available from http://www.gsmworld.com/using/algorithms/docs/snow_3g_spec.pdf.
- [AC] Schneier, Bruce. *Applied Cryptography. Protocols, Algorithms, and Source Code in C.* Second Edition. John Wiley & Sons, Inc., 1996.
- [AES] Daemen, Joan, and Vincent Rijmen. *The Rijndael Block Cipher. AES Proposal.* Available from <http://www.nist.gov/aes>.
- [ANSI] *ANSI X9.62-1998 Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA).* American Bankers Association, 1999.
- [ANT] Cohen, Henri. *A Course in Computational Algebraic Number Theory.* Springer, 1998.
- [BF] Schneier, Bruce. *Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish).* Available from <http://www.schneier.com/blowfish.html>.
- [EC] Koblitz, Neal. *Introduction to Elliptic Curves and Modular Forms.* Springer, 1993.
- [EHCC] Cohen, Henri, and Gerald Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Chapman & Hall/CRC, 2006.
- [FIPS PUB 46-3] *Federal Information Processing Standards Publications, FIPS PUB 46-3.* Data Encryption Standard (DES), October 1999. Available from <http://csrc.nist.gov/publications/fips>.
- [FIPS PUB 113] *Federal Information Processing Standards Publications, FIPS PUB 113.* Computer Data Authentication, May 1985. Available from <http://csrc.nist.gov/publications/fips>.
- [FIPS PUB 180-2] *Federal Information Processing Standards Publications, FIPS PUB 180-2.* Secure Hash Standard, August 2002. Available from <http://csrc.nist.gov/publications/fips>.
- [FIPS PUB 186-2] *Federal Information Processing Standards Publications, FIPS PUB 186-2.* Digital Signature Standard (DSS), January 2000. Available from <http://csrc.nist.gov/publications/fips>.
- [FIPS PUB 198-1] *Federal Information Processing Standards Publications, FIPS PUB 198.* The Key-Hash Message Authentication Code (HMAC), July 2008. Available from <http://csrc.nist.gov/publications/fips>.
- [IEEE P1363A] *Standard Specifications for Public-Key Cryptography: Additional Techniques.* May, 2000. Working Draft.

- [NIST SP 800-38A] *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*. NIST Special Publication 800-38A, December 2001. Available from <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>.
- [NIST SP 800-38B] *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. NIST Special Publication 800-38B, May 2005. Available from http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf.
- [NIST SP 800-38C] *Draft Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. NIST Special Publication 800-38C, September 2003. Available from <http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf>.
- [NIST SP 800-38D] *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D, November 2007. Available from <http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf>.
- [PKCS 1.2.1] *RSA Laboratories. PKCS #1 v2.1: RSA Cryptography Standard*. June 2002. Available from <http://www.rsasecurity.com/rsalabs/pkcs>.
- [PKCS 7] *RSA Laboratories. PKCS #7: Cryptographic Message Syntax Standard*. An RSA Laboratories Technical Note Version 1.5 Revised, November 1, 1993.
- [RC5] Rivest, Ronald L. *The RC5 Encryption Algorithm*. Proceedings of the 1994 Leuven Workshop on Algorithms (Springer), 1994. Revised version, dated March 1997, is available from <http://theory.lcs.mit.edu/~cis/pubs/rivest/rc5rev.ps>.
- [RFC 1321] Rivest, Ronald L. *The MD5 Message-Digest Algorithm*. RFC 1321, MIT and RSA Data Security, Inc, April 1992. Available from <http://www.faqs.org/rfc1321.html>.
- [RFC 2401] Krawczyk, Hugo, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2401, February 1997. Available from <http://www.faqs.org/rfcs/rfc2401.html>.
- [RFC 3566] Frankel, Sheila, and Howard C. Herbert. *The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec*. RFC 3566, September 1996. Available from <http://www.rfc-archive.org/getrfc.php?rfc=3566>.
- [SEC1] *SEC1: Elliptic Curve Cryptography*. Standards for Efficient Cryptography Group, September 2000. Available from http://www.secg.org/secg_docs.htm.
- [SEC2] *SEC2: Recommended Elliptic Curve Domain Parameters*. Standards for Efficient Cryptography Group, September 2000. Available from http://www.secg.org/secg_docs.htm/.
- [TF] Schneier, Bruce, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *Twofish: A 128-Bit Block Cipher*. Available from <http://www.counterpane.com/twofish.html>.
- [X9.42] *X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography*. American National Standards Institute, 2003.

Index

A

AES-CCM Functions 85, 87, 88, 89, 90, 91, 92, 93
 Rijndael128CCMDecrypt 91
 Rijndael128CCMDecryptMessage 88
 Rijndael128CCMEncrypt 90
 Rijndael128CCMEncryptMessage 87
 Rijndael128CCMGetSize 89
 Rijndael128CCMGetTag 92
 Rijndael128CCMInit 89
 Rijndael128CCMMessageLen 92
 Rijndael128CCMStart 90
 Rijndael128CCMTagLen 93
AES-GCM Functions 93, 95, 96, 97, 98, 99, 101, 102, 103, 104
 Rijndael128GCMDecrypt 104
 Rijndael128GCMDecryptMessage 96
 Rijndael128GCMEncrypt 103
 Rijndael128GCMEncryptMessage 95
 Rijndael128GCMGetSize 97
 Rijndael128GCMGetSizeManaged 97
 Rijndael128GCMGetTag 104
 Rijndael128GCMInit 98
 Rijndael128GCMInitManaged 99
 Rijndael128GCMProcessAAD 103
 Rijndael128GCMProcessIV 102
 Rijndael128GCMReset 101
 Rijndael128GCMStart 101
AES-XCBC Functions 227, 228, 229, 230, 231
 XCBCRijndael128Final 230
 XCBCRijndael128GetSize 228
 XCBCRijndael128GetTag 230
 XCBCRijndael128Init 228
 XCBCRijndael128MessageTag 231
 XCBCRijndael128Update 229
AES-XCBC-MAC-96A 191
ARCFour Functions 150, 151, 152, 153, 154
 ARCFourCheckKey 151
 ARCFourDecrypt 153
 ARCFourEncrypt 153
 ARCFourGetSize 151
 ARCFourInit 152
 ARCFourPack 152
 ARCFourReset 154
 ARCFourUnpack 152
ARCFour stream cipher 150

