Contents

Legal Information ........................................................................................................ 11
Getting Help and Support ....................................................................................... 13
Introducing Cryptography for Intel® Integrated Performance Primitives .......... 15
What's New ................................................................................................................ 17
Notational Conventions ......................................................................................... 19
Related Products ..................................................................................................... 21

Chapter 1: Overview
- Basic Features ...................................................................................................... 23
- Function Context Structures .............................................................................. 23
- Data Security Considerations .......................................................................... 24

Chapter 2: Symmetric Cryptography Primitive Functions
- Block Cipher Modes of Operation ..................................................................... 25
- Rijndael Functions ............................................................................................... 26
  - AESGetSize ..................................................................................................... 26
  - AESInit .......................................................................................................... 27
  - AESSetKey ..................................................................................................... 28
  - AESPack, AESUnpack .................................................................................. 28
  - AESEncryptECB ......................................................................................... 29
  - AESDecryptECB ......................................................................................... 30
  - AESEncryptCBC ......................................................................................... 31
  - AESDecryptCBC ......................................................................................... 32
  - AESEncryptCFB ......................................................................................... 32
  - AESDecryptCFB ......................................................................................... 33
  - AESEncryptOFB ......................................................................................... 34
  - AESDecryptOFB ......................................................................................... 35
  - AESEncryptCTR ......................................................................................... 36
  - AESDecryptCTR ......................................................................................... 37
  - Example of Using AES Functions ................................................................ 38
- AES-CCM Functions ............................................................................................ 39
  - AES_CCMGetSize ......................................................................................... 39
  - AES_CCMInit ............................................................................................... 40
  - AES_CCMStart ............................................................................................. 41
  - AES_CCMEncrypt ....................................................................................... 41
  - AES_CCMDecrypt ....................................................................................... 42
  - AES_CCMGetTag ......................................................................................... 43
  - AES_CCMMessageLen ............................................................................... 43
  - AES_CCMTagLen ......................................................................................... 44
- AES-GCM Functions ............................................................................................ 45
  - AES_GCMGetSize ......................................................................................... 46
  - AES_GCMInit ............................................................................................... 46
  - AES_GCMStart ............................................................................................. 47
  - AES_GCMReset ............................................................................................ 48
Chapter 3: One-Way Hash Primitives

Hash Functions

HashGetSize ................................................................. 83
HashInit ................................................................. 84
HashPack, HashUnpack ........................................... 85
HashDuplicate .......................................................... 86
HashUpdate ............................................................. 87
Chapter 4: Data Authentication Primitive Functions

Message Authentication Functions

Keyed Hash Functions

HMACGetSize

HMAC_Init

HMAC_Pack, HMAC_Unpack

HMAC_Duplicate

HMAC_Update

HMAC_Final

HMAC_GetTag

HMAC_Message

CMAC Functions

AES_CMACGetSize

AES_CMACInit

AES_CMACUpdate

AES_CMACFinal

AES_CMACGetTag

Chapter 5: Public Key Cryptography Functions

Big Number Arithmetic

BigNumGetSize

BigNumInit

Set_BN

SetOctString_BN

GetSize_BN

Get_BN

ExtGet_BN

Ref_BN

GetOctString_BN

Cmp_BN

CmpZero_BN

Add_BN

Sub_BN

Mul_BN

MAC_BN_I
RSA Primitives ................................................................. 197
  RSA_Encrypt ............................................................... 198
  RSA_Decrypt ............................................................... 199
Example of Using RSA Primitive Functions ................. 200
RSA Encryption Schemes ................................................ 202
  RSA-OAEP Scheme Functions ........................................ 202
PKCS V1.5 Encryption Scheme Functions .................... 205
RSA Signature Schemes .................................................. 207
  RSA-SSA Scheme Functions ......................................... 207
PKCS V1.5 Signature Scheme Functions .......................... 210
Discrete-Logarithm-Based Cryptography Functions .......... 213
  DLPGetSize .................................................................. 213
  DLPInit ....................................................................... 214
  DLPPack, DLPUunpack ................................................ 215
  DLPSet ................................................................. 215
  DLPGet ................................................................. 216
  DLPSetDP .............................................................. 217
  DLPGetDP .............................................................. 218
  DLPGenKeyPair ....................................................... 219
  DLPPublicKey .......................................................... 220
  DLPValidateKeyPair .................................................. 220
  DLPSetKeyPair ........................................................ 221
  DLPGenerateDSA ....................................................... 222
  DLPValidateDSA ....................................................... 223
  DLPSignDSA ........................................................... 224
  DLPVerifyDSA ........................................................ 225
Example of Using Discrete-logarithm Based Primitive Functions 226
  DLPGenerateDH ......................................................... 228
  DLPValidateDH ......................................................... 229
  DLPSharedSecretDH .................................................. 230
  DLGetResultString ..................................................... 231
Elliptic Curve Cryptography Functions .......................... 232
  Functions Based on GF(p) ............................................ 232
    ECCPGetSize .......................................................... 233
    ECCPGetSizeStd ..................................................... 233
    ECCPInit ............................................................. 234
    ECCPInitStd ......................................................... 235
    ECCPSet ............................................................ 236
    ECCPSetStd ........................................................ 237
    ECCPGet ............................................................ 238
    ECCPGetOrderBitSize ............................................. 239
    ECCPValidate ....................................................... 240
    ECCPPointGetSize ............................................... 241
    ECCPPointInit ...................................................... 242
    ECCPSetPoint ....................................................... 243
    ECCPSetPointAtInfinity ......................................... 243
    ECCPGetPoint ...................................................... 244
    ECCPCheckpoint .................................................. 245
    ECCPComparePoint ............................................... 246
    ECCPNegativePoint ................................................. 247
### Contents

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECCPAddPoint</td>
<td>247</td>
</tr>
<tr>
<td>ECCPMulPointScalar</td>
<td>248</td>
</tr>
<tr>
<td>ECCPGenKeyPair</td>
<td>249</td>
</tr>
<tr>
<td>ECCPPublicKey</td>
<td>250</td>
</tr>
<tr>
<td>ECCPValidateKeyPair</td>
<td>251</td>
</tr>
<tr>
<td>ECCPSetKeyPair</td>
<td>252</td>
</tr>
<tr>
<td>ECCPSharedSecretDH</td>
<td>253</td>
</tr>
<tr>
<td>ECCPSharedSecretDHC</td>
<td>254</td>
</tr>
<tr>
<td>ECCPSignDSA</td>
<td>255</td>
</tr>
<tr>
<td>ECCPVerifyDSA</td>
<td>256</td>
</tr>
<tr>
<td>ECCPSignNR</td>
<td>257</td>
</tr>
<tr>
<td>ECCPVerifyNR</td>
<td>259</td>
</tr>
<tr>
<td>ECCPSignSM2</td>
<td>260</td>
</tr>
<tr>
<td>ECCPVerifySM2</td>
<td>261</td>
</tr>
<tr>
<td>Signing/Verification Using the Elliptic Curve Cryptography Functions over a Prime Finite Field</td>
<td>262</td>
</tr>
<tr>
<td>ECCGetResultString</td>
<td>264</td>
</tr>
</tbody>
</table>

### Appendix A: Support Functions and Classes

- Version Information Function ............................................. 267
- GetLibVersion ................................................................ 267
- Classes and Functions Used in Examples ............................. 268
  - BigNumber Class ......................................................... 268
  - Functions for Creation of Cryptographic Contexts ............. 276

### Appendix B: Removed Functions

### Appendix C: Bibliography
Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications.


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.

Cilk, Intel, the Intel logo, Intel Atom, Intel Core, Intel Inside, Intel NetBurst, Intel SpeedStep, Intel vPro, Intel Xeon Phi, Intel XScale, Itanium, MMX, Pentium, Thunderbolt, Ultrabook, VTune and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

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

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

Java is a registered trademark of Oracle and/or its affiliates.

Copyright© 2000-2015, Intel Corporation. All rights reserved.
Getting Help and Support

Getting Help
You can get context-sensitive help for the Intel® Integrated Performance Primitives (Intel® IPP) in the Microsoft Visual Studio* development system on Windows* OS. To do this, select the function name in the code editor, and click F1.

Getting Technical Support
If you did not register your Intel software product during installation, please do so now at the Intel® Software Development Products Registration Center. Registration entitles you to free technical support, product updates and upgrades for the duration of the support term.

For general information about Intel technical support, product updates, user forums, FAQs, tips and tricks and other support questions, please visit http://www.intel.com/software/products/support/.

NOTE
If your distributor provides technical support for this product, please contact them rather than Intel.

Introducing Cryptography for Intel® Integrated Performance Primitives

The Intel® Integrated Performance Primitives (Intel® IPP) is a software library that provides a comprehensive set of application domain-specific highly optimized functions for signal, image, and video processing.

- The Intel IPP signal processing software is a collection of low-overhead, high-performance operations performed on one-dimensional (1D) data arrays. Examples of such operations are linear transforms, filtering, and vector math.
- The Intel IPP image and video processing software is a collection of low-overhead, high-performance operations performed on two-dimensional (2D) arrays of pixels. Examples of such operations are linear transforms, filtering, and arithmetic on image data.

The Intel IPP software enables taking advantage of the parallelism of single-instruction, multiple data (SIMD) instructions, which make the core of the MMX technology and Streaming SIMD Extensions. These technologies improve the performance of computation-intensive signal, image, and video processing applications. Plenty of the Intel IPP functions are tuned and threaded for multi-core systems.

Intel IPP supports application development for various Intel® architectures. By providing a single cross-architecture application programmer interface, Intel IPP permits software application repurposing and enables developers to port to unique features across Intel® processor-based desktop, server, mobile, and handheld platforms. Use of the Intel IPP primitive functions can help drastically reduce development costs and accelerate time-to-market by eliminating the need of writing processor-specific code for computation intensive routines.

Cryptography for Intel IPP is an add-on library that offers Intel IPP users a cross-platform and cross operating system application programming interface (API) for routines commonly used for cryptographic operations.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

Notice revision #20110804
What's New

This Reference Manual documents Intel® Integrated Performance Primitives (Intel® IPP) 9.0 release.

The document has been updated to reflect the following changes to the product:

- Primitives for the SMS4 block cipher have been added (See SMS4 Functions for details.)
- Primitives that implement the SM3 hash algorithm have been added (See One-Way Hash Primitives for details.)
- The functions `ECCPSignSM2` and `ECCPVerifySM2` that implement SM2 scheme have been added.
- The functions based on hash algorithms (Hash Functions, Hash Functions for Non-Streaming Messages, Mask Generation Functions, key Hash Functions, RSA-OAEP Scheme Functions, and RSA Signature Schemes) now use the `IppHashAlgId` enumerator to specify a particular hash algorithm used. (See One-Way Hash Primitives for details.)
- To facilitate usage of cryptosystems based on standard elliptic curves, the `IppECCType` enumerator has been added (see Elliptic Curve Cryptography Functions), as well as functions `ECCPGetSizeStd`, `ECCPInitStd`, and extensions of the function `ECCPSetStd`.
- Functions that implement RSA Encryption Schemes and RSA Signature Schemes have been renamed for consistency and enhanced.

Numerous deprecated functions have been removed (see Removed Functions for details).

Additionally, minor updates have been made to fix inaccuracies in the document.
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.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
<tr>
<td>Notice revision #20110804</td>
</tr>
</tbody>
</table>

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`, `pSrcMesg`.

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`.


Related Products

Intel® Integrated Performance Primitives (Intel® IPP)
Cryptography for Intel IPP is an add-on library for the main Intel IPP library, which provides a comprehensive set of application domain-specific highly optimized functions for signal processing, image and video processing, operations on small matrices, three-dimensional (3D) data processing and rendering. Search http://www.intel.com/software/products for more information.

Intel IPP Samples
An extensive library of code samples and codecs has been implemented using the Intel IPP functions to demonstrate the use of Intel IPP and to help accelerate the development of your applications, components, and codecs. The samples can be downloaded from www.intel.com/software/products/ipp/samples.htm.
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.

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

Function Context Structures

Some Intel IPP Cryptography functions use special structures to store function-specific (context) information. For example, the IppsRijndael128Spec structure stores a set of round keys, a set of round inverse keys, and key management information for the Rijndael cipher scheme with the block size equal to 128.

Two different kinds of context structures are used:
• Specification structures, which are not modified during the function's operation. Their names include the Spec suffix.
• State structures, which are modified during operation. Their names include the State suffix.

**Important**
It is your application that defines the life cycle of the context: initialization, updating, and destruction.

Context structures are initialized with the initialization functions. For example, the `ippsRijndael128CCMInit` function initializes the user-supplied memory as the `IppsRijndael128CCMState` context.

**See Also**
Data Security Considerations

## Data Security Considerations

IPP Cryptography functions use several types of buffers during operation, and some of them may contain sensitive information. These buffers may be reused multiple times, and there is no way for the underlying Intel IPP implementation to know when this data is no longer needed and sensitive information should be scrubbed from those buffers. Examples of sensitive information include but are not be limited to:

- Keys
- Initialization Vectors
- Context Structures

**Important**
If any such sensitive data is passed to Intel IPP, it is the responsibility of your application to scrub this information from the memory buffers.

**See Also**
Function Context Structures
Symmetric Cryptography
Primitive Functions

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: Rijndael [AES], including AES-CCM [NIST SP 800-38C] and AES-GCM [NIST SP 800-38D], Triple DES (TDES) [FIPS PUB 46-3], and SMS4 [SM4].
- 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

<table>
<thead>
<tr>
<th>Block Cipher Name</th>
<th>Data Block Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rijndael128 (AES)</td>
<td>128</td>
</tr>
<tr>
<td>TDES</td>
<td>64</td>
</tr>
</tbody>
</table>

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.

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,

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

only IppsCPPaddingNONE remains acceptable.
Rijndael Functions

Rijndael cipher scheme is an iterated block cipher with a variable block size and a variable key length. Rijndael functions with the 128-bit key length are, in fact, American Encryption Standard (AES) cipher functions implemented in the way to comply with the American Standard FIPS 197.

The AES functions use the IppsAESSpec context. This context serves as an operational vehicle 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 either encrypting or decrypting the streaming data with the specified padding scheme.

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 IppsAESSpec by calling the function AESGetSize.
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 AESGetSize.
3. Initialize the context IppsAESSpec*pCtx by calling the function AESInit with the allocated buffer and the respective 128-bit AES key.
4. Specify the initialization vector and the padding scheme, then call the function AESEncryptCBC to encrypt the input data stream using the AES encryption function with CBC mode.
5. Clean up secret data stored in the context.
6. Call the operating system memory free service function to release the buffer allocated for the context IppsAESSpec, if needed.

The IppsAESSpec context is position-dependent. The AESPack/AESUnpack function transforms the respective position-dependent context to a position-independent form and vice versa.

See Also
AES-CCM Functions
AES-GCM Functions
Data Security Considerations

AESGetSize

Gets the size of the IppsAESSpec context.

Syntax

IppStatus ippsAESGetSize(int* pSize);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSize Pointer to the IppsAESSpec context size value.