B

Big Number Arithmetic 255
Big Number Arithmetic Functions
 Add_BN 272
 Add_BNU 256
 BigNumGetSize 264
 BigNumInit 264
 Cmp_BN 271
 CmpZero_BN 272
 Div_64u32u 260
 Div_BN 276
 ExtGet_BN 268
 Gcd_BN 277
 Get_BN 268
 GetOctString_BN 270
 GetOctString_BNU 263
 GetSize_BN 267
 MAC_BN_I 275
 MACOne_BNU_I 259
 Mod_BN 277
 ModInv_BN 278
 Mul_BN 274
 Mul_BNU4 259
 Mul_BNU8 260
 MulOne_BNU 258
 Ref_BN 269
 Set_BN 265
 SetOctString_BN 266
 SetOctString_BNU 263
 Sqr_32u64u 261
 Sqr_BNU4 262
 Sqr_BNU8 262
 Sub_BN 274
 Sub_BNU 257
Blowfish Functions 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115
 BlowfishDecryptCBC 110
 BlowfishDecryptCFB 111
 BlowfishDecryptCTR 115
 BlowfishDecryptECB 108
 BlowfishDecryptOFB 113
 BlowfishEncryptCBC 109
 BlowfishEncryptCFB 111
 BlowfishEncryptCTR 114
 BlowfishEncryptECB 108
 BlowfishEncryptOFB 112
 BlowfishGetSize 106
 BlowfishInit 106

Blowfish Functions (*continued*)

BlowfishPack 107
BlowfishUnpack 107

C

CMAC 191

CMAC Functions 224, 225, 226, 227
CMACRijndael128Final 226
CMACRijndael128GetSize 224
CMACRijndael128Init 225
CMACRijndael128MessageDigest 227
CMACRijndael128Update 225
CMACSafeRijndael128Init 225

concepts of IPP 22

cross-platform applications 22

D

Data Authentication Functions 231, 234, 235, 236, 237, 238,
239, 240, 241, 242, 243, 244, 245, 246, 247, 248,
249, 250, 251, 252, 253, 254

DAA Rijndael Functions 240
DAABlowfishFinal 250
DAABlowfishGetSize 249
DAABlowfishInit 249
DAABlowfishMessageDigest 251
DAABlowfishUpdate 250
DAADESFinal 235
DAADESGetSize 234
DAADESInit 234
DAADESMessageDigest 236
DAADESUpdate 235
DAARijndael128Final 241
DAARijndael128GetSize 240
DAARijndael128Init 240
DAARijndael128MessageDigest 242
DAARijndael128Update 241
DAARijndael192Final 244
DAARijndael192GetSize 243
DAARijndael192Init 243
DAARijndael192MessageDigest 245
DAARijndael192Update 244
DAARijndael256Final 247
DAARijndael256GetSize 246
DAARijndael256Init 246
DAARijndael256MessageDigest 248
DAARijndael256Update 247
DAASafeRijndael128Init 240
DAATDESFinal 238
DAATDESGetSize 237
DAATDESInit 237
DAATDESMessageDigest 239
DAATDESUpdate 238
DAATwofishFinal 253
DAATwofishGetSize 252
DAATwofishInit 252
DAATwofishMessageDigest 254
DAATwofishUpdate 253

Data Encryption Standard (DES) 26

DES/TDES Functions 26, 28, 29, 30, 31, 32, 33, 34, 35, 36,
37, 38, 39, 40, 41, 42, 43, 44

DESDecryptCBC 31
DESDecryptCFB 33
DESDecryptCTR 36
DESDecryptECB 30
DESDecryptOFB 34
DESEncryptCBC 31
DESEncryptCFB 32
DESEncryptCTR 35
DESEncryptECB 29
DESEncryptOFB 34
DESGetSize 28
DESInit 28
DESPack 29
DESUnpack 29
TDESDecryptCBC 39
TDESDecryptCFB 41
TDESDecryptCTR 44
TDESDecryptECB 37
TDESDecryptOFB 43
TDESEncryptCBC 38
TDESEncryptCFB 40
TDESEncryptCTR 43
TDESEncryptECB 37
TDESEncryptOFB 42

Discrete Logarithm Based Functions

DLPGenerateDH 366
DLPGenerateDSA 359
DLPGenKeyPair 357
DLPGet 354
DLPGetDP 356
DLPGetSize 352
DLPInit 352
DLPPack 353
DLPPublicKey 357
DLPSet 353
DLPSetDP 355
DLPSetKeyPair 359
DLPSecretDH 368
DLPSharedSecretDH 368
DLPSignDSA 361
DLPUnpack 353
DLPValidateDH 367
DLPValidateDSA 360
DLPValidateKeyPair 358
DLPVerifyDSA 362

E

Elliptic Curve Cryptographic Functions

ECCBAddPoint 410
ECCBCheckPoint 408
ECCBComparePoint 409
ECCBGenKeyPair 411
ECCBGet 402
ECCBGetOrderBitSize 403
ECCBGetPoint 407
ECCBGetSize 398
ECCBInit 399
ECCBMulPointScalar 411
ECCBNegativePoint 409

Elliptic Curve Cryptographic Functions (*continued*)

ECCBPointGetSize 405
 ECCBPointInit 405
 ECCBPublicKey 412
 ECCBSet 400
 ECCBSetKeyPair 414
 ECCBSetPoint 406
 ECCBSetPointAtInfinity 407
 ECCBSetStd 401
 ECCBSharedSecretDH 415
 ECCBSharedSecretDHC 416
 ECCBSignDSA 417
 ECCBSignNR 419
 ECCBValidate 403
 ECCBValidateKeyPair 413
 ECCBVerifyDSA 418
 ECCBVerifyNR 420
 ECCPAddPoint 383
 ECCPCheckPoint 381
 ECCPComparePoint 381
 ECCPGenKeyPair 384
 ECCPGet 375
 ECCPGetOrderBitSize 376
 ECCPGetPoint 380
 ECCPGetSize 372
 ECCPInit 372
 ECCPMulPointScalar 383
 ECCPNegativePoint 382
 ECCPPointGetSize 378
 ECCPPointInit 378
 ECCPPublicKey 385
 ECCPSet 373
 ECCPSetKeyPair 387
 ECCPSetPoint 379
 ECCPSetPointAtInfinity 379
 ECCPSetStd 374
 ECCPSharedSecretDH 388
 ECCPSharedSecretDHC 389
 ECCPSignDSA 390
 ECCPSignNR 392
 ECCPValidate 376
 ECCPValidateKeyPair 386
 ECCPVerifyDSA 391
 ECCPVerifyNR 393

Elliptic Curve Point Arithmetic Functions 463, 465, 466, 467,
 468, 469, 470, 471, 472, 473, 474, 475, 476, 477,
 478, 479, 480, 481, 482, 483, 484, 485, 486, 487

GFPECAAddPoint 475
 GFPECCmpPoint 474
 GFPECCpyPoint 472
 GFPECCGet 467
 GFPECCGetPoint 472
 GFPECCGetSize 465
 GFPECCInit 466
 GFPECMulPointScalar 476
 GFPECNegPoint 474
 GFPECPPointGetSize 469
 GFPECPPointInit 469
 GFPECCSet 467
 GFPECCSetPoint 470
 GFPECCSetPointAtInfinity 471

Elliptic Curve Point Arithmetic Functions (*continued*)

GFPECCSetPointRandom 471
 GFPECCVerify 468
 GFPECCVerifyPoint 473
 GFPECAAddPoint 486
 GFPECECCmpPoint 485
 GFPECECCpyPoint 483
 GFPECCGet 479
 GFPECCGetPoint 484
 GFPECCGetSize 476
 GFPECCInit 477
 GFPECECMulPointScalar 487
 GFPECCNegPoint 486
 GFPECCPointGetSize 480
 GFPECCPointInit 481
 GFPECCSet 478
 GFPECCSetPoint 481
 GFPECCSetPointAtInfinity 482
 GFPECCSetPointRandom 482
 GFPECCVerify 479
 GFPECCVerifyPoint 484

encryption, decryption, and encryption (E-D-E) sequence 26

F

Finite Field Arithmetic Functions 421, 424, 425, 426, 427,
 428, 429, 430, 431, 432, 433, 434, 435, 436, 437,
 438, 439, 440, 441, 442, 443, 444, 445, 446, 447,
 448, 449, 450, 451, 452, 453, 454, 455, 456, 457,
 458, 459, 460, 461, 462, 463

GFPAAdd 433
 GFPCmpElement 429
 GFPCpyElement 430
 GFPElementGetSize 426
 GFPElementInit 426
 GFPEXP 435
 GFPGet 425
 GFPGetElement 430
 GFPGetSize 424
 GFPGFPMontDecode 436
 GFPGFPMontEncode 435
 GFPIInit 425
 GFPIInv 432
 GFPMul 434
 GFPNeg 431
 GFPSetElement 427
 GFPSetElementPower2 428
 GFPSetElementRandom 429
 GFPSetElementZero 428
 GFPSqrt 432
 GFPSub 433
 GFPXAdd 445
 GFPXAdd_GFP 446
 GFPECCmpElement 442
 GFPECCpyElement 443
 GFPEXDiv 450
 GFPEXElementGetSize 439
 GFPEXElementInit 439
 GFPEXExp 450
 GFPEXGet 438
 GFPEXGetElement 444

Finite Field Arithmetic Functions (*continued*)

- GFPXGetSize 437
- GFPXInit 437
- GFPXInv 445
- GFPXMul 448
- GFPXMul_GFP 449
- GFPXNeg 444
- GFPXQAdd 460
- GFPXQCmpElement 457
- GFPXQCpyElement 457
- GFPXQElementGetSize 453
- GFPXQElementInit 454
- GFPXQExp 463
- GFPXQGet 453
- GFPXQGetElement 458
- GFPXQGetSize 451
- GFPXQInit 452
- GFPXQInv 459
- GFPXQMul 461
- GFPXQMul_GFP 462
- GFPXQNeg 459
- GFPXQSetElement 454
- GFPXQSetElementPowerX 456
- GFPXQSetElementRandom 456
- GFPXQSetElementZero 455
- GFPXQSub 461
- GFPXSetElement 440
- GFPXSetElementPowerX 441
- GFPXSetElementRandom 442
- GFPXSetElementZero 441
- GFPXSub 447
- GFPXSub_GFP 447

font conventions 21

Fortran-90 interface to cryptography functions 505

G

GetLibVersion 491

H

hash function 155

Hash Functions for Non-Streaming Messages 180, 181, 182, 184, 185

- general definition 180
- MD5MessageDigest 181
- SHA1MessageDigest 182
- SHA224MessageDigest 184
- SHA256MessageDigest 184
- SHA384MessageDigest 185
- SHA512MessageDigest 185
- user-implemented 180

HMAC 191

HMAC Functions 191

I

initialization vector iv 26

Intel Performance Library Suite 22

- ippcpGetLibVersion 491
- ippsAdd_BN 272
- ippsAdd_BNU 256
- ippsARCFive128DecryptCBC 145
- ippsARCFive128DecryptCFB 146
- ippsARCFive128DecryptCTR 149
- ippsARCFive128DecryptECB 143
- ippsARCFive128DecryptOFB 148
- ippsARCFive128EncryptCBC 144
- ippsARCFive128EncryptCFB 145
- ippsARCFive128EncryptCTR 148
- ippsARCFive128EncryptECB 142
- ippsARCFive128EncryptOFB 147
- ippsARCFive128GetSize 141
- ippsARCFive128Init 141
- ippsARCFive128Pack 142
- ippsARCFive128Unpack 142
- ippsARCFive64DecryptCBC 135
- ippsARCFive64DecryptCFB 137
- ippsARCFive64DecryptCTR 140
- ippsARCFive64DecryptECB 134
- ippsARCFive64DecryptOFB 138
- ippsARCFive64EncryptCBC 134
- ippsARCFive64EncryptCFB 136
- ippsARCFive64EncryptCTR 139
- ippsARCFive64EncryptECB 133
- ippsARCFive64EncryptOFB 137
- ippsARCFive64GetSize 131
- ippsARCFive64Init 132
- ippsARCFive64Pack 132
- ippsARCFive64Unpack 132
- ippsARCFourCheckKey 151
- ippsARCFourDecrypt 153
- ippsARCFourEncrypt 153
- ippsARCFourGetSize 151
- ippsARCFourInit 152
- ippsARCFourPack 152
- ippsARCFourReset 154
- ippsARCFourUnpack 152
- ippsBigNumGetSize 264
- ippsBigNumInit 264
- ippsBlowfishDecryptCBC 110
- ippsBlowfishDecryptCFB 111
- ippsBlowfishDecryptCTR 115
- ippsBlowfishDecryptECB 108
- ippsBlowfishDecryptOFB 113
- ippsBlowfishEncryptCBC 109
- ippsBlowfishEncryptCFB 111
- ippsBlowfishEncryptCTR 114
- ippsBlowfishEncryptECB 108
- ippsBlowfishEncryptOFB 112
- ippsBlowfishGetSize 106
- ippsBlowfishInit 106
- ippsBlowfishPack 107
- ippsBlowfishUnpack 107
- ippsCMACRijndael128Final 226
- ippsCMACRijndael128GetSize 224
- ippsCMACRijndael128Init 225
- ippsCMACRijndael128MessageDigest 227
- ippsCMACRijndael128Update 225
- ippsCMACSafeRijndael128Init 225