Description

The function gets the IppsAESSpec 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.

AESInit
*Initializes user-supplied memory as IppsAESSpec context for future use.*

Syntax
IppStatus ippsAESInit(const Ipp8u* pKey, int keylen, IppsAESSpec* pCtx, int ctxSize);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pKey**: Pointer to the AES key.
- **keylen**: Key byte stream length in bytes defined by the IppsRijndaelKeyLength enumerator.
- **pCtx**: Pointer to the buffer being initialized as IppsAESSpec context.
- **ctxSize**: Available size of the buffer being initialized.

Description
This function initializes the memory pointed by pCtx as IppsAESSpec. The key is used to provide all necessary key material for both encryption and decryption operations.

NOTE
If the pKey pointer is NULL, the function initializes the context with the zero key, which can help you to clean up the actual secret before releasing the context.

Return Values

ippStsNoErr
Indicates no error. Any other value indicates an error or warning.

ippStsNullPtrErr
Indicates an error condition if the pCtx pointer is NULL.

ippStsLengthErr
Returns an error condition if keyLen is not equal to 16, 24, or 32.

ippStsMemAllocErr
Indicates an error condition if the allocated memory is insufficient for the operation.

See Also
Data Security Considerations
**AESSetKey**

*Resets the AES secret key in the initialized IppsAESSpec context.*

**Syntax**

```c
IppStatus ippsAESSetKey(const Ipp8u* pKey, int keylen, IppsAESSpec* pCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- **pKey**: Pointer to the AES key.
- **keylen**: Length of the secret key.
- **pCtx**: Pointer to the initialized IppsAESSpec context.

**Description**

This function resets the AES secret key in the initialized IppsAESSpec context with the user-supplied secret key.

**NOTE**

If the `pKey` pointer is NULL, the function resets the context with the zero key, which can help you to clean up the actual secret before releasing the context.

**Return Values**

- **ippStsNoErr**: Indicates no error. Any other value indicates an error or warning.
- **ippStsNullPtrErr**: Indicates an error condition if the `pCtx` pointer is NULL.
- **ippStsLengthErr**: Returns an error condition if `keyLen` is not equal to 16, 24, or 32.

**See Also**

Data Security Considerations

**AESPack, AESUnpack**

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

**Syntax**

```c
IppStatus ippsAESPack (const IppsAESSpec* pCtx, Ipp8u* pBuffer, int bufSize);
IppStatus ippsAESUnpack (const Ipp8u* pBuffer, IppsAESSpec* pCtx, int ctxSize);
```
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\begin{itemize}
  \item \textit{pCtx} \hspace{1cm} Pointer to the IppsAESSpec context.
  \item \textit{pBuffer} \hspace{1cm} Pointer to the user-defined buffer.
  \item \textit{bufSize} \hspace{1cm} Available size of the buffer.
  \item \textit{ctxSize} \hspace{1cm} Available size of the context.
\end{itemize}

Description

The AESPack function transforms the *pCtx context to a position-independent form and stores it in the *pBuffer buffer. The AESUnpack function performs the inverse operation, that is, transforms the contents of the *pBuffer buffer into a normal IppsAESSpec context. The AESPack and AESUnpack functions enable replacing the position-dependent IppsAESSpec context in the memory.

Call the AESGetSize function prior to AESPack/AESUnpack to determine the size of the buffer.

Return Values

\begin{itemize}
  \item ippStsNoErr \hspace{1cm} Indicates no error. Any other value indicates an error or warning.
  \item ippStsNullPtrErr \hspace{1cm} Indicates an error condition if any of the specified pointers is NULL.
  \item ippStsLengthErr \hspace{1cm} Indicates an error condition if bufSize or ctxSize is less than the real size of the IppsAESSpec context.
\end{itemize}

\textbf{AESEncrypt ECB}

\textit{Encrypts plaintext message by using ECB encryption mode.}

\textbf{Syntax}

IppStatus ippsAESEncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int srclen, const IppsAESSpec* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\begin{itemize}
  \item \textit{pSrc} \hspace{1cm} Pointer to the input plaintext data stream of variable length.
\end{itemize}
Description
The function encrypts the input data stream of a variable length according to the cipher scheme 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 input data stream length is less than or equal to zero.
- ippStsUnderRunErr: Indicates an error condition if srclen is not divisible by cipher block size.
- ippStsContextMatchErr: Indicates an error condition if the context parameter does not match the operation.

AESDecryptECB
Decrpts byte data stream by using the AES algorithm in the ECB mode.

Syntax
IppStatus ippsAESDecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsAESSpec* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- pSrc: Pointer to the input ciphertext data stream of variable length.
- pDst: Pointer to the resulting plaintext data stream of variable length.
- srclen: Length of the ciphertext data stream in bytes.
- pCtx: Pointer to the IppsAESSpec context.

Description
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.

AESEncryptCBC

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

Syntax

IppStatus ippsAESEncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsAESSpec* pCtx, const Ipp8u* pIV);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

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 length in bytes.
pCtx: Pointer to the IppsAESSpec context.
pIV: Pointer to the initialization vector for the CBC mode operation.

Description

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.

ippStsLengthErr: Indicates an error condition if the input data stream length is less than or equal to zero.
**AESDecryptCBC**

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

**Syntax**

IppStatus ippsAESDecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, const IppsAESSpec* pCtx, const Ipp8u* pIV);

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pSrc**
  Pointer to the input ciphertext data stream.
- **pDst**
  Pointer to the resulting plaintext data stream of the variable length.
- **srclen**
  Length of the ciphertext data stream length in bytes.
- **pCtx**
  Pointer to the IppsAESSpec context.
- **pIV**
  Pointer to the initialization vector for CBC mode operation.

**Description**

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**

- **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.

**AESEncryptCFB**

Encrypts byte data stream according to AES in the CFB mode.
Syntax
IppStatus ippsAESEncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, int cfbBlkSize, const IppsAESSpec* pCtx, const Ipp8u* pIV);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

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.
cfbBlkSize Size of the CFB block in bytes.
pCtx Pointer to the IppsAESSpec context.
pIV Pointer to the initialization vector for the CFB mode operation.

Description
The function encrypts the input data stream of variable length according to the CFB 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 input data stream length is less than or equal to zero.
ippStsUnderRunErr Indicates an error condition if srcLen is not divisible by cfbBlkSize parameter value.
ippStsCFBSizeErr Indicates an error condition if the value for cfbBlkSize is illegal.
ippStsContextMatchErr Indicates an error condition if the context parameter does not match the operation.

AESDecryptCFB
Decrypts byte data stream according to AES in CFB mode.

Syntax
IppStatus ippsAESDecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int cfbBlkSize, const IppsAESSpec* pCtx, const Ipp8u* pIV);
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\textit{pSrc} \\
Pointer to the input ciphertext data stream.

\textit{pDst} \\
Pointer to the resulting plaintext data stream of variable length.

\textit{srclen} \\
Length of the ciphertext data stream in bytes.

\textit{cfbBlkSize} \\
Size of the CFB block in bytes.

\textit{pCtx} \\
Pointer to the \texttt{IppsAES} context.

\textit{pIV} \\
Pointer to the initialization vector for the CFB mode operation.

Description

The function decrypts the input data stream of variable length according to the CFB mode as specified in [NIST SP 800-38A].

Return Values

\texttt{ippStsNoErr} \\
Indicates no error. Any other value indicates an error or warning.

\texttt{ippStsNullPtrErr} \\
Indicates an error condition if any of the specified pointers is NULL.

\texttt{ippStsLengthErr} \\
Indicates an error condition if the output data stream length is less than or equal to zero.

\texttt{ippStsCFBSizeErr} \\
Indicates an error condition if the value for \textit{cfbBlkSize} is illegal.

\texttt{ippStsContextMatchErr} \\
Indicates an error condition if the context parameter does not match the operation.

\texttt{ippStsUnderRunErr} \\
Indicates an error condition if \textit{srclen} is not divisible by cipher block size.

\texttt{AESEncryptOFB} \\
\textit{Encrypts a variable length data stream according to AES in the OFB mode.}

Syntax

\begin{lstlisting}
IppStatus ippsAESEncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int cfbBlkSize, const IppsAESSpec* pCtx, Ipp8u* pIV);
\end{lstlisting}

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

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.
- **ofbBlkSize**: Size of the OFB block in bytes.
- **pCtx**: Pointer to the IppsAESSpec context.
- **pIV**: Pointer to the initialization vector for the OFB mode operation.

Description

The function encrypts the input data stream of a variable length in the OFB 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 input data stream length is less than or equal to zero.
- **ippStsUnderRunErr**: Indicates an error condition if srclen is not divisible by the ofbBlkSize parameter value.
- **ippStsOFBSizeErr**: Indicates an error condition if the value of ofbBlkSize is illegal.
- **ippStsContextMatchErr**: Indicates an error condition if the context parameter does not match the operation.

AESDecryptOFB

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

Syntax

```c
IppStatus ippsAESDecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int srclen, int ofbBlkSize, const IppsAESSpec* pCtx, Ipp8u* pIV);
```

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pSrc**: Pointer to the input ciphertext data stream of variable length.
- **pDst**: Pointer to the resulting plaintext data stream.
srclen | Length of the ciphertext data stream in bytes.
ofbBlkSize | Size of the OFB block in bytes.
pCtx | Pointer to the IppsAESSpec context.
pIV | Pointer to the initialization vector for the OFB mode operation.

Description
The function decrypts the input data stream of a variable length in the OFB 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 input data stream length is less than or equal to zero.
ippStsUnderRunErr | Indicates an error condition if srclen is not divisible by the ofbBlkSize parameter value.
ippStsOFBSizeErr | Indicates an error condition if the value of ofbBlkSize is illegal.
ippStsContextMatchErr | Indicates an error condition if the context parameter does not match the operation.

AESEncryptCTR
Encrypts a variable length data stream in the CTR mode.

Syntax
IppStatus ippsAESEncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen,const IppsAESSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pSrc | Pointer to the input plaintext data stream of a variable length.
pNext | Pointer to the resulting ciphertext data stream.
srcLen | Length of the plaintext data stream in bytes.
pCtx | Pointer to the IppsAESSpec context.
pCtrValue | Pointer to the counter data block.
ctrNumBitSize | Number of bits in the specific part of the counter to be incremented.
Description
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
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 or equal to zero.
ippStsCTRSIZEErr Indicates an error condition if the value of the ctrNumBitSize is illegal.
ippStsContextMatchErr Indicates an error condition if the context parameter does not match the operation.

AESDecryptCTR
Decrypts a variable length data stream in the CTR mode.

Syntax
IppStatus ippsAESDecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int srcLen, const IppsAESSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

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

Description
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
ippStsNoErr Indicates no error. Any other value indicates an error or warning.
Example of Using AES Functions

AES Encryption and Decryption

```c
// use of the CTR mode
int AES_sample(void)
{
    // secret key
    Ipp8u key[] = "\x00\x01\x02\x03\x04\x05\x06\x07"
    "\x08\x09\x10\x11\x12\x13\x14\x15";
    // define and setup AES cipher
    int ctxSize;
    ippsAESGetSize(&ctxSize);
    IppsAESSpec* pAES = (IppsAESSpec*)( new Ipp8u [ctxSize] ) ;
    ippsAESInit(key, sizeof(key)-1, pAES, ctxSize);

    // message to be encrypted
    Ipp8u msg[] = "the quick brown fox jumps over the lazy dog";
    // and initial counter
    Ipp8u ctr0[] = "\xff\xee\xdd\xcc\xbb\xaa\x99\x88"
    "\x77\x66\x55\x44\x33\x22\x11\x00";

    // counter
    Ipp8u ctr[16];

    // init counter before encryption
    memcpy(ctr, ctr0, sizeof(ctr));
    // encrypted message
    Ipp8u ctext[sizeof(msg)];
    // encryption
    ippsAESEncryptCTR(msg, ctext, sizeof(msg), pAES, ctr, 64);

    // init counter before decryption
    memcpy(ctr, ctr0, sizeof(ctr));
    // decrypted message
    Ipp8u rtext[sizeof(ctrtext)];
    // decryption
    ippsAESDecryptCTR(ctext, rtext, sizeof(ctrtext), pAES, ctr, 64);

    // remove secret and release resource
    ippsAESInit(0, sizeof(key)-1, pAES, ctxSize);
    delete [] (Ipp8u*)pAES;

    int error = memcmp(rtext, msg, sizeof(msg));
    return 0==error;
}
```
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.

The AES-CCM functions enable authenticated encryption/decryption of several messages using one key that the AES_CCMInit function sets. Processing of each new message starts with a call to the AES_CCMStart 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 IppsAES_CCMState by calling the function AES_CCMGetSize.
2. Call the system memory-allocation service function to allocate a buffer whose size is not less than the function AES_CCMGetSize specifies.
3. Initialize the context IppsAES_CCMState*pCtx by calling the function AES_CCMInit with the allocated buffer and respective AES key.
4. Optionally call AES_CCMMessageLen and/or AES_CCMTagLen to set up message and tag parameters.
5. Call AES_CCMStart to start authenticated encryption of the first/next message.
6. Keep calling AES_CCMEncrypt until the entire message is processed.
7. Request the authentication tag by calling AES_CCMGetTag.
8. Proceed to the next message, if any, that is, go to step 5.
9. Clean up secret data stored in the context.
10. Call the system memory free service function to release the buffer allocated for the context IppsAES_CCMState, if needed.

See Also
Data Security Considerations

AES_CCMGetSize

Gets the size of the IppsAES_CCMState context.

Syntax

IppStatus ippsAES_CCMGetSize(int* pSize);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSize          Pointer to the size of the IppsAES_CCMState context.

Description

The function gets the size of the IppsAES_CCMState context in bytes and stores it in *pSize.

Return Values

ippStsNoErr     Indicates no error. Any other value indicates an error or warning.
AES_CCMInit

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

Syntax

IppStatus ippsAES_CCMInit(const Ipp8u* pKey, int keyLen, IppsAES_CCMState* pState, int ctxSize);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pKey Pointer to the secret key.
keyLen Length of the secret key.
pState Pointer to the buffer being initialized as IppsAES_CCMState context.
ctxSize Size of the buffer being initialized.

Description

The function initializes the memory pointed by pState as the IppsAES_CCMState context. In addition, the function uses the initialization variable and additional authenticated data to provide all necessary key material for both encryption and decryption.

NOTE
If the pKey pointer is NULL, the function initializes the context with the zero key, which can help you to clean up the actual secret before releasing the context.

Return Values

ippStsNoErr Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr Indicates an error condition if the pState pointer is NULL.
ippStsLengthErr Indicates an error condition an error condition if keyLen is not equal to 16, 24, or 32.
ippStsMemAllocErr Indicates an error condition if the allocated memory is insufficient for the operation.

See Also

Data Security Considerations
AES_CCMStart

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

Syntax

IppStatus ippsAES_CCMStart(const Ipp8u* pIV, int ivLen, const Ipp8u* pAAD, int aadLen, IppsAES_CCMState* pState);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pIV Pointer to the initialization vector.
ivLen Length of the initialization vector *pIV (in bytes).
pAAD Pointer to the additional authenticated data.
aadLen Length of additional authenticated data *pAAD (in bytes).
pState Pointer to the IppsAES_CCMState context.

Description

The function resets internal counters and buffers of the *pState context.

Return Values

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

AES_CCMEncrypt

Encrypts a data buffer in the CCM mode.

Syntax

IppStatus ippsAES_CCMEncrypt(const Ipp8u* pSrc, Ipp8u* pDst, int len, IppsAES_CCMState* pState);

Include Files

ippcp.h
Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- pSrc: Pointer to the input plaintext data stream of a variable length.
- pdst: Pointer to the resulting ciphertext data stream.
- len: Length of the plaintext and ciphertext data stream in bytes.
- pState: Pointer to the IppsAES_CCMState context.

Description

The function encrypts the input data stream of a variable length in the CCM mode as specified in [NIST SP 800-38C].

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.

AES_CCMDecrypt

Decrypts a data buffer in the CCM mode.

Syntax

IppStatus ippsAES_CCMDecrypt(const Ipp8u* pSrc, Ipp8u* pdst, int len, IppsAES_CCMState* pState);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- pSrc: Pointer to the input ciphertext data stream of variable length.
- pdst: Pointer to the resulting plaintext data stream.
- len: Length of the plaintext and ciphertext data stream in bytes.
- pState: Pointer to the IppsAES_CCMState context.

Description

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

- **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.

### AES_CCMGetTag

*Generates the message authentication tag in the CCM mode.*

#### Syntax