ippsCmp_BN 271
ippsCmpZero_BN 272
ippsDAABlowfishFinal 250
ippsDAABlowfishGetSize 249
ippsDAABlowfishInit 249
ippsDAABlowfishMessageDigest 251
ippsDAABlowfishUpdate 250
ippsDAADESFinal 235
ippsDAADESGetSize 234
ippsDAADESInit 234
ippsDAADESMessageDigest 236
ippsDAADESUpdate 235
ippsDAARijndael128Final 241
ippsDAARijndael128GetSize 240
ippsDAARijndael128Init 240
ippsDAARijndael128MessageDigest 242
ippsDAARijndael128Update 241
ippsDAARijndael192Final 244
ippsDAARijndael192GetSize 243
ippsDAARijndael192MessageDigest 245
ippsDAARijndael192Update 244
ippsDAARijndael256Final 247
ippsDAARijndael256GetSize 246
ippsDAARijndael256Init 246
ippsDAARijndael256MessageDigest 248
ippsDAARijndael256Update 247
ippsDAASafeRijndael128Init 240
ippsDAATDESFinal 238
ippsDAATDESGetSize 237
ippsDAATDESInit 237
ippsDAATDESMessageDigest 239
ippsDAATDESUpdate 238
ippsDAATwofishFinal 253
ippsDAATwofishGetSize 252
ippsDAATwofishInit 252
ippsDAATwofishMessageDigest 254
ippsDAATwofishUpdate 253
ippsDESDecryptCBC 31
ippsDESDecryptCFB 33
ippsDESDecryptCTR 36
ippsDESDecryptECB 30
ippsDESDecryptOFB 34
ippsDESEncryptCBC 31
ippsDESEncryptCFB 32
ippsDESEncryptCTR 35
ippsDESEncryptECB 29
ippsDESEncryptOFB 34
ippsDESGetSize 28
ippsDESInit 28
ippsDESPack 29
ippsDESUnpack 29
ippsDiv_64u32u 260
ippsDiv_BN 276
IppsDLPGenerateDH 366
ippsDLPGenerateDSA 359
ippsDLPGenKeyPair 357
ippsDLPGet 354
ippsDLPGetDP 356
ippsDLPGetSize 352
IppsDLPInit 352
ippsDLPPack 353
ippsDLPPublicKey 357
ippsDLPSet 353
ippsDLPSetDP 355
ippsDLPSetKeyPair 359
ippsDLPSharedSecretDH 368
ippsDLPSignDSA 361
ippsDLPUnpack 353
ippsDLPValidateDH 367
ippsDLPValidateDSA 360
ippsDLPValidateKeyPair 358
ippsDLPVerifyDSA 362
ippsECCBAddPoint 410
ippsECCBCheckPoint 408
ippsECCBComparePoint 409
ippsECCBGenKeyPair 411
ippsECCBGet 402
ippsECCBGetOrderBitSize 403
ippsECCBGetPoint 407
ippsECCBGetSize 398
ippsECCBInit 399
ippsECCBMulPointScalar 411
ippsECCBNegativePoint 409
ippsECCBPointGetSize 405
ippsECCBPointInit 405
ippsECCBPublicKey 412
ippsECCBSet 400
ippsECCBSetKeyPair 414
ippsECCBSetPoint 406
ippsECCBSetPointAtInfinity 407
ippsECCBSetStd 401
ippsECCBSharedSecretDH 415
ippsECCBSharedSecretDHC 416
ippsECCBSignDSA 417
ippsECCBSignNR 419
ippsECCBValidate 403
ippsECCBValidateKeyPair 413
ippsECCBVerifyDSA 418
ippsECCBVerifyNR 420
ippsECCPAddPoint 383
ippsECCPCheckPoint 381
ippsECCPComparePoint 381
ippsECCPGenKeyPair 384
ippsECCPGet 375
ippsECCPGetOrderBitSize 376
ippsECCPGetPoint 380
ippsECCPGetSize 372
ippsECCPInit 372
ippsECCPMulPointScalar 383
ippsECCPNegativePoint 382
ippsECCPPointGetSize 378
ippsECCPPointInit 378
ippsECCPPublicKey 385
ippsECCPSet 373
ippsECCPSetKeyPair 387
ippsECCPSetPoint 379
ippsECCPSetPointAtInfinity 379
ippsECCPSetStd 374
ippsECCPSharedSecretDH 388
ippsECCPSharedSecretDHC 389
ippsECCPSignDSA 390
ippsECCPSignNR 392

ippsECCPValidate 376	ippsGFPXECNegPoint 486
ippsECCPValidateKeyPair 386	ippsGFPXECPointGetSize 480
ippsECCPVerifyDSA 391	ippsGFPXECPointInit 481
ippsECCPVerifyNR 393	ippsGFPXECSet 478
ippsExtGet_BN 268	ippsGFPXECSetPoint 481
ippsGcd_BN 277	ippsGFPXECSetPointAtInfinity 482
ippsGet_BN 268	ippsGFPXECSetPointRandom 482
ippsGetOctString_BN 270	ippsGFPXECVerify 479
ippsGetOctString_BNU 263	ippsGFPXECVerifyPoint 484
ippsGetSize_BN 267	ippsGFPXElementGetSize 439
ippsGFPAAdd 433	ippsGFPXElementInit 439
ippsGFPCmpElement 429	ippsGFPXExp 450
ippsGFPCpyElement 430	ippsGFPXGet 438
ippsGFPECAAddPoint 475	ippsGFPXGetElement 444
ippsGFPECCmpPoint 474	ippsGFPXGetSize 437
ippsGFPECCpyPoint 472	ippsGFPXInit 437
ippsGFPECCGet 467	ippsGFPXInv 445
ippsGFPECCGetPoint 472	ippsGFPXMul 448
ippsGFPECCGetSize 465	ippsGFPXMul_GFP 449
ippsGFPECCInit 466	ippsGFPXNeg 444
ippsGFPECCMulPointScalar 476	ippsGFPXQAdd 460
ippsGFPECCNegPoint 474	ippsGFPXQCmpElement 457
ippsGFPECCPointGetSize 469	ippsGFPXQCpyElement 457
ippsGFPECCPointInit 469	ippsGFPXQElementGetSize 453
ippsGFPECCSet 467	ippsGFPXQElementInit 454
ippsGFPECCSetPoint 470	ippsGFPXQExp 463
ippsGFPECCSetPointAtInfinity 471	ippsGFPXQGet 453
ippsGFPECCSetPointRandom 471	ippsGFPXQGetElement 458
ippsGFPECCVerify 468	ippsGFPXQGetSize 451
ippsGFPECCVerifyPoint 473	ippsGFPXQInit 452
ippsGFPEElementGetSize 426	ippsGFPXQInv 459
ippsGFPEElementInit 426	ippsGFPXQMul 461
ippsGFPEExp 435	ippsGFPXQMul_GFP 462
ippsGFPGGet 425	ippsGFPXQNeg 459
ippsGFPGGetElement 430	ippsGFPXQSetElement 454
ippsGFPGGetSize 424	ippsGFPXQSetElementPowerX 456
ippsGFPGFPMontDecode 436	ippsGFPXQSetElementRandom 456
ippsGFPGFPMontEncode 435	ippsGFPXQSetElementZero 455
ippsGFPGInit 425	ippsGFPXQSub 461
ippsGFPGInv 432	ippsGFPXSetElement 440
ippsGFPGMul 434	ippsGFPXSetElementPowerX 441
ippsGFPGNeg 431	ippsGFPXSetElementRandom 442
ippsGFPGSetElement 427	ippsGFPXSetElementZero 441
ippsGFPGSetElementPower2 428	ippsGFPXSub 447
ippsGFPGSetElementRandom 429	ippsGFPXSub_GFP 447
ippsGFPGSetElementZero 428	ippsHMACMD5Duplicate 220
ippsGFPGSqrt 432	ippsHMACMD5Final 221
ippsGFPGSub 433	ippsHMACMD5GetSize 218
ippsGFPXAdd 445	ippsHMACMD5GetTag 222
ippsGFPXAdd_GFP 446	ippsHMACMD5Init 219
ippsGFPXCmpElement 442	ippsHMACMD5MessageDigest 223
ippsGFPXCpyElement 443	ippsHMACMD5Pack 219
ippsGFPXDiv 450	ippsHMACMD5Unpack 219
ippsGFPXECAddPoint 486	ippsHMACMD5Update 221
ippsGFPXECCmpPoint 485	ippsHMACSHA1Duplicate 196
ippsGFPXECCpyPoint 483	ippsHMACSHA1Final 197
ippsGFPXECGet 479	ippsHMACSHA1GetSize 195
ippsGFPXECGetPoint 484	ippsHMACSHA1GetTag 198
ippsGFPXECGetSize 476	ippsHMACSHA1Init 195
ippsGFPXECInit 477	ippsHMACSHA1MessageDigest 199
ippsGFPXECCMulPointScalar 487	ippsHMACSHA1Pack 196

ippsHMACSHA1Unpack 196
ippsHMACSHA1Update 197
ippsHMACSHA224Final 202
ippsHMACSHA224GetSize 199
ippsHMACSHA224GetTag 203
ippsHMACSHA224Init 200
ippsHMACSHA224MessageDigest 203
ippsHMACSHA224Pack 200
ippsHMACSHA224Unpack 200
ippsHMACSHA224Update 201
ippsHMACSHA256Duplicate 205
ippsHMACSHA256Final 207
ippsHMACSHA256GetTag 207
ippsHMACSHA256Init 204
ippsHMACSHA256MessageDigest 208
ippsHMACSHA256Pack 205
ippsHMACSHA256Unpack 205
ippsHMACSHA256Update 206
ippsHMACSHA384Duplicate 211
ippsHMACSHA384Final 212
ippsHMACSHA384GetSize 209
ippsHMACSHA384GetTag 213
ippsHMACSHA384Init 210
ippsHMACSHA384MessageDigest 213
ippsHMACSHA384Pack 210
ippsHMACSHA384Unpack 210
ippsHMACSHA384Update 211
ippsHMACSHA512Duplicate 215
ippsHMACSHA512Final 217
ippsHMACSHA512GetSize 214
ippsHMACSHA512GetTag 217
ippsHMACSHA512Init 214
ippsHMACSHA512MessageDigest 218
ippsHMACSHA512Pack 215
ippsHMACSHA512Unpack 215
ippsHMACSHA512Update 216
ippsMAC_BN_I 275
ippsMACOne_BNU_I 259
ippsMD5Duplicate 159
ippsMD5Final 160
ippsMD5GetSize 158
ippsMD5GetTag 161
ippsMD5Init 158
ippsMD5MessageDigest 181
ippsMD5Pack 159
ippsMD5Unpack 159
ippsMD5Update 160
ippsMGF_MD5 187
ippsMGF_SHA1 187
ippsMGF_SHA224 188
ippsMGF_SHA256 188
ippsMGF_SHA384 189
ippsMGF_SHA512 190
ippsMod_BN 277
ippsModInv_BN 278
ippsMontExp 286
ippsMontForm 283
ippsMontGet 282
ippsMontGetSize 281
ippsMontInit 281
ippsMontMul 284
ippsMontSet 282
ippsMul_BN 274
ippsMul_BNU4 259
ippsMul_BNU8 260
ippsMulOne_BNU 258
ippsPrimeGen 296
ippsPrimeGet 299
IppsPrimeGet_BN 299
ippsPrimeGetSize 295
ippsPrimeInit 296
ippsPrimeSet 298
ippsPrimeSet_BN 298
ippsPrimeTest 297
ippsPRNGen 291
ippsPRNGen_BN 292
ippsPRNGGetSize 288
ippsPRNGInit 288
ippsPRNGSetAugment 290
ippsPRNGSetH0 291
ippsPRNGSetModulus 290
ippsPRNGSetSeed 289
ippsRef_BN 269
ippsRijndael128CCMDecrypt 91
ippsRijndael128CCMDecryptMessage 88
ippsRijndael128CCMEncrypt 90
ippsRijndael128CCMEncryptMessage 87
ippsRijndael128CCMGetSize 89
ippsRijndael128CCMGetTag 92
ippsRijndael128CCMInit 89
ippsRijndael128CCMMessageLen 92
ippsRijndael128CCMStart 90
ippsRijndael128CCMTagLen 93
ippsRijndael128DecryptCBC 55
ippsRijndael128DecryptCCM 62
ippsRijndael128DecryptCCM_u8 63
ippsRijndael128DecryptCFB 56
ippsRijndael128DecryptCTR 60
ippsRijndael128DecryptECB 53
ippsRijndael128DecryptOFB 58
ippsRijndael128EncryptCBC 54
ippsRijndael128EncryptCCM 60
ippsRijndael128EncryptCCM_u8 61
ippsRijndael128EncryptCFB 56
ippsRijndael128EncryptCTR 59
ippsRijndael128EncryptECB 53
ippsRijndael128EncryptOFB 57
ippsRijndael128GCMDecrypt 104
ippsRijndael128GCMDecryptMessage 96
ippsRijndael128GCMEncrypt 103
ippsRijndael128GCMEncryptMessage 95
ippsRijndael128GCMGetSize 97
ippsRijndael128GCMGetSizeManaged 97
ippsRijndael128GCMGetTag 104
ippsRijndael128GCMInit 98
ippsRijndael128GCMInitManaged 99
ippsRijndael128GCMProcessAAD 103
ippsRijndael128GCMProcessIV 102
ippsRijndael128GCMReset 101
ippsRijndael128GCMStart 101
ippsRijndael128GetSize 51
ippsRijndael128Init 51