```c
IppStatus ippsAES_CCMGetTag (Ipp8u* pTag, int tagLen, const IppsAES_CCMState* pState);
```

#### Include Files

- ippcp.h

#### Domain Dependencies

**Headers:** ippcore.h  
**Libraries:** ippcore.lib

#### Parameters

- **pTag**: Pointer to the authentication tag.  
- **tagLen**: Length of the authentication tag *pTag* (in bytes).  
- **pState**: Pointer to the IppsAES_CCMState context.

#### Description

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

- **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.
- **ippStsLengthErr**: Indicates an error condition if `tagLen < 1` or `tagLen` does not exceed the tag length specified in the previous call to `AES_CCMStart`.

### AES_CCMMessageLen

*Sets up the length of the message to be processed.*
### Syntax

```c
IppStatus ippsAES_CCMMessageLen(Ipp64u msgLen, IppsAES_CCMState* pState);
```

### Include Files

`ippcp.h`

### Domain Dependencies

**Headers:** `ippcore.h`

**Libraries:** `ippcore.lib`

### Parameters

- **msgLen**
  - Length of the message to be processed (in bytes).

- **pState**
  - Pointer to the `IppsAES_CCMState` context.

### Description

The function assigns the value of `msgLen` to the length of the message to be processed in the `*pState` 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.

- **ippStsContextMatchErr**
  - Indicates an error condition if the context parameter does not match the operation.

- **ippStsLengthErr**
  - Indicates an error condition if `msgLen`=0.

### AES_CCMTagLen

Sets up the length of the required authentication tag.

### Syntax

```c
IppStatus ippsAES_CCMTagLen(int tagLen, IppsAES_CCMState* pState);
```

### Include Files

`ippcp.h`

### Domain Dependencies

**Headers:** `ippcore.h`

**Libraries:** `ippcore.lib`

### Parameters

- **tagLen**
  - Length of the required authentication tag (in bytes).

- **pState**
  - Pointer to the `IppsAES_CCMState` context.
Description
The function assigns the value of tagLen to the length of the required authentication tag in the *pState 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.
ippStsContextMatchErr  Indicates an error condition if the context parameter does not match the operation.
ippStsLengthErr  Indicates an error condition if tagLen < 4 or tagLen > 16 or taglen 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.

The AES-GCM function set includes incremental functions, which enable authenticated encryption/decryption of several messages using one key. 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 IppsAES_GCMState by calling the function AES_GCMGetSize.
2. Call the system memory-allocation service function to allocate a buffer whose size is not less than the function AES_GCMGetSize specifies.
3. Initialize the context IppsAES_GCMState*pCtx by calling the function AES_GCMInit with the allocated buffer and the respective AES key.
4. Call AES_GCMStart to start authenticated encryption of the first/next message.
5. Keep calling AES_GCMEncrypt until the entire message is processed.
6. Request the authentication tag by calling AES_GCMGetTag.
7. Proceed to the next message, if any, that is, go to step 4.
8. Clean up secret data stored in the context.
9. Call the system memory free service function to release the buffer allocated for the context IppsAES_GCMState, if needed.

If the size of the initial vector and/or additional authenticated data (IV and AAD parameters of the AES_GCMStart 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 AES_GCMReset to prepare the IppsAES_GCMState context for authenticated encryption of the first/new message.
2. Keep calling AES_GCMProcessIV for successive parts of IV until the entire IV is processed.
3. Keep calling AES_GCMProcessAAD for successive parts of AAD until the entire AAD is processed.
See Also
Data Security Considerations

AES_GCMGetSize

*Gets the size of the IppsAES_GCMState context for use of the AES-GCM implementation with the specified characteristics.*

**Syntax**

```c
IppStatus ippsAES_GCMGetSize(int* pSize);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

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

**Description**

The function gets the size of the IppsAES_GCMState context (in bytes) and stores the size 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.

AES_GCMInit

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

**Syntax**

```c
IppStatus ippsAES_GCMInit(const Ipp8u* pKey, int keyLen, IppsAES_GCMState* pState, int ctxSize);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

`pKey`  
Pointer to the secret key.
**Description**

The function initializes the memory pointed by `pState` as the `IppsAES_GCMState` context. In addition, the function uses the initialization variable and additional authenticated data to provide all necessary key material for both encryption and decryption.

Call the `AES_GCMGetSize` function prior to `AES_GCMInit` 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 the `pState` pointer is `NULL`.
- **ippStsLengthErr**
  - Indicates an error condition if `keyLen` is not equal to 16, 24, or 32.
- **ippStsMemAllocErr**
  - Indicates an error condition if the allocated memory is insufficient for the operation.

---

### See Also

**AES_GCMStart**

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

---

### Syntax

```c
IppStatus ippsAES_GCMStart(const Ipp8u* pIV, int ivLen, const Ipp8u* pAAD, int aadLen, IppsAES_GCMState* pState);
```

### Include Files

`ippcp.h`

---

### Domain Dependencies

**Headers:**  `ippcore.h`

**Libraries:**  `ippcore.lib`

---

### Parameters

- **pIV**
  - Pointer to the initialization vector.
- **ivLen**
  - Length of the initialization vector `*pIV` (in bytes).
- **pAAD**
  - Pointer to the additional authenticated data.
**aadLen**
Length of additional authenticated data \( p_{AAD} \) (in bytes).

**pState**
Pointer to the **IppsAES_GCMState** context.

**Description**
The function resets internal counters and buffers of the \( *pState \) 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.
- **ippStsContextMatchErr**
  Indicates an error condition if the context parameter does not match the operation.
- **ippStsLengthErr**
  Indicates an error condition if the length of the initialization vector is zero.

**AES_GCMReset**
*Resets the IppsAES_GCMState context for authenticated encryption/decryption of a new message.*

**Syntax**

```c
IppStatus ippsAES_GCMReset(IppsAES_GCMState* pState);
```

**Include Files**
ippcp.h

**Domain Dependencies**
Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**
- **pState**
  Pointer to the **IppsAES_GCMState** context.

**Description**
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**
- **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.
AES_GCMProcessIV

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

Syntax

IppStatus ippsAES_GCMProcessIV(const Ipp8u* pIV, int ivLen, IppsAES_GCMState* pState);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pIV Pointer to the initialization vector.
ivLen Length of the initialization vector *pIV (in bytes).
pState Pointer to the IppsAES_GCMState context.

Description

The function processes \textit{ivLen} bytes of the initial vector \textit{pIV} as specified in [NIST SP 800-38D].

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.
ippStsLengthErr Indicates an error condition if the length of the initialization vector is zero.

AES_GCMProcessAAD

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

Syntax

IppStatus ippsAES_GCMProcessAAD(const Ipp8u* pAAD, int aadLen, IppsAES_GCMState* pState);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib
Parameters

- **pAAD**: Pointer to the additional authenticated data.
- **aadLen**: Length of additional authenticated data *pAAD* (in bytes).
- **pState**: Pointer to the IppsAES_GCMState context.

Description

The function processes `aadLen` bytes of additional authenticated data *pAAD* as specified in [NIST SP 800-38D].

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.

AES_GCMEncrypt

Encrypts a data buffer in the GCM mode.

Syntax

```c
IppStatus ippsAES_GCMEncrypt(const Ipp8u* pSrc, Ipp8u* pDst, int len, IppsAES_GCMState* pState);
```

Include Files

- ippcp.h

Domain Dependencies

- Headers: ippcore.h
- Libraries: ippcore.lib

Parameters

- **pSrc**: Pointer to the input plaintext data stream of a variable length.
- **pDst**: Pointer to the resulting ciphertext data stream.
- **len**: Length of the plaintext and ciphertext data stream in bytes.
- **pState**: Pointer to the IppsAES_GCMState context.

Description

The function encrypts the input data stream of a variable length according to GCM as specified in [NIST SP 800-38D].

Return Values

- **ippStsNoErr**: Indicates no error. Any other value indicates an error or warning.
 Indicates an error condition if any of the specified pointers is NULL.

Indicates an error condition if the context parameter does not match the operation.

**AES_GCMDecrypt**

*Decrypts a data buffer in the GCM mode.*

**Syntax**

```c
IppStatus ippsAES_GCMDecrypt (const Ipp8u* pSrc, Ipp8u* pDst, int len, IppsAES_GCMState* pState);
```

**Include Files**

ippcp.h

**Domain Dependencies**

**Headers:** ippcore.h

**Libraries:** ippcore.lib

**Parameters**

- **pSrc**: Pointer to the input ciphertext data stream of a variable length.
- **pDst**: Pointer to the resulting plaintext data stream.
- **len**: Length of the plaintext and ciphertext data stream in bytes.
- **pState**: Pointer to the IppsAES_GCMState context.

**Description**

The function decrypts the input cipher data stream of a variable length according to GCM as specified in [NIST SP 800-38D].

**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.

**AES_GCMGetTag**

*Generates the authentication tag in the GCM mode.*

**Syntax**

```c
IppStatus ippsAES_GCMGetTag (Ipp8u* pTag, int tagLen, const IppsAES_GCMState* pState);
```

**Include Files**

ippcp.h
Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pTag**: Pointer to the authentication tag.
- **tagLen**: Length of the authentication tag *pTag (in bytes).
- **pState**: Pointer to the IppsAES_GCMState context.

Description

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

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.
- ippStsLengthErr: Indicates an error condition if tagLen < 1 or taglen > 16.

TDES Functions

The Triple Data Encryption Algorithm (TDEA) is a revised symmetric algorithm scheme built on the Data Encryption Standard (DES) system. The Triple DES (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. While AES (Rijndael) is preferred, TDEA is an approved cipher. Use implementations of AES where possible. In cases where using AES is impossible or inconvenient, use TDES functions.

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 TDES cipher systems.

**NOTE**

Intel IPP functions for cryptography do not allocate memory internally. TheGetSize function does not require allocated memory. You need to call theGetSize function 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 TDESEncryptECB function operates under the ECB mode.

The encryption function TDESEncryptCBC operates under the CBC mode using its cipher scheme and requires to have an initialization vector iv. Since there are a number of ways to initialize the initialization vector iv, you should remember which of them you used to be able to decrypt the message when needed.
The encryption function \texttt{TDESEncryptCFB} operates under the CFB mode using its cipher scheme and requires having the initialization vector \texttt{pIV} and CFB block size \texttt{cfbBlkSize}.

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

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 \texttt{IppsDESSpec} by calling the function \texttt{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 \texttt{DESGetSize}. Initialize pointers to contexts \texttt{pCtx1}, \texttt{pCtx2}, and \texttt{pCtx3} by calling the function \texttt{DESInit} three times, each with the allocated buffer and the respective DES key.
3. Specify the initialization vector and then call the function \texttt{TDESEncryptCBC} to encrypt the input data stream under CBC mode using TDES scheme.
4. Clean up secret data stored in the contexts.
5. 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.

\textbf{NOTE}
Similar procedure can be applied for ECB, CFB, and CTR mode operation.

The \texttt{IppsDESSpec} context is position-dependent. The \texttt{DESPack/DESUnpack} functions transform the position-dependent context to a position-independent form and vice versa.

\section*{See Also}
\textbf{Data Security Considerations}

\subsection*{DESGetSize}
\textit{Gets the size of the IppsDESSpec context.}

\textbf{Syntax}
\begin{verbatim}
IppStatus ippsDESGetSize(int* pSize);
\end{verbatim}

\textbf{Include Files}
\begin{verbatim}
 IPPCP.H
\end{verbatim}

\textbf{Domain Dependencies}
Headers: \texttt{ippcore.h}
Libraries: \texttt{ippcore.lib}

\textbf{Parameters}
\begin{verbatim}
pSize Pointer to the IppsDESSpec context size value.
\end{verbatim}

\textbf{Description}
This function gets the \texttt{IppsDESSpec} context size in bytes and stores it in \texttt{*pSize}.

\textbf{Return Values}
\begin{verbatim}
ippStsNoErr Indicates no error. Any other value indicates an error or warning.
\end{verbatim}
Indicates an error condition if any of the specified pointers is NULL.

**DESInit**

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

**Syntax**

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

**Include Files**

`ippcp.h`

**Domain Dependencies**

- **Headers:** `ippcore.h`
- **Libraries:** `ippcore.lib`

**Parameters**

- `pKey` Pointer to the DES key.
- `pCtx` Pointer to the IppsDESSpec context being initialized.

**Description**

This 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**

- `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.

**See Also**

Data Security Considerations

**DESPack, DESUnpack**

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

**Syntax**

```c
IppStatus ippsDESPack (const IppsDESSpec* pCtx, Ipp8u* pBuffer);
IppStatus ippsDESUnpack (const Ipp8u* pBuffer, IppsDESSpec* pCtx);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

- **Headers:** `ippcore.h`
- **Libraries:** `ippcore.lib`
Parameters

pCtx
Pointer to the IppsDESSpec context.

pBuffer
Pointer to the user-defined buffer.

Description

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

Call the DESGetSize function prior to DESPack/DESUnpack 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.

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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc
Input plaintext data stream of a variable length.

pDst
Resulting ciphertext data stream.

srclen
Input data stream length in bytes.

pCtx1
First set of round keys scheduled for TDES internal operations.

pCtx2
Second set of round keys scheduled for TDES internal operations.

pCtx3
Third set of round keys scheduled for TDES internal operations.

padding
IppsPaddingNONE padding scheme.

Description

This 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.
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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

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

Description

This 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

ippStsNoErr
Indicates no error. Any other value indicates an error or warning.
TDESEncryptCBC

Encrypts variable length data stream in the CBC mode.

Syntax

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

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc
Input plaintext data stream of a variable length.

pDst
Resulting ciphertext data stream.

pIV
Initialization vector for TDES CBC mode operation.

srclen
Input data stream length in bytes.

pCtx1
First set of round keys scheduled for TDES internal operations.

pCtx2
Second set of round keys scheduled for TDES internal operations.

pCtx3
Third set of round keys scheduled for TDES internal operations.

padding
IppsCPPaddingNONE padding scheme.

Description

This 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

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 or equal to zero.
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);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

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

Description
This 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
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 decrypted plaintext data stream length is less than or equal to zero.
ippStsContextMatchErr
Indicates an error condition if the context parameter does not match the operation.
**TDESEncryptCFB**

*Encrypts variable length data stream in the CFB mode.*

**Syntax**

```c
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);
```

**Include Files**

ippcp.h

**Domain Dependencies**

*Headers:* ippcore.h

*Libraries:* ippcore.lib

**Parameters**

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

**Description**

This 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**

- 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 or equal to zero.
- ippStsUnderRunErr Indicates an error condition if *srclen* is not divisible by *cfbBlkSize* parameter value.
- ippStsCFBSIZEErr Indicates an error condition if the value for *cfbBlkSize* is illegal.
Indicates an error condition if the context parameter does not match the operation.

**TDESDecryptCFB**

Decrypts variable length data stream in the CFB mode.

**Syntax**

```c
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);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

**Headers:** `ippcore.h`

**Libraries:** `ippcore.lib`

**Parameters**

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

**Description**

This 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**

- `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 decrypted plaintext data stream length is less than or equal to zero.
- `ippStsCFBSizeErr` Indicates an error condition if the value for `cfbBlkSize` is illegal.
Indicates an error condition if the context parameter does not match the operation.

Indicates an error condition if \( srcLen \) is not divisible by cipher block size.

**TDESEncryptOFB**

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

**Syntax**

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

**Include Files**

`ippcp.h`

**Domain Dependencies**

*Headers:* `ippcore.h`

*Libraries:* `ippcore.lib`

**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.
- `ofbBlkSize`
  Size of the OFB block in bytes.
- `pCtx1`
  First set of round keys scheduled for TDES internal operations.
- `pCtx2`
  Second set of round keys scheduled for TDES internal operations.
- `pCtx3`
  Third set of round keys scheduled for TDES internal operations.
- `pIV`
  Pointer to the initialization vector for the OFB mode operation.

**Description**

This function encrypts the input data stream of a variable length in the OFB 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 input data stream length is less than or equal to zero.
- `ippStsUnderRunErr`
  Indicates an error condition if \( srcLen \) is not divisible by the \( ofbBlkSize \) parameter value.
- `ippStsOFBSizeErr`
  Indicates an error condition if the value of \( ofbBlkSize \) is illegal.
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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

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

Description

This function decrypts the input data stream of a variable length in the OFB 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 input data stream length is less than or equal to zero.
ippStsUnderRunErr
    Indicates an error condition if srclen is not divisible by the ofbBlkSize parameter value.
ippStsOFBSizeErr
    Indicates an error condition if the value of ofbBlkSize is illegal.
ippStsContextMatchErr
    Indicates an error condition if the context parameter does not match the operation.
**TDESEncryptCTR**

Encrypted 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);

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

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

**Description**

This 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**

- **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 or equal to zero.
- **ippStsCTRSizeErr**
  Indicates an error condition if the value of the `ctrNumBitSize` is illegal.
- **ippStsContextMatchErr**
  Indicates an error condition if the context parameter does not match the operation.
**TDESDecryptCTR**

*Decrypts a variable length data stream in the CTR mode.*

**Syntax**

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

**Include Files**

ippcp.h

**Domain Dependencies**

*Headers:* ippcore.h

*Libraries:* ippcore.lib

**Parameters**

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

**Description**

This 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**

- **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 decrypted plaintext data stream length is less than or equal to zero.
- **ippStsCTRSizeErr**: Indicates an error condition if the value of the `ctrNumBitSize` is illegal.
- **ippStsContextMatchErr**: Indicates an error condition if the context parameter does not match the operation.
Example of Using TDES Functions

TDES Encryption and Decryption

// Use of the ECB mode
void TDES_sample(void)
{
    // size of the TDES algorithm block is equal to 8
    const int tdesBlkSize = 8;

    // get size of the context needed for the encryption/decryption operation
    int ctxSize;
    ippsDESGetSize(&ctxSize);
    // and allocate one
    IppsDESSpec* pCtx1 = (IppsDESSpec*)(new Ipp8u [ctxSize] ) ;
    IppsDESSpec* pCtx2 = (IppsDESSpec*)(new Ipp8u [ctxSize] ) ;
    IppsDESSpec* pCtx3 = (IppsDESSpec*)(new Ipp8u [ctxSize] ) ;

    // define the key
    Ipp8u key1[] = {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
    Ipp8u key2[] = {0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18};
    Ipp8u key3[] = {0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28};
    Ipp8u keyX[] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

    // and prepare the context for the TDES usage
    ippsDESInit(key1, pCtx1);
    ippsDESInit(key2, pCtx2);
    ippsDESInit(key3, pCtx3);

    // define the message to be encrypted
    Ipp8u ptext[] = {"the quick brown fox jumps over the lazy dog"};
    // allocate enough memory for the ciphertext
    // note that
    // the size of ciphertext is always a multiple of the 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
    ippsTDESEncryptECB(ptext, ctext, sizeof(ctext), pCtx1, pCtx2, pCtx3, 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
    ippsTDESDecryptECB(ctext, rtext, sizeof(ctext), pCtx1, pCtx2, pCtx3, IppsCPPaddingNONE);

    // remove actual secret from contexts
    ippsDESInit(keyX, pCtx1);
    ippsDESInit(keyX, pCtx2);
    ippsDESInit(keyX, pCtx3);
    // release resources
    delete (Ipp8u*)pCtx1;
    delete (Ipp8u*)pCtx2;
    delete (Ipp8u*)pCtx3;
}
SMS4 Functions

You can use the functions described in this section for various operational modes of SMS4 cipher systems [SM4].

Intel IPP for cryptography supports ECB, CBC, CFB, CTR, and OFB modes. You can tell which algorithm a given function supports from the function base name, for example, the SMS4EncryptECB function operates under the ECB mode.

All functions for the SMS4 block cipher use the context IppsSMS4Spec, which serves as an operational vehicle to carry the material required for various modes of operation.

Application code for conducting a typical encryption under the CBC mode using the SMS4 scheme must perform the following sequence of operations:

1. Get the size required to configure the context IppsSMS4Spec by calling the function SMS4GetSize.
2. Call an operating system memory allocation service function to allocate a buffer of size not less than the one specified by the function SMS4GetSize.
3. Initialize the pointer to the context by calling the function SMS4Init.
4. Specify the initialization vector and then call the function SMS4EncryptCBC to encrypt the input data stream under CBC mode using SMS4 scheme.
5. Clean up secret data stored in the context.
6. Free the memory allocated to the buffer once SMS4 encryption under the CBC mode has been completed.

**NOTE**
You can apply a similar procedure to ECB, CFB, CTR, and OFB modes of operation.

A similar scheme also holds for decryption.

See Also

Data Security Considerations

SMS4GetSize

*Gets the size of the IppsSMS4Spec context.*

**Syntax**

IppStatus ippsSMS4GetSize(int* pSize);

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

*pSize*  
Pointer to the IppsSMS4Spec context size value.

**Description**

The function gets the IppsSMS4Spec 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.

SMS4Init

Initializes user-supplied memory as IppsSMS4Spec context for future use.

Syntax

IppStatus ippsSMS4Init(const Ipp8u* pKey, int keyLen, IppsSMS4Spec* pCtx, int ctxSize);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pKey Pointer to the SMS4 key.
keyLen Key byte stream length. Must equal 16.
pCtx Pointer to the buffer being initialized as IppsSMS4Spec context.
ctxSize Available size of the buffer being initialized.

Description

This function initializes the memory pointed by pCtx as IppsSMS4Spec. The key is used to provide all necessary key material for both encryption and decryption operations.

NOTE

If the pKey pointer is NULL, the function initializes the context with the zero key, which can help you to clean up the actual secret before releasing the context.

Return Values

ippStsNoErr Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr Indicates an error condition if the pCtx pointer is NULL.
ippStsLengthErr Returns an error condition if keyLen is not equal to 16.
ippStsMemAllocErr Indicates an error condition if the allocated memory is insufficient for the operation.

See Also

Data Security Considerations
SMS4SetKey

Resets the SMS4 secret key in the initialized IppsSMS4Spec context.

Syntax

IppStatus ippsSMS4SetKey(const Ipp8u *pKey, int keyLen, IppsSMS4Spec* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pKey Pointer to the SMS4 key.
keyLen Length of the secret key.
pCtx Pointer to the initialized IppsSMS4Spec context.

Description

This function resets the SMS4 secret key in the initialized IppsSMS4Spec context with the user-supplied secret key.

NOTE

If the pKey pointer is NULL, the function resets the context with the zero key, which can help you to clean up the actual secret before releasing the context.

Return Values

ippStsNoErr Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr Indicates an error condition if the pCtx pointer is NULL.
ippStsLengthErr Returns an error condition if keyLen is not equal to 16.

See Also

Data Security Considerations

SMS4EncryptECB

Encrypts plaintext message by using ECB encryption mode.

Syntax

IppStatus ippsSMS4EncryptECB(const Ipp8u *pSrc, Ipp8u *pDst, int len, const IppsSMS4Spec* pCtx);

Include Files

ippcp.h
Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc  
Pointer to the input plaintext data stream of variable length.
pDst  
Pointer to the resulting ciphertext data stream.
len  
Length of the input plaintext data in bytes.
pCtx  
Pointer to the IppsSMS4Spec context.

Description

The function encrypts the input data stream of a variable length according to the cipher scheme 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 input data stream length is less than or equal to zero.
ippStsUnderRunErr  
Indicates an error condition if len is not divisible by cipher block size.
ippStsContextMatchErr  
Indicates an error condition if the context parameter does not match the operation.

SMS4DecryptECB

Decrypts byte data stream by using the SMS4 algorithm in the ECB mode.

Syntax

IppStatus ippsSMS4DecryptECB(const Ipp8u* pSrc, Ipp8u* pDst, int len, const IppsSMS4Spec* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc  
Pointer to the input ciphertext data stream of variable length.
pDst  
Pointer to the resulting plaintext data stream of variable length.
len  
Length of the ciphertext data stream in bytes.
Description
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 len is not divisible by cipher block size.

SMS4EncryptCBC
Encrypts byte data stream according to SMS4 in the CBC mode.

Syntax
IppStatus ippsSMS4EncryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int len, const IppsSMS4Spec* pCtx, const Ipp8u* pIV);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pSrc Pointer to the input plaintext data stream of variable length.
pDst Pointer to the resulting ciphertext data stream.
len Length of the plaintext data stream length in bytes.
pCtx Pointer to the IppsSMS4Spec context.
pIV Pointer to the initialization vector for the CBC mode operation.

Description
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.
- **ippStsLengthErr**: Indicates an error condition if the input data stream length is less than or equal to zero.
- **ippStsUnderRunErr**: Indicates an error condition if \( len \) is not divisible by data block size.
- **ippStsContextMatchErr**: Indicates an error condition if the context parameter does not match the operation.

**SMS4DecryptCBC**

*Decrypts byte data stream according to SMS4 in the CBC mode.*

**Syntax**

```c
IppStatus ippsSMS4DecryptCBC(const Ipp8u* pSrc, Ipp8u* pDst, int len, const IppsSMS4Spec* pCtx, const Ipp8u* pIV);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

**Headers**: `ippcore.h`

**Libraries**: `ippcore.lib`

**Parameters**

- **pSrc**: Pointer to the input ciphertext data stream.
- **pDst**: Pointer to the resulting plaintext data stream of the variable length.
- **len**: Length of the ciphertext data stream length in bytes.
- **pCtx**: Pointer to the `IppsSMS4Spec` context.
- **pIV**: Pointer to the initialization vector for CBC mode operation.

**Description**

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**

- **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.
SMS4EncryptCFB

Encrypts byte data stream using SMS4 block cipher in the CFB mode.

Syntax

IppStatus ippsSMS4EncryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int len, int cfbBlkSize, const IppsSMS4Spec* pCtx, const Ipp8u *pIV);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc
Pointer to the input plaintext data stream of variable length.

pDst
Pointer to the resulting ciphertext data stream.

len
Length of the plaintext data stream in bytes.

cfbBlkSize
Size of the CFB block in bytes.

pCtx
Pointer to the IppsSMS4Spec context.

pIV
Pointer to the initialization vector for the CFB mode operation.

Description

The function encrypts the input data stream of variable length according to the CFB 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 input data stream length is less than or equal to zero.

ippStsUnderRunErr
Indicates an error condition if len is not divisible by cfbBlkSize parameter value.

ippStsCFBSizeErr
Indicates an error condition if the value for cfbBlkSize is illegal.

ippStsContextMatchErr
Indicates an error condition if the context parameter does not match the operation.
SMS4DecryptCFB

Decrypts byte data stream using SMS4 block cipher in CFB mode.

Syntax

IppStatus ippsSMS4DecryptCFB(const Ipp8u* pSrc, Ipp8u* pDst, int len, int cfbBlkSize,
const IppsSMS4Spec* pCtx, const Ipp8u* pIV);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc
Pointer to the input ciphertext data stream.
pDst
Pointer to the resulting plaintext data stream of variable length.
len
Length of the ciphertext data stream in bytes.
cfbBlkSize
Size of the CFB block in bytes.
pCtx
Pointer to the IppsSMS4Spec context.
pIV
Pointer to the initialization vector for the CFB mode operation.

Description

The function decrypts the input data stream of variable length according to the CFB 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.
ippStsCFBSizeErr
Indicates an error condition if the value for cfbBlkSize is illegal.
ippStsContextMatchErr
Indicates an error condition if the context parameter does not match the operation.
ippStsUnderRunErr
Indicates an error condition if len is not divisible by cipher block size.

SMS4EncryptOFB

Encrypts a variable length data stream using SMS4 block cipher in the OFB mode.
Syntax

IppStatus ippsSMS4EncryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int len, int ofbBlkSize, const IppsSMS4Spec* pCtx, Ipp8u* pIV);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc  
Pointer to the input plaintext data stream of variable length.

pDst  
Pointer to the resulting ciphertext data stream.

len  
Length of the plaintext data stream in bytes.

ofbBlkSize  
Size of the OFB block in bytes.

pCtx  
Pointer to the IppsSMS4Spec context.

pIV  
Pointer to the initialization vector for the OFB mode operation.

Description

The function encrypts the input data stream of a variable length in the OFB 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 input data stream length is less than or equal to zero.

ippStsUnderRunErr  
Indicates an error condition if len is not divisible by the ofbBlkSize parameter value.

ippStsOFBSizeErr  
Indicates an error condition if the value of ofbBlkSize is illegal.

ippStsContextMatchErr  
Indicates an error condition if the context parameter does not match the operation.