ippsRijndael128Pack 52	ippsRSASSASign_SHA256 333
ippsRijndael128Unpack 52	ippsRSASSASign_SHA256_PKCSv15 343
ippsRijndael192DecryptCBC 68	ippsRSASSASign_SHA384 334
ippsRijndael192DecryptCFB 70	ippsRSASSASign_SHA384_PKCSv15 344
ippsRijndael192DecryptCTR 73	ippsRSASSASign_SHA512 334
ippsRijndael192DecryptECB 67	ippsRSASSASign_SHA512_PKCSv15 345
ippsRijndael192DecryptOFB 71	ippsRSASSAVerify 335
ippsRijndael192EncryptCBC 67	ippsRSASSAVerify_MD5 336
ippsRijndael192EncryptCFB 69	ippsRSASSAVerify_MD5_PKCSv15 346
ippsRijndael192EncryptCTR 72	ippsRSASSAVerify_SHA1 337
ippsRijndael192EncryptECB 66	ippsRSASSAVerify_SHA1_PKCSv15 346
ippsRijndael192EncryptOFB 71	ippsRSASSAVerify_SHA224 337
ippsRijndael192GetSize 64	ippsRSASSAVerify_SHA224_PKCSv15 347
ippsRijndael192Init 65, 243	ippsRSASSAVerify_SHA256 338
ippsRijndael192Pack 65	ippsRSASSAVerify_SHA256_PKCSv15 348
ippsRijndael192Unpack 65	ippsRSASSAVerify_SHA384 339
ippsRijndael256DecryptCBC 78	ippsRSASSAVerify_SHA384_PKCSv15 349
ippsRijndael256DecryptCFB 79	ippsRSASSAVerify_SHA512 339
ippsRijndael256DecryptCTR 82	ippsRSASSAVerify_SHA512_PKCSv15 350
ippsRijndael256DecryptECB 76	ippsRSAUnpack 305
ippsRijndael256DecryptOFB 81	ippsRSAValidate 309
ippsRijndael256EncryptCBC 77	ippsSafeRijndael128Init 51
ippsRijndael256EncryptCFB 78	ippsSet_BN 265
ippsRijndael256EncryptCTR 82	ippsSetOctString_BN 266
ippsRijndael256EncryptECB 75	ippsSetOctString_BNU 263
ippsRijndael256EncryptOFB 80	ippsSHA1Duplicate 163
ippsRijndael256GetSize 74	ippsSHA1Final 164
ippsRijndael256Init 74	ippsSHA1GetSize 161
ippsRijndael256Pack 75	ippsSHA1GetTag 165
ippsRijndael256Unpack 75	ippsSHA1Init 162
ippsRSADecrypt 312	ippsSHA1MessageDigest 182
ippsRSADecrypt_PKCSv15 328	ippsSHA1Pack 162
ippsRSAEncrypt 312	ippsSHA1Unpack 162
ippsRSAEncrypt_PKCSv15 327	ippsSHA1Update 163
ippsRSAGenerate 308	ippsSHA224Duplicate 167
ippsRSAGetKey 307	ippsSHA224Final 168
ippsRSAGetSize 303	ippsSHA224GetSize 165
ippsRSAInit 304	ippsSHA224GetTag 168
ippsRSAOAEPDecrypt 322	ippsSHA224Init 166
ippsRSAOAEPDecrypt_MD5 323	ippsSHA224MessageDigest 184
ippsRSAOAEPDecrypt_SHA1 324	ippsSHA224Pack 166
ippsRSAOAEPDecrypt_SHA224 324	ippsSHA224Unpack 166
ippsRSAOAEPDecrypt_SHA256 325	ippsSHA224Update 167
ippsRSAOAEPDecrypt_SHA384 325	ippsSHA256Duplicate 170
ippsRSAOAEPDecrypt_SHA512 326	ippsSHA256Final 172
ippsRSAOAEPDecrypt_SHA512 326	ippsSHA256GetSize 169
ippsRSAOAEPDecrypt_SHA512 326	ippsSHA256GetTag 172
ippsRSAOAEPDecrypt_SHA512 326	ippsSHA256Init 169
ippsRSAOAEPDecrypt_SHA512 326	ippsSHA256MessageDigest 184
ippsRSAOAEPDecrypt_SHA512 326	ippsSHA256Pack 170
ippsRSAOAEPDecrypt_SHA512 326	ippsSHA256Unpack 170
ippsRSAOAEPDecrypt_SHA512 326	ippsSHA256Update 171
ippsRSAPack 305	ippsSHA384Duplicate 174
ippsRSASetKey 305	ippsSHA384Final 175
ippsRSASSASign 329	ippsSHA384GetSize 173
ippsRSASSASign_MD5 331	ippsSHA384GetTag 176
ippsRSASSASign_MD5_PKCSv15 341	ippsSHA384Init 173
ippsRSASSASign_SHA1 331	ippsSHA384MessageDigest 185
ippsRSASSASign_SHA1_PKCSv15 341	ippsSHA384Pack 174
ippsRSASSASign_SHA224 332	ippsSHA384Unpack 174
ippsRSASSASign_SHA224_PKCSv15 342	ippsSHA384Update 175

ippsSHA512Duplicate 178
 ippsSHA512Final 179
 ippsSHA512GetSize 176
 ippsSHA512GetTag 180
 ippsSHA512Init 177
 ippsSHA512MessageDigest 185
 ippsSHA512Pack 177
 ippsSHA512Unpack 177
 ippsSHA512Updat 178
 ippsSqr_32u64u 261
 ippsSqr_BNU4 262
 ippsSqr_BNU8 262
 ippsSub_BN 274
 ippsSub_BNU 257
 ippsTatePairingDE3Apply 490
 ippsTatePairingDE3Get 489
 ippsTatePairingDE3GetSize 488
 ippsTatePairingDE3Init 488
 ippsTDESDecryptCBC 39
 ippsTDESDecryptCFB 41
 ippsTDESDecryptCTR 44
 ippsTDESDecryptECB 37
 ippsTDESDecryptOFB 43
 ippsTDESEncryptCBC 38
 ippsTDESEncryptCFB 40
 ippsTDESEncryptCTR 43
 ippsTDESEncryptECB 37
 ippsTDESEncryptOFB 42
 ippsTwofishDecryptCBC 122
 ippsTwofishDecryptCFB 123
 ippsTwofishDecryptCTR 127
 ippsTwofishDecryptECB 120
 ippsTwofishDecryptOFB 125
 ippsTwofishEncryptCBC 121
 ippsTwofishEncryptCFB 123
 ippsTwofishEncryptCTR 126
 ippsTwofishEncryptECB 120
 ippsTwofishEncryptOFB 124
 ippsTwofishGetSize 118
 ippsTwofishInit 119
 ippsTwofishPack 119
 ippsTwofishUnpack 119
 ippsXCBCRijndael128Final 230
 ippsXCBCRijndael128GetSize 228
 ippsXCBCRijndael128GetTag 230
 ippsXCBCRijndael128Init 228
 ippsXCBCRijndael128MessageTag 231
 ippsXCBCRijndael128Update 229