SMS4DecryptOFB

Decrypts a variable length data stream using SMS4 block cipher in the OFB mode.

Syntax

IppStatus ippsSMS4DecryptOFB (const Ipp8u* pSrc, Ipp8u* pDst, int len, int ofbBlkSize, const IppsSMS4Spec* pCtx, Ipp8u* pIV);

Include Files

ippcp.h
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
- **pSrc**: Pointer to the input ciphertext data stream of variable length.
- **pDst**: Pointer to the resulting plaintext data stream.
- **len**: Length of the ciphertext data stream in bytes.
- **ofbBlkSize**: Size of the OFB block in bytes.
- **pCtx**: Pointer to the IppsSMS4Spec context.
- **pIV**: Pointer to the initialization vector for the OFB mode operation.

Description
The function decrypts the input data stream of a variable length in the OFB 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 input data stream length is less than or equal to zero.
- **ippStsUnderRunErr**: Indicates an error condition if the value of **len** is not divisible by the **ofbBlkSize** parameter value.
- **ippStsOFBSizeErr**: Indicates an error condition if the value of **ofbBlkSize** is illegal.
- **ippStsContextMatchErr**: Indicates an error condition if the context parameter does not match the operation.

**SMS4EncryptCTR**
*Encrypts a variable length data stream using SMS4 block cipher in the CTR mode.*

Syntax
```c
IppStatus ippsSMS4EncryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int len, const IppsSMS4Spec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Include Files
- ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib
Parameters

- **pSrc**: Pointer to the input plaintext data stream of a variable length.
- **pDst**: Pointer to the resulting ciphertext data stream.
- **len**: Length of the plaintext data stream in bytes.
- **pCtx**: Pointer to the **IppsSMS4Spec** context.
- **pCtrValue**: Pointer to the counter data block.
- **ctrNumBitSize**: Number of bits in the specific part of the counter to be incremented.

Description

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

- **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 or equal to zero.
- **ippStsCTRSizeErr**: Indicates an error condition if the value of the **ctrNumBitSize** is illegal.
- **ippStsContextMatchErr**: Indicates an error condition if the context parameter does not match the operation.

**SMS4DecryptCTR**

Decrypts a variable length data stream using SMS4 block cipher in the CTR mode.

Syntax

```c
IppStatus ippsSMS4DecryptCTR(const Ipp8u* pSrc, Ipp8u* pDst, int len, const IppsAESSpec* pCtx, Ipp8u* pCtrValue, int ctrNumBitSize);
```

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pSrc**: Pointer to the input ciphertext data stream.
- **pDst**: Pointer to the resulting plaintext data stream of a variable length.
- **len**: Length of the plaintext data stream in bytes.
- **pCtx**: Pointer to the **IppsAESSpec** context.
Description
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

- 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.
- ippStsCTRSizeErr: Indicates an error condition if the value of the ctrNumBitSize is illegal.
- ippStsContextMatchErr: 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.

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. Clean up secret data stored in the context.
6. 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.

See Also
Data Security Considerations
**ARCFourGetSize**

*Gets the size of the IppsARCFourState context.*

**Syntax**

```c
IppStatus ippsARCFourGetSize(int* pSize);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- `pSize`: Pointer to the size value of the IppsARCFourState context.

**Description**

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**

```c
IppStatus ippsARCFourCheckKey(const Ipp8u* pKey, int keyLen, IppBool* pIsWeak);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**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**

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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- pKey: Pointer to the user-defined key.
- keyLen: Length of the user-defined key in octets.
- pCtx: Pointer to the IppsARCFourState context being initialized.

Description

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

- 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.

See Also

Data Security Considerations

ARCFourPack, ARCFourUnpack

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

Syntax

IppStatus ippsARCFourPack (const IppsARCFourState* pCtx, Ipp8u* pBuffer);
IppStatus ippsARCFourUnpack (const Ipp8u* pBuffer, IppsARCFourState* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pCtx
Pointer to the IppsARCFourState context.
pBuffer
Pointer to the user-defined buffer.

Description
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 IppsARCFourState context. The ARCFourPack and ARCFourUnpack functions enable replacing the position-dependent IppsARCFourState context in the memory.
Call the ARCFourGetSize function prior to ARCFourPack/ARCFourUnpack 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.

ARCFourEncrypt
Encrypts a variable length data stream according to ARCFour.

Syntax
IppStatus ippsARCFourEncrypt(const Ipp8u* pSrc, Ipp8u* pDst, int srclen, IppsARCFourState* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

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 octets.

PCTX

Pointer to the ARCFourState context.

Description

The function decrypts the input data stream of a variable length according to the ARCFour algorithm.

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 length of the input data stream is less than one octet.
- ippStsContextMatchErr: 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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- pSrc: Pointer to the input ciphertext data stream of variable length.
- pDst: Pointer to the resulting plaintext data stream.
- srclen: Length of the ciphertext data stream in octets.
- pCtx: Pointer to the ARCFourState context.

Description

The function decrypts the input data stream of a variable length according to the ARCFour algorithm.

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 length of the input data stream is less than one octet.

**ippStsContextMatchErr** Indicates an error condition if the context parameter does not match the operation.

---

**ARCFourReset**

*Resets the IppsARCFourState context to the initial state.*

**Syntax**

```c
IppStatus ippsARCFourReset(IppsARCFourState* pCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

*pCtx* Pointer to the IppsARCFourState context being reset.

**Description**

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**

**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.
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, SHA-512 [FIPS PUB 180-2], and SM3 [SM3]. These algorithms are widely used in enterprise applications nowadays.

Subsequent sections of this chapter describe 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.

Additionally Intel® Integrated Performance Primitives (Intel® IPP) supports two relatively new variants of SHA-512, the so called SHA-512/224 and SHA-512/256 algorithms. Both employ much of the basic SHA-512 algorithm but have some specifics. Intel IPP does not provide a separate API exactly targeting SHA-512/224 and SHA-512/256. To enable SHA-512/224 and SHA-512/256, Intel IPP declares extensions of the Hash Functions, Generalized Hash Functions, Mask Generation Functions, and Keyed Hash Functions. These extensions use the IppHashAlgId enumerator associated with a particular hash algorithm as shown in the table below.

<table>
<thead>
<tr>
<th>Supported Hash Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value of IppHashAlgId</strong></td>
</tr>
<tr>
<td>ippHashAlg_SHA1</td>
</tr>
<tr>
<td>ippHashAlg_SHA224</td>
</tr>
<tr>
<td>ippHashAlg_SHA256</td>
</tr>
<tr>
<td>ippHashAlg_SHA384</td>
</tr>
<tr>
<td>ippHashAlg_SHA512</td>
</tr>
<tr>
<td>ippHashAlg_SHA512_224</td>
</tr>
<tr>
<td>ippHashAlg_SHA512_256</td>
</tr>
<tr>
<td>ippHashAlg_MD5</td>
</tr>
<tr>
<td>ippHashAlg_SM3</td>
</tr>
</tbody>
</table>

Hash Functions

Functions described in this section apply hash algorithms to digesting streaming messages.

Usage model of the generalized hash functions is similar to the model explained below.
A primitive implementing a hash algorithm uses the state context `IppsHashState` as an operational vehicle to carry all necessary variables to manage the computation of the chaining 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 `HashGetSize` to get the size required to configure the `IppsHashState` context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the `HashInit` function with the value of `hashAlg` equal to `ippHashAlg_SHA1` to set up the initial context state with the SHA-1 specified initialization vectors.
3. Keep calling the function `HashUpdate` to digest incoming message stream in the queue till its completion. To determine the current value of the digest, call `HashGetTag` between the two calls to `HashUpdate`.
4. Call the function `HashFinal` to pad the partial block into a final SHA-1 message block and transform it into a 160-bit message digest value.
5. Clean up secret data stored in the context.
6. Call the operating system memory free service function to release the `IppsSHA1State` context.

The `IppsHashState` context is position-dependent. The `HashPack`, `HashUnpack` functions transform this context to a position-independent form and vice versa:

**See Also**

Data Security Considerations

**HashGetSize**

*Gets the size of the `IppsHashState` context in bytes.*

**Syntax**

```c
IppStatus ippsHashGetSize(int *pSize);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

Headers: `ippcore.h`
Libraries: `ippcore.lib`

**Parameters**

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

**Description**

The function gets the size of the `IppsHashState` 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 any of the specified pointers is `NULL`. 
HashInit

Initializes user-supplied memory as IppsHashState context for future use.

Syntax

IppStatus ippsHashInit(IppsHashState* pCtx, IppHashAlgId hashAlg);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pCtx Pointer to the IppsHashState context being initialized.
hashAlg Identifier of the hash algorithm.

Description

The function initializes the memory pointed by pCtx as IppsHashState context. The algID parameter defines the hash algorithm to be used in subsequent calls to HashUpdate, HashFinal, or HashGetTag functions. The hashAlg parameter can take one of the values listed in table Supported Hash Algorithms.

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.
ippStsNotSupportedModeErr Indicates an error condition if the hashAlg parameter does not match any value of IppHashAlg listed in table Supported Hash Algorithms.

See Also

Data Security Considerations

HashPack, HashUnpack

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

Syntax

IppStatus ippsHashPack (const IppsHashState* pCtx, Ipp8u* pBuffer, int bufSize);
IppStatus ippsHashUnpack (const Ipp8u* pBuffer, IppsHashState* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\texttt{pCtx}  
Pointer to the \texttt{IppsHashState} context.

\texttt{pBuffer}  
Pointer to the user-defined buffer.

\texttt{bufSize}  
The size of the user-defined buffer in bytes.

Description

The \texttt{HashPack} function transforms the \texttt{*pCtx} context to a position-independent form and stores it in the the \texttt{*pBuffer} buffer. The \texttt{HashUnpack} function performs the inverse operation, that is, transforms the contents of the \texttt{*pBuffer} buffer into a normal \texttt{IppsHashState} context. The \texttt{HashPack} and \texttt{HashUnpack} functions enable replacing the position-dependent \texttt{IppsHashState} context in the memory.

The value of the \texttt{bufSize} parameter must be not less than the size of \texttt{IppsHashState} context. Call the \texttt{HashGetSize} function prior to \texttt{HashPack} to determine the size of the buffer.

Return Values

\begin{itemize}
  \item \texttt{ippStsNoErr}  
  Indicates no error. Any other value indicates an error or warning.
  \item \texttt{ippStsNullPtrErr}  
  Indicates an error condition if any of the specified pointers is \texttt{NULL}.
  \item \texttt{ippStsMemErr}  
  Indicates an error condition if the value of \texttt{bufSize} is less than the size of the \texttt{IppsHashState} context.
\end{itemize}

\textbf{HashDuplicate}

\textit{Copies one \texttt{IppsHashState} context to another.}

Syntax

\begin{verbatim}
IppStatus ippsHashDuplicate(const IppsHashState* pSrcCtx, IppsHashState* pDstCtx);
\end{verbatim}

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\texttt{pSrcCtx}  
Pointer to the input \texttt{IppsHashState} context to be cloned.

\texttt{pDstCtx}  
Pointer to the output \texttt{IppsHashState} context.

Description

The function copies one \texttt{IppsHashState} context to another.

Return Values

\begin{itemize}
  \item \texttt{ippStsNoErr}  
  Indicates no error. Any other value indicates an error or warning.
\end{itemize}
HashUpdate

Digests the current input message stream of the specified length.

Syntax

IppStatus ippsHashUpdate(const Ipp8u *pSrc, int len, IppsHashState *pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc  
Pointer to the buffer containing a part of or the whole message.

len  
Length of the actual part of the message in bytes.

pCtx  
Pointer to the IppsHashState context.

Description

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

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.

ippStsLengthErr  
Indicates an error condition in any of the following cases:

- The length of the input data stream is less than zero
- The length of the totally processed stream (including the current update request) exceeds the limit defined by the particular hash algorithm.
HashFinal

Completes computation of the digest value.

Syntax

IppStatus ippsHashFinal(Ipp8u *pMD, IppsHashState *pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pMD Pointer to the resultant digest value.
pCtx Pointer to the IppsHashState context.

Description

The function completes calculation of the digest value and stores the result at the specified memory location.

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.

HashGetTag

Computes the current digest value of the processed part of the message.

Syntax

IppStatus ippsHashGetTag(Ipp8u* pTag, int tagLen, const IppsHashState* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pTag Pointer to the authentication tag.
The length of the tag (in bytes).

Pointer to the IppsHashState context.

**Description**

The function computes the message digest based on the current context as specified in [FIPS PUB 180-2], [FIPS PUB 180-4] and [RFC 1321]. A call to this function retains the possibility to update the digest.

**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 tagLen < 1 or tagLen exceeds the maximal length of a particular digest.
- **ippStsContextMatchErr**: Indicates an error condition if the context parameter does not match the operation.

**SM3GetSize**

*Gets the size of the IppsSM3State context in bytes.*

**Syntax**

```c
IppStatus ippsSM3GetSize(int *pSize);
```

**Include Files**

ippcp.h

**Domain Dependencies**

*Headers*: ippcore.h

*Libraries*: ippcore.lib

**Parameters**

- **pSize**: Pointer to the IppsSM3State context size value.

**Description**

The function gets the IppsSM3State 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.

**SM3Init**

*Initializes user-supplied memory as IppsSM3State context for future use.*
Syntax
IppStatus ippsSM3Init(IppsSM3State* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pCtx Pointer to the IppsSM3State context being initialized.

Description
The function initializes the memory pointed by pCtx as IppsSM3State 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.

See Also
Data Security Considerations

SM3Pack, SM3Unpack
Packs/unpacks the IppsSM3State context into/from a user-defined buffer.

Syntax
IppStatus ippsSM3Pack (const IppsSM3State* pCtx, Ipp8u* pBuffer);
IppStatus ippsSM3Unpack (const Ipp8u* pBuffer, IppsSM3State* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pCtx Pointer to the IppsSM3State context.
pBuffer Pointer to the user-defined buffer.
Description
The SM3Pack function transforms the *pCtx context to a position-independent form and stores it in the *pBuffer buffer. The SM3Unpack function performs the inverse operation, that is, transforms the contents of the *pBuffer buffer into a normal IppsSM3State context. The SM3Pack and SM3Unpack functions enable replacing the position-dependent IppsSM3State context in the memory.

Call the SM3GetSize function prior to SM3Pack/SM3Unpack 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.

SM3Duplicate
Copies one IppsSM3State context to another.

Syntax
IppStatus ippsSM3Duplicate(const IppsSM3State* pSrcCtx, IppsSM3State* pDstCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pSrcCtx Pointer to the source IppsSM3State context to be cloned.
pDstCtx Pointer to the destination IppsSM3State context.

Description
The function copies one IppsSM3State context to another.

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.

SM3Update
Digests the current input message stream of the specified length.
Syntax
IppStatus ippsSM3Update(const Ipp8u *pSrc, int len, IppsSM3State *pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pSrc
     Pointer to the buffer containing a part of or the whole message.

len
     Length of the actual part of the message in bytes.

pCtx
     Pointer to the IppsSM3State context.

Description
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
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.

ippStsLengthErr
     Indicates an error condition if the input data stream length is less than zero.

SM3Final
 Completes computation of the SM3 digest value.

Syntax
IppStatus ippsSM3Final(Ipp8u *pMD, IppsSM3State *pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib
Parameters

\( pMD \quad \) Pointer to the resultant digest value.

\( pCtx \quad \) Pointer to the IppsSM3State context.

Description

The function completes calculation of the digest value and stores the result into the specified memory.

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.

SM3GetTag

*Computes the current SM3 digest value of the processed part of the message.*

Syntax

\[
\text{IppStatus ippsSM3GetTag(Ipp8u* } pTag, \text{ Ipp32u tagLen, const IppsSM3State* } pCtx);\]

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

\( pTag \quad \) Pointer to the authentication tag.

\( tagLen \quad \) Length of the tag (in bytes).

\( pCtx \quad \) Pointer to the IppsSM3State context.

Description

The function computes the message digest based on the current context as specified in [SM3]. A call to this function retains the possibility to update the digest.

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 \( \text{tagLen} < 1 \) or \( \text{tagLen} \) exceeds the maximal length of a particular digest.
MD5GetSize

*Gets the size of the IppsMD5State context in bytes.*

**Syntax**

```c
IppStatus ippsMD5GetSize(int *pSize);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pSize**
  
  Pointer to the IppsMD5State context size value.