K

Keyed Hash Functions 191, 195, 196, 197, 198, 199, 200,
 201, 202, 203, 204, 205, 206, 207, 208, 209, 210,
 211, 212, 213, 214, 215, 216, 217, 218, 219, 220,
 221, 222, 223
 HMACMD5Duplicate 220
 HMACMD5Final 221
 HMACMD5GetSize 218
 HMACMD5GetTag 222
 HMACMD5Init 219
 HMACMD5MessageDigest 223

Keyed Hash Functions (*continued*)

HMACMD5Pack 219
 HMACMD5Unpack 219
 HMACMD5Update 221
 HMACSHA1Duplicate 196
 HMACSHA1Final 197
 HMACSHA1GetSize 195
 HMACSHA1GetTag 198
 HMACSHA1Init 195
 HMACSHA1MessageDigest 199
 HMACSHA1Pack 196
 HMACSHA1Unpack 196
 HMACSHA1Update 197
 HMACSHA224Duplicate 201
 HMACSHA224Final 202
 HMACSHA224GetSize 199
 HMACSHA224GetTag 203
 HMACSHA224Init 200
 HMACSHA224MessageDigest 203
 HMACSHA224Pack 200
 HMACSHA224Unpack 200
 HMACSHA224Update 201
 HMACSHA256BufferSize 204
 HMACSHA256Duplicate 205
 HMACSHA256Final 207
 HMACSHA256GetTag 207
 HMACSHA256Init 204
 HMACSHA256MessageDigest 208
 HMACSHA256Pack 205
 HMACSHA256Unpack 205
 HMACSHA256Update 206
 HMACSHA384Duplicate 211
 HMACSHA384Final 212
 HMACSHA384GetSize 209
 HMACSHA384GetTag 213
 HMACSHA384Init 210
 HMACSHA384MessageDigest 213
 HMACSHA384Pack 210
 HMACSHA384Unpack 210
 HMACSHA384Update 211
 HMACSHA512Duplicate 215
 HMACSHA512Final 217
 HMACSHA512GetSize 214
 HMACSHA512GetTag 217
 HMACSHA512Init 214
 HMACSHA512MessageDigest 218
 HMACSHA512Pack 215
 HMACSHA512Unpack 215
 HMACSHA512Update 216

M

mask generation function 186
 Mask Generation Functions 186, 187, 188, 189, 190
 MGF_MD5 187
 MGF_SHA1 187
 MGF_SHA224 188
 MGF_SHA256 188
 MGF_SHA384 189
 MGF_SHA512 190
 user-implemented 186

MD5 and SHA Algorithms 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180
MD5Duplicate 159
MD5Final 160
MD5GetSize 158
MD5GetTag 161
MD5Init 158
MD5MessageDigest 161
MD5Pack 159
MD5Unpack 159
MD5Update 160
SHA1Duplicate 163
SHA1Final 164
SHA1GetSize 161
SHA1GetTag 165
SHA1Init 162
SHA1Pack 162
SHA1Unpack 162
SHA1Update 163
SHA224Duplicate 167
SHA224Final 168
SHA224GetSize 165
SHA224GetTag 168
SHA224Init 166
SHA224Pack 166
SHA224Unpack 166
SHA224Update 167
SHA256Duplicate 170
SHA256Final 172
SHA256GetSize 169
SHA256GetTag 172
SHA256Init 169
SHA256MessageDigest 173
SHA256Pack 170
SHA256Unpack 170
SHA256Update 171
SHA384Duplicate 174
SHA384Final 175
SHA384GetSize 173
SHA384GetTag 176
SHA384Init 173
SHA384MessageDigest 176
SHA384Pack 174
SHA384Unpack 174
SHA384Update 175
SHA512Duplicate 178
SHA512Final 179
SHA512GetSize 176
SHA512GetTag 180
SHA512Init 177
SHA512Pack 177
SHA512Unpack 177
SHA512Update 178
Message Authentication Functions 191, 224, 227
 AES-XCBC Functions 227
 CMAC Functions 224
 Keyed Hash Functions 191
MGF 186
modes of operation for block ciphers
 CBC 25

modes of operation for block ciphers (*continued*)
 CCM 85
 CFB 25
 CTR 25
 ECB 25
 OFB 25
Montgomery Reduction Scheme Functions 279, 281, 282, 283, 284, 286
 MontInit 281
 MontSet 282
 MontExp 286
 MontForm 283
 MontGet 282
 MontGetSize 281
 MontMul 284

N

naming conventions 21
notational conventions 21

P

PKCS V1.5 Encryption Scheme Functions 326
PKCS V1.5 Signature Scheme Functions 340
Prime Number Generation Functions 294, 295, 296, 297, 298, 299
 PrimeGen 296
 PrimeGetSize 295
 PrimeInit 296
 PrimeGet 299
 PrimeGet_BN 299
 PrimeSet 298
 PrimeSet_BN 298
 PrimeTest 297
Pseudorandom Number Generation Functions 286, 287, 288, 289, 290, 291, 292
 PRNGen 291
 PRNGen_BN 292
 PRNGGetSize 288
 PRNGInit 288
 PRNGSetAugment 290
 PRNGSetH0 291
 PRNGSetModulus 290
 PRNGSetSeed 289
 user-implemented 287

R

RC4 stream cipher 150
RC5 Functions 129, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149
 ARCFive128DecryptCBC 145
 ARCFive128DecryptCFB 146
 ARCFive128DecryptCTR 149
 ARCFive128DecryptECB 143
 ARCFive128DecryptOFB 148
 ARCFive128EncryptCBC 144

RC5 Functions (*continued*)

ARCFive128EncryptCFB 145
 ARCFive128EncryptCTR 148
 ARCFive128EncryptECB 142
 ARCFive128EncryptOFB 147
 ARCFive128GetSize 141
 ARCFive128Init 141
 ARCFive128Pack 142
 ARCFive128Unpack 142
 ARCFive64DecryptCBC 135
 ARCFive64DecryptCFB 137
 ARCFive64DecryptCTR 140
 ARCFive64DecryptECB 134
 ARCFive64DecryptOFB 138
 ARCFive64EncryptCBC 134
 ARCFive64EncryptCFB 136
 ARCFive64EncryptCTR 139
 ARCFive64EncryptECB 133
 ARCFive64EncryptOFB 137
 ARCFive64GetSize 131
 ARCFive64Init 132
 ARCFive64Pack 132
 ARCFive64Unpack 132

reference code 22

Rijndael Functions 47, 51, 52, 53, 54, 55, 56, 57, 58, 59,
 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 85, 87, 88,
 89, 90, 91, 92, 93, 95, 96, 97, 98, 99, 101, 102,
 103, 104