**Description**

The function gets the IppsMD5State 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.

MD5Init

*Initializes user-supplied memory as IppsMD5State context for future use.*

**Syntax**

```c
IppStatus ippsMD5Init(IppsMD5State* pCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pCtx**
  
  Pointer to the IppsMD5State context being initialized.

**Description**

The function initializes the memory pointed by pCtx as IppsMD5State 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.

See Also
Data Security Considerations

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);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pCtx Pointer to the IppsMD5State context.
pBuffer Pointer to the user-defined buffer.

Description
The MD5Pack function transforms the *pCtx context to a position-independent form and stores it in 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

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.

MD5Duplicate
Copies one IppsMD5State context to another.

Syntax
IppStatus ippsMD5Duplicate(const IppsMD5State* pSrcCtx, IppsMD5State* pDstCtx);
Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrcCtx  
Pointer to the source IppsMD5State context to be cloned.

pDstCtx  
Pointer to the destination IppsMD5State context.

Description

The function copies one IppsMD5State context to another.

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.

MD5Update

Digests the current input message stream of the specified length.

Syntax

IppStatus ippsMD5Update(const Ipp8u *pSrcMesg, int mesglen, IppsMD5State *pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrcMesg  
Pointer to the buffer containing a part of or the whole message.

mesglen  
Length of the actual part of the message in bytes.

pCtx  
Pointer to the IppsMD5State context.

Description

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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippStsNoErr</td>
<td>Indicates no error. Any other value indicates an error or warning.</td>
</tr>
<tr>
<td>ippStsNullPtrErr</td>
<td>Indicates an error condition if any of the specified pointers is NULL.</td>
</tr>
<tr>
<td>ippStsContextMatchErr</td>
<td>Indicates an error condition if the context parameter does not match the operation.</td>
</tr>
<tr>
<td>ippStsLengthErr</td>
<td>Indicates an error condition if the input data stream length is less than zero.</td>
</tr>
</tbody>
</table>

**MD5Final**

*Completes computation of the MD5 digest value.*

**Syntax**

```c
IppStatus ippsMD5Final(Ipp8u *pMD, IppsMD5State *pCtx);
```

**Include Files**

IPP_cp.h

**Domain Dependencies**

*Headers:* IPP_core.h

*Libraries:* IPP_core.lib

**Parameters**

- **pMD**
  
  Pointer to the resultant digest value.

- **pCtx**
  
  Pointer to the IppsMD5State context.

**Description**

The function completes calculation of the digest value and stores the result into the specified memory.

**Return Values**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippStsNoErr</td>
<td>Indicates no error. Any other value indicates an error or warning.</td>
</tr>
<tr>
<td>ippStsNullPtrErr</td>
<td>Indicates an error condition if any of the specified pointers is NULL.</td>
</tr>
<tr>
<td>ippStsContextMatchErr</td>
<td>Indicates an error condition if the context parameter does not match the operation.</td>
</tr>
</tbody>
</table>

**MD5GetTag**

*Computes the current MD5 digest value of the processed part of the message.*
Syntax
IppStatus ippsMD5GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsMD5State* pState);

Include Files
ippcp.h

Domain Dependencies
headers: ippcore.h
Libraries: ippcore.lib

Parameters
pDstTag  Pointer to the authentication tag.
tagLen  Length of the tag (in bytes).
pState  Pointer to the IppsMD5State context.

Description
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
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 tagLen < 1 or tagLen exceeds the maximal length of a particular digest.
ippStsContextMatchErr 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 ippsSHA1GetSize(int *pSize);

Include Files
ippcp.h

Domain Dependencies
headers: ippcore.h
Libraries: ippcore.lib

Parameters
pSize  Pointer to the IppsSHA1State context size value.
Description
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 ippsSHA1Init(IppsSHA1State* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pCtx Pointer to the IppsSHA1State context being initialized.

Description
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.

See Also
Data Security Considerations

SHA1Pack, SHA1Unpack
Packs/unpacks the IppsSHA1State context into/from a user-defined buffer.

Syntax
IppStatus ippsSHA1Pack (const IppsSHA1State* pCtx, Ipp8u* pBuffer);
IppStatus ippsSHA1Unpack (const Ipp8u* pBuffer, IppsSHA1State* pCtx);

Include Files
ippcp.h
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- pCtx: Pointer to the IppsSHA1State context.
- pBuffer: Pointer to the user-defined buffer.

Description
The SHA1Pack function transforms the *pCtx context to a position-independent form and stores it in 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

- 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.

SHA1Duplicate
Copies one IppsSHA1State context to another.

Syntax
IppStatus ippsSHA1Duplicate(const IppsSHA1State* pSrcCtx, IppsSHA1State* pDstCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- pSrcCtx: Pointer to the source IppsSHA1State context to be cloned.
- pDstCtx: Pointer to the destination IppsSHA1State context.

Description
The function copies one IppsSHA1State context to another.

Return Values

- ippStsNoErr: Indicates no error. Any other value indicates an error or warning.
SHA1Update

**Digests the current input message stream of the specified length.**

**Syntax**

```c
IppStatus ippsSHA1Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA1State *pCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pSrcMesg**: Pointer to the buffer containing a part of or the whole message.
- **mesglen**: Length of the actual part of the message in bytes.
- **pCtx**: Pointer to the IppsSHA1State context.

**Description**

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**

- **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.
- **ippStsLengthErr**: Indicates an error condition if the input data stream length is less than zero.

SHA1Final

**Completes computation of the SHA-1 digest value.**

**Syntax**

```c
IppStatus ippsSHA1Final(Ipp8u *pMD, IppsSHA1State *pCtx);
```
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\texttt{pMD} \hspace{1cm} \text{Pointer to the resultant digest value.}

\texttt{pCtx} \hspace{1cm} \text{Pointer to the IppsSHA1State context.}

Description
The function completes calculation of the digest value and stores the result into the specified memory.

Return Values

\texttt{ippStsNoErr} \hspace{1cm} \text{Indicates no error. Any other value indicates an error or warning.}

\texttt{ippStsNullPtrErr} \hspace{1cm} \text{Indicates an error condition if any of the specified pointers is NULL.}

\texttt{ippStsContextMatchErr} \hspace{1cm} \text{Indicates an error condition if the context parameter does not match the operation.}

\textbf{SHA1GetTag}

\underline{Computes the current SHA-1 digest value of the processed part of the message.}

Syntax
IppStatus ippsSHA1GetTag(Ipp8u* \texttt{pDstTag}, Ipp32u \texttt{tagLen}, const IppsSHA1State* \texttt{pState});

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\texttt{pDstTag} \hspace{1cm} \text{Pointer to the authentication tag.}

\texttt{tagLen} \hspace{1cm} \text{Length of the tag (in bytes).}

\texttt{pState} \hspace{1cm} \text{Pointer to the IppsSHA1State context.}

Description
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

- 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 \( tagLen < 1 \) or \( tagLen \) exceeds the maximal length of a particular digest.
- ippStsContextMatchErr: Indicates an error condition if the context parameter does not match the operation.

SHA224GetSize

*Gets the size of the IppsSHA224State context in bytes.*

**Syntax**

```c
IppStatus ippsSHA224GetSize(int *pSize);
```

**Include Files**

ippcp.h

**Domain Dependencies**

**Headers:** ippcore.h  
**Libraries:** ippcore.lib

**Parameters**

- pSize: Pointer to the IppsSHA224State context size value.

**Description**

The function gets the IppsSHA224State 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.

SHA224Init

*Initializes user-supplied memory as IppsSHA224State context for future use.*

**Syntax**

```c
IppStatus ippsSHA224Init(IppsSHA224State* pCtx);
```

**Include Files**

ippcp.h
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pCtx               Pointer to the IppsSHA224State context being initialized.

Description
The function initializes the memory pointed by pCtx as IppsSHA224State 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.

See Also
Data Security Considerations

SHA224Pack, SHA224Unpack
Packs/unpacks the IppsSHA224State context into/from a user-defined buffer.

Syntax
IppStatus ippsSHA224Pack (const IppsSHA224State* pCtx, Ipp8u* pBuffer);
IppStatus ippsSHA224Unpack (const Ipp8u* pBuffer, IppsSHA224State* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pCtx               Pointer to the IppsSHA224State context.
pBuffer             Pointer to the user-defined buffer.

Description
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

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.

SHA224Duplicate

Copies one IppsSHA224State context to another.

Syntax

IppStatus ippsSHA224Duplicate(const IppsSHA224State* pSrcCtx, IppsSHA224State* pDstCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrcCtx  
Pointer to the source SHA224State context to be cloned.
pDstCtx  
Pointer to the destination IppsSHA224State context.

Description

The function copies one IppsSHA224State context to another.

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.

SHA224Update

Digests the current input message stream of the specified length.

Syntax

IppStatus ippsSHA224Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA224State *pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

*pSrcMesg  
Pointer to the buffer containing a part of or the whole message.

mesglen  
Length of the actual part of the message in bytes.

*pCtx  
Pointer to the IppsSHA224State context.

Description

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

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.

ippStsLengthErr  
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 ippsSHA224Final(Ipp8u *pMD, IppsSHA224State *pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

*pMD  
Pointer to the resultant digest value.

*pCtx  
Pointer to the IppsSHA224State context.

Description

The function completes calculation of the digest value and stores the result into the specified memory.
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.

**SHA224GetTag**

*Computes the current SHA-224 digest value of the processed part of the message.*

**Syntax**

```c
IppStatus ippsSHA224GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA224State* pState);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

- **Headers**: `ippcore.h`
- **Libraries**: `ippcore.lib`

**Parameters**

- **pDstTag**: Pointer to the authentication tag.
- **tagLen**: Length of the tag (in bytes).
- **pState**: Pointer to the `IppsSHA224State` context.

**Description**

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**

- **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 `tagLen < 1` or `tagLen` exceeds the maximal length of a particular digest.
- **ippStsContextMatchErr**: 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 ippsSHA256GetSize(int *pSize);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSize  Pointer to the IppsSHA256State context size value.

Description
The function gets the IppsSHA256State 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.

SHA256Init
Initializes user-supplied memory as IppsSHA256State context for future use.

Syntax
IppStatus ippsSHA256Init(IppsSHA256State *pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pCtx  Pointer to the IppsSHA256State context being initiaized.

Description
The function initializes the memory pointed by pCtx as IppsSHA256State context.

Return Values
ippStsNoErr  Indicates no error. Any other value indicates an error or warning.
Indicates an error condition if any of the specified pointers is NULL.

See Also
Data Security Considerations

SHA256Pack, SHA256Unpack
*Packs/unpacks the IppsSHA256State context into/from a user-defined buffer.*

Syntax
IppStatus ippsSHA256Pack (const IppsSHA256State* pCtx, Ipp8u* pBuffer);
IppStatus ippsSHA256Unpack (const Ipp8u* pBuffer, IppsSHA256State* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

* pCtx    Pointer to the IppsSHA256State context.
* pBuffer  Pointer to the user-defined buffer.

Description
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

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.

SHA256Duplicate
Copies one IppsSHA256State context to another.

Syntax
IppStatus ippsSHA256Duplicate(const IppsSHA256State* pSrcCtx, IppsSHA256State* pDstCtx);

Include Files
ippcp.h
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrcCtx  
Pointer to the source IppsSHA256State context to be cloned.

pDstCtx  
Pointer to the destination IppsSHA256State context.

Description
The function copies one IppsSHA256State context to another.

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.

SHA256Update
Digests the current input message stream of the specified length.

Syntax
IppStatus ippsSHA256Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA256State *pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrcMesg  
Pointer to the buffer containing a part of or the whole message.

mesglen  
Length of the actual part of the message in bytes.

pCtx  
Pointer to the IppsSHA256State context.

Description
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

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.

ippStsLengthErr  
Indicates an error condition if the input data stream length is less than zero.

SHA256Final
Complete computation of the SHA-256 digest value.

Syntax
IppStatus ippsSHA256Final(Ipp8u *pMD, IppsSHA256State *pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pMD  
Pointer to the resultant digest value.

pCtx  
Pointer to the IppsSHA256State context.

Description
The function completes calculation of the digest value and stores the result into the specified memory.

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.

SHA256GetTag
Computes the current SHA-256 digest value of the processed part of the message.

Syntax
IppStatus ippsSHA256GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA256State* pState);
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pDstTag  
Pointer to the authentication tag.

tagLen  
Length of the tag (in bytes).

pState  
Pointer to the IppsSHA265State context.

Description
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

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 tagLen < 1 or tagLen exceeds the maximal length of a particular digest.

ippStsContextMatchErr  
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 ippsSHA384GetSize(int *pSize);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSize  
Pointer to the IppsSHA384State context size value.

Description
The function gets the IppsSHA384State 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.

SHA384Init

Initializes user-supplied memory as IppsSHA384State context for future use.

Syntax

IppStatus ippsSHA384Init(IppsSHA384State* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pCtx Pointer to the IppsSHA384State context being initialized.

Description

The function initializes the memory pointed by pCtx as IppsSHA384State 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.

See Also

Data Security Considerations

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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Parameters

- **pCtx**: Pointer to the `IppsSHA384State` context.
- **pBuffer**: Pointer to the user-defined buffer.

Description

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

- `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.

SHA384Duplicate

*Copies one `IppsSHA384State` context to another.*

Syntax

```c
IppStatus ippsSHA384Duplicate(const IppsSHA384State* pSrcCtx, IppsSHA384State* pDstCtx);
```

Include Files

- ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pSrcCtx**: Pointer to the source `IppsSHA384State` context to be cloned.
- **pDstCtx**: Pointer to the destination `IppsSHA384State` context.

Description

The function copies one `IppsSHA384State` context to another.

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.
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**

```c
IppStatus ippsSHA384Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA384State *pCtx);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

*Headers:* `ippcore.h`

*Libraries:* `ippcore.lib`

**Parameters**

- `pSrcMesg`: Pointer to the buffer containing a part of or the whole message.
- `mesglen`: Length of the actual part of the message in bytes.
- `pCtx`: Pointer to the `IppsSHA384State` context.

**Description**

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**

- `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.
- `ippStsLengthErr`: Indicates an error condition if the input data stream length is less than zero.

**SHA384Final**

*Completes computing of the SHA-384 digest value.*

**Syntax**

```c
IppStatus ippsSHA384Final( Ipp8u *pMD, IppsSHA384State *pCtx);
```

**Include Files**

`ippcp.h`
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\[ \text{pMD} \quad \text{Pointer to the resultant digest value.} \]
\[ \text{pCtx} \quad \text{Pointer to the IppsSHA384State context.} \]

Description
The function completes calculation of the digest value and stores the result into the specified memory.

Return Values

\[ \text{ippStsNoErr} \quad \text{Indicates no error. Any other value indicates an error or warning.} \]
\[ \text{ippStsNullPtrErr} \quad \text{Indicates an error condition if any of the specified pointers is NULL.} \]
\[ \text{ippStsContextMatchErr} \quad \text{Indicates an error condition if the context parameter does not match the operation.} \]

SHA384GetTag

*Computes the current SHA-384 digest value of the processed part of the message.*

Syntax

\[
\text{IppStatus ippsSHA384GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA384State* pState);} \]

Include Files

ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\[ \text{pDstTag} \quad \text{Pointer to the authentication tag.} \]
\[ \text{tagLen} \quad \text{Length of the tag (in bytes).} \]
\[ \text{pState} \quad \text{Pointer to the IppsSHA384State context.} \]

Description
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

\[ \text{ippStsNoErr} \quad \text{Indicates no error. Any other value indicates an error or warning.} \]
### SHA512GetSize

**Gets the size of the IppsSHA512State context in bytes.**

**Syntax**

```c
IppStatus ippsSHA512GetSize(int *pSize);
```

**Include Files**

ippcp.h

**Domain Dependencies**

- **Headers:** ippcore.h
- **Libraries:** ippcore.lib

**Parameters**

- **pSize**
  
  Pointer to the IppsSHA512State context size value.

**Description**

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**

```c
IppStatus ippsSHA512Init(IppsSHA512State* pCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

- **Headers:** ippcore.h
- **Libraries:** ippcore.lib
Parameters

\( pCtx \)

**Pointer to the IppsSHA512State context being initialized.**

Description

The function initializes the memory pointed by \( pCtx \) as IppsSHA512State context.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippStsNoErr</td>
<td>Indicates no error. Any other value indicates an error or warning.</td>
</tr>
<tr>
<td>ippStsNullPtrErr</td>
<td>Indicates an error condition if any of the specified pointers is NULL.</td>
</tr>
</tbody>
</table>

See Also

Data Security Considerations

SHA512Pack, SHA512Unpack

*Packs/unpacks the IppsSHA512State context into/from a user-defined buffer.*

Syntax

```
IppStatus ippsSHA512Pack (const IppsSHA512State* pCtx, Ipp8u* pBuffer);
IppStatus ippsSHA512Unpack (const Ipp8u* pBuffer, IppsSHA512State* pCtx);
```

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pCtx )</td>
<td><strong>Pointer to the IppsSHA512State context.</strong></td>
</tr>
<tr>
<td>( pBuffer )</td>
<td><strong>Pointer to the user-defined buffer.</strong></td>
</tr>
</tbody>
</table>

Description

The SHA512Pack function transforms the \( *pCtx \) context to a position-independent form and stores it in 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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippStsNoErr</td>
<td>Indicates no error. Any other value indicates an error or warning.</td>
</tr>
<tr>
<td>ippStsNullPtrErr</td>
<td>Indicates an error condition if any of the specified pointers is NULL.</td>
</tr>
</tbody>
</table>
**SHA512Duplicate**

*Copies one IppsSHA512State context to another.*

**Syntax**

```c
IppStatus ippsSHA512Duplicate(const IppsSHA512State* pSrcCtx, IppsSHA512State* pDstCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

- **Headers:** ippcore.h
- **Libraries:** ippcore.lib

**Parameters**

- `pSrcCtx`  
  Pointer to the source IppsSHA512State context to be cloned.
- `pDstCtx`  
  Pointer to the destination IppsSHA512State context.

**Description**

The function copies one IppsSHA512State context to another.

**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.

**SHA512Update**

*Digests the current input message stream of the specified length.*

**Syntax**

```c
IppStatus ippsSHA512Update(const Ipp8u *pSrcMesg, int mesglen, IppsSHA512State *pCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

- **Headers:** ippcore.h
- **Libraries:** ippcore.lib

**Parameters**

- `pSrcMesg`  
  Pointer to the buffer containing a part of or the whole message.
**Description**
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**
- **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.
- **ippStsLengthErr**: 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 ippsSHA512Final(Ipp8u *pMD, IppsSHA512State *pCtx);

**Include Files**
ippcp.h

**Domain Dependencies**
Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**
- **pMD**: Pointer to the resultant digest value.
- **pCtx**: Pointer to the IppsSHA512State context.

**Description**
The function completes calculation of the digest value and stores the result into the specified memory.

**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.
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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pDstTag**: Pointer to the authentication tag.
- **tagLen**: Length of the tag (in bytes).
- **pState**: Pointer to the IppsSHA512State context.

Description

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

- **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 tagLen < 1 or tagLen exceeds the maximal length of a particular digest.
- **ippStsContextMatchErr**: Indicates an error condition if the context parameter does not match the operation.

Hash Functions for Non-Streaming Messages

This section describes functions that calculate a digest of an entire (non-streaming) input message by applying a selected hash algorithm, as well as a possibility to use a different implementation of a hash algorithm.
General Definition of a Hash Function

Syntax

typedef IppStatus(_STDCALL *IppHASH)(const Ipp8u* pMsg, int msgLen, Ipp8u* pMD);

Parameters

pMsg Pointer to the input octet string.
msgLen Length of the input string in octets.
pMD 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 $2^{32}-1$.

HashMessage

Computes the digest value of an input message.

Syntax

IppStatus ippsHashMessage(const Ipp8u *pMsg, int len, Ipp8u *pMD, IppHashAlgId hashAlg);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pMsg Pointer to the input message.
len Message length in octets.
pMD Pointer to the resultant digest.
hashAlg Identifier of the hash algorithm.

Description

The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message. The hashAlg parameter defines the hash algorithm used, as explained in table Supported Hash Algorithms.
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 length of the input data stream is less than zero.

ippStsNotSupportedModeErr  Indicates an error condition if the hashAlg parameter does not match any value of IppHashAlg listed in table Supported Hash Algorithms.

Example

The code below computes MD5 digest of a message.

```c
void MD5_sample(void)
{
    // define message
    Ipp8u msg[] = "abcdefghijklmnopqrstuvwxyz";

    // once the whole message is placed into memory,
    // you can use the integrated primitive
    Ipp8u digest[16];
    ippsHashMessage(msg, strlen((char*)msg), digest, IPP_ALG_HASH_MD5);
}
```

SM3MessageDigest

*Computes SM3 digest value of the input message.*

Syntax

```c
IppStatus ippsSM3MessageDigest(const Ipp8u *pMsg, int len, Ipp8u *pMD);
```

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h

Libraries: ippcore.lib

Parameters

- `pMsg`  Pointer to the input message.
- `len`  Message length in octets.
- `pMD`  Pointer to the resultant digest.

Description

The function uses the selected hash algorithm to compute digest value of the entire (non-streaming) input message.
MD5MessageDigest

*Computes MD5 digest value of the input message.*

**Syntax**

```c
IppStatus ippsMD5MessageDigest(const Ipp8u *pSrcMesg, int mesgLen, Ipp8u *pMD);
```

**Include Files**

ippcp.h

**Domain Dependencies**

**Headers:** ippcore.h  
**Libraries:** ippcore.lib

**Parameters**

- `pSrcMesg`  
  Pointer to the input message.

- `mesgLen`  
  Message length in octets.

- `pMD`  
  Pointer to the resultant digest.

**Description**

The function uses the selected hash algorithm to compute digest value of the entire (non-streaming) input message.

**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.
Example
The code example below shows MD5 digest of a message.

```c
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**

```c
IppStatus ippsSHA1MessageDigest(const Ipp8u *pSrcMesg, int mesgLen, Ipp8u *pMD);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pSrcMesg**
  Pointer to the input message.
- **mesgLen**
  Message length in octets.
- **pMD**
  Pointer to the resultant digest.

**Description**

The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

**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.
Example

The code example below shows SHA1 digest of a message.

```c
// 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[] = "abcdbcdecdefgefghfghighijhijklklmklmnlmnomnopnopq";

    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 *pSrcMesg, int mesgLen, Ipp8u *pMD);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrcMesg  Pointer to the input message.
mesgLen   Message length in octets.
pMD       Pointer to the resultant digest.

Description

The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming)
input message.

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.

SHA256MessageDigest

Computes SHA-256 digest value of the input message.

Syntax

IppStatus ippsSHA256MessageDigest(const Ipp8u *pSrcMesg, int mesgLen, Ipp8u *pMD);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrcMesg  Pointer to the input message.
**Description**
The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

**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.

**SHA384MessageDigest**
*Computes SHA-384 digest value of the input message.*

**Syntax**
```c
IppStatus ippsSHA384MessageDigest(const Ipp8u *pSrcMesg, int mesgLen, Ipp8u *pMD);
```

**Include Files**
ippcp.h

**Domain Dependencies**
*Headers: ippcore.h*  
*Libraries: ippcore.lib*

**Parameters**
- **pSrcMesg**: Pointer to the input message.  
- **mesgLen**: Message length in octets.  
- **pMD**: Pointer to the resultant digest.

**Description**
The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message.

**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.
SHA512MessageDigest
Computes SHA-512 digest value of the input message.

Syntax
IppStatus ippsSHA512MessageDigest(const Ipp8u *pSrcMesg, int mesgLen, Ipp8u *pMD);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pSrcMesg         Pointer to the input message.
mesgLen          Message length in octets.
pMD              Pointer to the resultant digest.

Description
The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming)
input message.

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.

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 hash algorithms, as well as a possibility to use a different
implementation of MGF.

- Intel IPP implementation of MGFs limits the 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**

- **pSeed**: Pointer to the input octet string.
- **seedLen**: Length of the input string.
- **pMask**: Pointer to the output pseudorandom mask.
- **maskLen**: 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**

*Generates a pseudorandom mask of the specified length using a selected hash algorithm.*

**Syntax**

IppStatus ippsMGF(const Ipp8u *pSeed, int seedLen, Ipp8u* pMask, int maskLen, IppHashAlgId hashAlg);

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- **pSeed**: Pointer to the input octet string.
- **seedLen**: Length of the input string.
- **pMask**: Pointer to the output pseudorandom mask.
- **maskLen**: Desired length of the output.
- **hashAlg**: Identifier of the hash algorithm.
Description

The function generates a pseudorandom mask of the specified length using the hash algorithm defined by \textit{algID}. The \textit{hashAlg} parameter can take one of the values listed in table Supported Hash Algorithms.

Return Values

- \texttt{ippStsNoErr} Indicates no error. Any other value indicates an error or warning.
- \texttt{ippStsNullPtrErr} Indicates an error condition if \texttt{pMask} pointer is NULL.
- \texttt{ippStsLengthErr} Indicates an error condition if any of the specified lengths is negative or zero.
- \texttt{ippStsNotSupportedModeErr} Indicates an error condition if the \textit{hashAlg} parameter does not match any value of \texttt{IppHashAlg} listed in table Supported Hash Algorithms.
Data Authentication Primitive Functions

This chapter describes the Intel® IPP functions for generating message authentication code (MAC), that is, Message Authentication Functions.

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.

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.

Usage model of the generalized HMAC functions is similar to the model explained below.

Each HMAC scheme is implemented as a set of the primitive functions. Each primitive implementing HMAC uses the HashState context as an operational vehicle to carry all necessary variables to manage computation of the chaining 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 HMACGetSize to get the size required to configure the HashState context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the function HMACInit with the value of hashAlg equal to ippHashAlg_SHA1 to set up key material and the initial context state with the SHA-1 specified initialization vectors.
3. Keep calling the function HMACUpdate to digest incoming message stream in the queue till its completion. To determine the current value of the message digest, call HMACGetTag between the two calls to HMACUpdate.
4. Call the function HMACSHA1Final to pad the partial block into a final SHA-1 message block and transform it into a resulting HMAC value.
5. Clean up secret data stored in the context.
6. Call the operating system memory free service function to release the HashState context.

The HashState context is position-dependent. The HMACPack, HMACUnpack functions transform it to a position-independent form and vice versa:

See Also
Data Security Considerations

HMACGetSize

Gets the size of the IppsHMACState context.

Syntax

IppStatus ippsHMACGetSize(int *pSize);
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSize

Pointer to the value of the IppsHMACState context size.

Description
The function gets the size of the IppsHMACState 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 any of the specified pointers is NULL.

HMAC_Init

Initializes user-supplied memory as IppsHMACState context for future use.

Syntax

IppStatus ippsHMAC_Init(const Ipp8u *pKey, int keyLen, IppsHMACState *pCtx, IppHashAlgId hashAlg);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pKey

Pointer to the user-supplied key.

keyLen

Key length in bytes.

pCtx

Pointer to the IppsHMACState context being initialized.

hashAlg

Identifier of the hash algorithm.

Description
The function initializes the memory pointed by pCtx as the IppsHMACState context. The function also sets up the initial chaining digest value according to the hash algorithm specified by the hashAlg parameter and computes necessary key material from the supplied key pKey. The hashAlg parameter can take one of the values listed in table Supported Hash Algorithms.
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.

ippStsNotSupportedModeErr  Indicates an error condition if the hashAlg parameter does not match any value of IppHashAlg listed in table Supported Hash Algorithms.

See Also

Data Security Considerations

HMAC_Pack, HMAC_Unpack

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

Syntax

IppStatus ippsHMAC_Pack (const IppsHMACState* pCtx, Ipp8u* pBuffer, int bufSize);
IppStatus ippsHMAC_Unpack (const Ipp8u* pBuffer, IppsHMACState* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pCtx  Pointer to the IppsHMACState context.

pBuffer  Pointer to the user-defined buffer.

bufSize  The size of the user-defined buffer in bytes.

Description

The HMAC_Pack function transforms the *pCtx context to a position-independent form and stores it in the *pBuffer buffer. The HMAC_Unpack function performs the inverse operation, that is, transforms the contents of the *pBuffer buffer into a normal IppsHMACState context. The HMAC_Pack and HMAC_Unpack functions enable replacing the position-dependent IppsHMACState context in the memory. Call the HMAC_GetSize function prior to HMAC_Pack 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.

ippStsMemErr  Indicates an error condition if the value of bufSize is less than the size of the IppsHMACState context.
HMAC_Duplicate

Copies one IppsHMACState context to another.