AES-CCM Functions 85

AES-GCM Functions 93

Rijndael128CCMDecrypt 91

Rijndael128CCMDecryptMessage 88

Rijndael128CCMEncrypt 90

Rijndael128CCMEncryptMessage 87

Rijndael128CCMGetSize 89

Rijndael128CCMGetTag 92

Rijndael128CCMInit 89

Rijndael128CCMMessageLen 92

Rijndael128CCMStart 90

Rijndael128CCMTagLen 93

Rijndael128DecryptCBC 55

Rijndael128DecryptCCM 62

Rijndael128DecryptCCM_u8 63

Rijndael128DecryptCFB 56

Rijndael128DecryptCTR 60

Rijndael128DecryptECB 53

Rijndael128DecryptOFB 58

Rijndael128EncryptCBC 54

Rijndael128EncryptCCM 60

Rijndael128EncryptCCM_u8 61

Rijndael128EncryptCFB 56

Rijndael128EncryptCTR 59

Rijndael128EncryptECB 53

Rijndael128EncryptOFB 57

Rijndael128GCMDecrypt 104

Rijndael128GCMDecryptMessage 96

Rijndael128GCMEncrypt 103

Rijndael128GCMEncryptMessage 95

Rijndael128GCMGetSize 97

Rijndael128GCMGetSizeManaged 97

Rijndael Functions (*continued*)

Rijndael128GCMGetTag 104

Rijndael128GCMInit 98

Rijndael128GCMInitManaged 99

Rijndael128GCMProcessAAD 103

Rijndael128GCMProcessIV 102

Rijndael128GCMReset 101

Rijndael128GCMStart 101

Rijndael128GetSize 51

Rijndael128Init 51

Rijndael128Pack 52

Rijndael128Unpack 52

Rijndael192DecryptCBC 68

Rijndael192DecryptCFB 70

Rijndael192DecryptCTR 73

Rijndael192DecryptECB 67

Rijndael192DecryptOFB 71

Rijndael192EncryptCBC 67

Rijndael192EncryptCFB 69

Rijndael192EncryptCTR 72

Rijndael192EncryptECB 66

Rijndael192EncryptOFB 71

Rijndael192GetSize 64

Rijndael192Init 65

Rijndael192Pack 65

Rijndael192Unpack 65

Rijndael256DecryptCBC 78

Rijndael256DecryptCFB 79

Rijndael256DecryptCTR 82

Rijndael256DecryptECB 76

Rijndael256DecryptOFB 81

Rijndael256EncryptCBC 77

Rijndael256EncryptCFB 78

Rijndael256EncryptCTR 82

Rijndael256EncryptECB 75

Rijndael256EncryptOFB 80

Rijndael256GetSize 74

Rijndael256Init 74

Rijndael256Pack 75

Rijndael256Unpack 75

SafeRijndael128Init 51

RSA Algorithm Functions 302

RSA Primitives 311, 312

RSADecrypt 312

RSASerialize 312

RSA System Building Functions 303, 304, 305, 307, 308, 309

RSAGenerate 308

RSAGetKey 307

RSAGetTag 303

RSASerialize 304

RSAPack 305

RSASerialize 305

RSASerialize 305

RSASerialize 309

RSA-based Encryption Scheme Functions 316, 317, 318,

319, 320, 321, 322, 323, 324, 325, 326, 327, 328

RSADecrypt_PKCSv15 328

RSASerialize_PKCSv15 327

RSASerializeDecrypt 322

RSASerializeDecrypt_MD5 323

RSASerializeDecrypt_SHA1 324

RSA-based Encryption Scheme Functions (*continued*)

- RSOAEPDecrypt_SHA224 324
- RSOAEPDecrypt_SHA256 325
- RSOAEPDecrypt_SHA384 325
- RSOAEPDecrypt_SHA512 326
- RSOAEPDecrypt 317
- RSOAEPDecrypt_MD5 318
- RSOAEPDecrypt_SHA1 319
- RSOAEPDecrypt_SHA224 319
- RSOAEPDecrypt_SHA256 320
- RSOAEPDecrypt_SHA384 321
- RSOAEPDecrypt_SHA512 321

RSA-based scheme 302

RSA-based Signature Scheme Functions 329, 331, 332, 333, 334, 335, 336, 337, 338, 339, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350

- RSASSASign 329
- RSASSASign_MD5 331
- RSASSASign_MD5_PKCSv15 341
- RSASSASign_SHA1 331
- RSASSASign_SHA1_PKCSv15 341
- RSASSASign_SHA224 332
- RSASSASign_SHA224_PKCSv15 342
- RSASSASign_SHA256 333
- RSASSASign_SHA256_PKCSv15 343
- RSASSASign_SHA384 334
- RSASSASign_SHA384_PKCSv15 344
- RSASSASign_SHA512 334
- RSASSASign_SHA512_PKCSv15 345
- RSASSAVerify 335
- RSASSAVerify_MD5 336
- RSASSAVerify_MD5_PKCSv15 346
- RSASSAVerify_SHA1 337
- RSASSAVerify_SHA1_PKCSv15 346
- RSASSAVerify_SHA224 337
- RSASSAVerify_SHA224_PKCSv15 347
- RSASSAVerify_SHA256 338
- RSASSAVerify_SHA256_PKCSv15 348
- RSASSAVerify_SHA384 339
- RSASSAVerify_SHA384_PKCSv15 349
- RSASSAVerify_SHA512 339
- RSASSAVerify_SHA512_PKCSv15 350

RSA-OAEP Scheme Functions 316

RSASSA-PSS Scheme Functions 329

S

- support functions and classes 491
- symmetric algorithm modes
 - Cipher Block Chain (CBC) 25
 - Cipher Feedback (CFB) 25
 - Counter (CTR) 25
 - Counter with Cipher Block Chaining-Message Authentication Code (CCM) 85
 - Electronic Code Book (ECB) 25
 - Output Feedback (OFB) 25

T

Tate Pairing Functions 487, 488, 489, 490

- TatePairingDE3Apply 490
- TatePairingDE3Get 489
- TatePairingDE3GetSize 488
- TatePairingDE3Init 488

Triple Data Encryption Standard (TDES) 26

Twofish Functions 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127

- TwofishDecryptCBC 122
- TwofishDecryptCFB 123
- TwofishDecryptCTR 127
- TwofishDecryptECB 120
- TwofishDecryptOFB 125
- TwofishEncryptCBC 121
- TwofishEncryptCFB 123
- TwofishEncryptCTR 126
- TwofishEncryptECB 120
- TwofishEncryptOFB 124
- TwofishGetSize 118
- TwofishInit 119
- TwofishPack 119
- TwofishUnpack 119

V

version information function 491