Syntax

IppStatus ippsHMAC_Duplicate(const IppsHMACState* pSrcCtx, IppsHMACState* pDstCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrcCtx  
Pointer to the input IppsHMACState context to be cloned.

pDstCtx  
Pointer to the output IppsHMACState context.

Description

The function copies one IppsHMACState context to another.

Return Values

ippStsNoErr  Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr  Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr  Indicates an error condition if any of the context parameters does not match the operation.

HMAC_Update

Digests the current input message stream of the specified length.

Syntax

IppStatus ippsHMAC_Update(const Ipp8u *pSrc, int len, IppsHMACState *pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSrc  
Pointer to the buffer containing a part of the whole message.

len  
The length of the actual part of the message in bytes.

pCtx  
Pointer to the IppsHMACState context.
Description
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
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.
ippStsLengthErr Indicates an error condition if the length of the input data stream is
less than zero.

HMAC_Final
Completes computation of the HMAC value.

Syntax
IppStatus ippsHMAC_Final(Ipp8u *pMD, int mdLen, IppsHMACState *pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pMD Pointer to the resultant HMAC value.
mdLen Specified HMAC length.
pCtx Pointer to the IppsHMACState context.

Description
The function completes calculation of the digest value and stores the result at the memory location specified
by pMD.

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.
Indicates an error condition if \( mdLen \) is less than one or greater than the length of the hash value.

**HMAC_GetTag**

*Computes the current HMAC value of the processed part of the message.*

**Syntax**

\[
\text{IppStatus ippsHMAC_GetTag(Ipp8u* } \ pMD, \ \text{int } \ mdLen, \ \text{const IppsHMACState* } \ pCtx)\;
\]

**Include Files**

ippcp.h

**Domain Dependencies**

**Headers:** ippcore.h

**Libraries:** ippcore.lib

**Parameters**

- **\( pMD \)**: Pointer to the authentication tag.
- **\( mdLen \)**: The length of the tag (in bytes).
- **\( pCtx \)**: Pointer to the IppsHMACState context.

**Description**

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**

- **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 \( mdLen \) < 1 or \( mdLen \) exceeds the maximal length of a particular digest.
- **ippStsContextMatchErr**: Indicates an error condition if the context parameter does not match the operation.

**HMAC_Message**

*Computes the HMAC value of an entire message.*

**Syntax**

\[
\text{IppStatus ippsHMAC_Message(const Ipp8u *pMsg, int } \ \text{msgLen}, \ \text{const Ipp8u *pKey, int } \ \text{keyLen, Ipp8u *pMD, int } \ \text{mdLen, IppHashAlgId hashAlg});
\]

**Include Files**

ippcp.h
Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- pMsg: Pointer to the input message.
- msgLen: Message length in bytes.
- pKey: Pointer to the user-supplied key.
- keyLen: Key length in bytes.
- pMD: Pointer to the resultant HMAC value.
- mdLen: Specified HMAC length.
- hashAlg: Identifier of the hash algorithm.

Description

The function takes the input secret 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 pMD of the specified length mdLen. The hashAlg parameter defines the hash algorithm applied and can take one of the values listed in table Supported Hash Algorithms.

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 msgLen is less than zero or mdLen is less than one or greater than the length of the hash value.
- ippStsNotSupportedModeErr: Indicates an error condition if the hashAlg parameter does not match any value of IppHashAlg listed in table Supported Hash Algorithms.

CMAC Functions

The Intel IPP CMAC primitive functions use CMAC schemes based on block ciphers described in the Symmetric Cryptography Primitive Functions chapter.

A CMAC scheme is implemented as a set of primitive functions.

Typical application code for computing CMAC of an input message stream should follow the sequence of operations as outlined below:

1. Call the function AES_CMACGetSize to get the size required to configure the IppsAES_CMACState context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the function AES_CMACInit to initialize the context.
3. Keep calling the function AES_CMACUpdate to update the MAC value of the incoming message stream in the queue till its completion. To determine the current MAC value, call AES_CMACGetTag between each two calls to AES_CMACUpdate.
4. Call the function AES_CMACFinal to complete computation of the MAC value of the streaming message and prepare the context for computation of MAC of another message.
5. Clean up secret data stored in the context.
6. Call the operating system memory free service function to release the IppsAES_CMACState context.

**AES_CMACGetSize**

*Gets the size of the IppsAES_CMACState context.*

**Syntax**

IppStatus ippsAES_CMACGetSize(int *pSize);

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- **pSize**: Pointer to the IppsAES_CMACState context.

**Description**

This function gets the size of the IppsAES_CMACState 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.

**AES_CMACInit**

*Initializes user-supplied memory as IppsAES_CMACState context for future use.*

**Syntax**

IppStatus ippsAES_CMACInit(const Ipp8u* pKey, int keyLen, IppsAES_CMACState* pState, int ctxSize);

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- **pKey**: Pointer to the AES key.

- **keyLen**: Key bytestream length (in bytes) defined by the IppsAESKeyLength enumerator.
pState

Pointer to the memory buffer being initialized as IppsAES_CMACState context.

ctxSize

Available size of the buffer.

Description

This function initializes the memory at the address of pState as the IppsAES_CMACState context. In addition, the function uses the key to provide all necessary key material for both encryption and decryption operations.

NOTE

If the pKey pointer is NULL, the function initializes the context with the zero key, which can help you to clean up the actual secret before releasing the context.

Return Values

ippStsNoErr

Indicates no error. Any other value indicates an error or warning.

ippStsNullPtrErr

Indicates an error condition if the pState pointer is NULL.

ippStsLengthErr

Indicates an error condition if keyLen is not equal to 16, 24, or 32.

ippStsMemAllocErr

Indicates an error condition if the allocated memory is insufficient for the operation.

See Also

Data Security Considerations

AES_CMACUpdate

Updates the MAC value depending on the current input message stream of the specified length.

Syntax

IppStatus ippsAES_CMACUpdate(const Ipp8u *pSrc, int len, IppsAES_CMACState* pState);
Description
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 AES cipher to transform the input block into a new chaining MAC 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.

AES_CMACFinal
Completes computation of the MAC value.

Syntax
IppStatus ippsAES_CMACFinal(Ipp8u *pMD, int mdLen, IppsAES_CMACState *pState);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pMD Pointer to the MAC value.
mdLen Specified length of the MAC.
pState Pointer to the IppsAES_CMACState context.

Description
The function completes calculation of the MAC of a message, stores the result in the memory at the address of pMD, and prepares the context for computation of the MAC of another message.

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 mdLen is less than 1 or greater than cipher's data block length.
ippStsContextMatchErr Indicates an error condition if the context parameter does not match the operation.
AES_CMACGetTag

_Computes the MAC value of the processed part of the message._

**Syntax**

IppStatus ippsAES_CMACGetTag(Ipp8u* pMD, int mdLen, const IppsAES_CMACState *pState);

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pMD**
  - Pointer to the MAC value.
- **mdLen**
  - Specified length of the MAC.
- **pState**
  - Pointer to the IppsAES_CMACState context.

**Description**

The function computes the MAC value based on the current context. A call to this function retains the possibility to update the MAC 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 `mdLen` is less than 1 or greater than cipher's data block length.
- ippStsContextMatchErr
  - Indicates an error condition if the context parameter does not match the operation.
Public Key Cryptography
Functions

Big Number Arithmetic

This section describes primitives for performing arithmetic operations with integer big numbers of variable length.

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

\[ r = \sum_{0 \leq i < length} rp[i] \times 2^{32i}. \]

This section uses the following definition for the sign of an integer big number:

```c
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, integers pointed to by `a`, `b`, and `r` are all of `(n*32)` bits.

**BigNumGetSize**

*Gets the size of the* `IppsBigNumState` *context in bytes.*

**Syntax**

```c
IppStatus ippsBigNumGetSize(int length, int *size);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

*Headers:* `ippcore.h`

*Libraries:* `ippcore.lib`
Parameters

length

Integer big number length in Ipp32u.

size

Size of the buffer in bytes required for initialization.

Description

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

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 length is less than or equal to 0.

BigNumInit

Initializes context and partitions allocated buffer.

Syntax

IppStatus ippsBigNumInit(int length, IppsBigNumState *b);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

length

Size of the big number for the context initialization.

b

Pointer to the supplied buffer used to store the initialized context IppsBigNumState.

Description

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

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 length is less than or equal to 0.

See Also

Data Security Considerations
**Set_BN**

*Defines the sign and value of the context.*

**Syntax**

```
IppStatus ippsSet_BN(IppsBigNumSGN sgn, int length, const Ipp32u *data, IppsBigNumState *x);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- `sgn`: Sign of `IppsBigNumState *x`.
- `length`: Array length of the input data.
- `data`: Data array.
- `x`: On output, the context `IppsBigNumState` updated with the input data.

**Description**

The function defines the sign and value for `IppsBigNumState *x` with the specified inputs `IppsBigNumSGN sgn` and `const Ipp32u *data`.

**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 `length` is less than or equal to 0.
- `ippStsOutOfRangeErr`: Indicates an error condition if `length` is more than the size of `IppsBigNumState *x`.
- `ippStsBadArgErr`: 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.

```c
ippiBigNumState* New_BN(int size, const Ipp32u* pData=0){
    // get the size of the Big Number context
    int ctxSize;
    ippiBigNumGetSize(size, &ctxSize);
    // allocate the Big Number context
    IippiBigNumState* pBN = (IippiBigNumState*) (new Ipp8u [ctxSize] );
    // and initialize one
    ippiBigNumInit(size, pBN);
    // if any data was supplied, then set up the Big Number value
    if(pData)
        ippiSet_BN(IippiBigNumPOS, 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**

```c
IppStatus ippiSetOctString_BN(const Ipp8u* pOctStr, int strLen, IippiBigNumState* pBN);
```

**Include Files**

ippcp.h

**Domain Dependencies**

- **Headers:** ippcore.h
- **Libraries:** ippcore.lib

**Parameters**

- `pOctStr`  
  Pointer to the input octet string.
- `strLen`  
  Octet string length in bytes.
- `pBN`  
  Pointer to the context of the output Big Number.

**Description**

This function converts octet string into a positive Big Number.
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.

ippStsLengthErr Indicates an error condition if specified `strLen` 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.

```c
void Set_BN_sample(void){
    // desired value of Big Number is 0x123456789abcdef0fedcba9876543210
    Ipp8u desiredBNvalue[] = "\x12\x34\x56\x78\x9a\xbc\xde\xf0"
        "\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
    ippsSetOctString_BN(desiredBNvalue, sizeof(desiredBNvalue)-1, pBN);

    Type_BN("Big Number value is:
    ", pBN);
}
```

GetSize_BN

*Returns the maximum length of the integer big number the structure can store.*

Syntax

```c
IppStatus ippsGetSize_BN(const IppsBigNumState *b, int *size);
```

Include Files

` IPPCP.H`

Domain Dependencies

**Headers:** `ippcore.h`

**Libraries:** `ippcore.lib`
Parameters

\[ b \quad \text{Integer big number of the data type IppsBigNumState.} \]
\[ size \quad \text{Maximum length of the integer big number.} \]

Description

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

\[
\text{IppStatus ippsGet_BN(IppsBigNumSGN } \ast \text{sgn, int } \ast \text{length, Ipp32u } \ast \text{data, const IppsBigNumState } \ast \text{x);}
\]

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- \( sgn \): Sign of IppsBigNumState \( \ast x \).
- \( length \): Array length of the input data.
- \( data \): Data array.
- \( x \): Integer big number of the context IppsBigNumState.

Description

The function extracts the sign and value of the integer big number from the input structure.

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.
**ExtGet_BN**

*Extracts the specified combination of the sign, data length, and value characteristics of the integer big number from the input structure.*

**Syntax**

```c
IppStatus ippsExtGet_BN(IppsBigNumSGN *pSgn, int *pLengthInBits, Ipp32u *pData, const IppsBigNumState *pBN);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pSgn**
  - Pointer to the sign of IppsBigNumState *pBN.

- **pLengthInBits**
  - Pointer to the length of *pData in bits.

- **pData**
  - Pointer to the data array.

- **pBN**
  - Pointer to the integer big number context IppsBigNumState.

**Description**

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,

- `ippsExtGet_BN(&sgn, 0,0, pBN);` extracts only the sign
- `ippsExtGet_BN(0, &dataLen, 0, pBN);` extracts only the data length
- `ippsExtGet_BN(&sgn, &dataLen, 0, pBN);` extracts the sign and data length
- `ippsExtGet_BN(0,0,0, pBN);` does nothing
- `ippsExtGet_BN(&sgn, &dataLen, pData, pBN);` does exactly what Get_BN does.

**Return Values**

- **ippStsNoErr** Indicates no error. Any other value indicates an error or warning.
- **ippStsNullPtrErr** Indicates an error condition if the pointer to the integer big number of the context is NULL.
- **ippStsContextMatchErr** 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 ippsRef_BN(IppsBigNumSGN *sgn, int *bitSize, Ipp32u** const ppData, const IppsBigNumState *x);

Include Files

ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

sgn  Sign of IppsBigNumState *x.
bitSize  Length of the integer big number in bits.
ppData  Pointer to the data array.
x  Integer big number of the context IppsBigNumState.

Description

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

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.

GetOctString_BN

Converts a positive Big Number into octet String.

Syntax

IppStatus ippsGetOctString_BN(Ipp8u* pOctStr, int strLen, const IppsBigNumState* pBN);

Include Files

ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pOctStr  Pointer to the input octet string.
Octet string length in bytes.

Pointer to the context of the input Big Number.

**Description**

This function converts a positive Big Number into the octet string.

**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.
- **ippStsLengthErr**: Indicates an error condition if specified `pOctStr` is insufficient in length.
- **ippStsRangeErr**: Indicates an error condition if Big Number is negative.

**Example**

The code example below types a big number.

```c
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<<setfill('0')<<setw(2)<<(int)bnValue[n];
    cout<<endl;
    delete [] bnValue;
}
```

---

**Cmp_BN**

*Compares two Big Numbers.*

**Syntax**

```c
IppStatus ippsCmp_BN(const IppsBigNumState *pA, const IppsBigNumState *pB, Ipp32u *pResult);
```

**Include Files**

ippcp.h
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\begin{itemize}
\item \texttt{pA} \hspace{1cm} Pointer to the context of the Big Number A.
\item \texttt{pB} \hspace{1cm} Pointer to the context of the Big Number B.
\item \texttt{pResult} \hspace{1cm} Pointer to the result of the comparison.
\end{itemize}

Description
This function compares Big Numbers A and B and sets up the result according to the following conditions:
\begin{itemize}
\item if \( A == B \), then \( *\texttt{pResult} = \text{IS\_ZERO} \)
\item if \( A > B \), then \( *\texttt{pResult} = \text{GREATER\_THAN\_ZERO} \)
\item if \( A < B \), then \( *\texttt{pResult} = \text{LESS\_THAN\_ZERO} \)
\end{itemize}

Return Values
\begin{itemize}
\item \texttt{ippStsNoErr} \hspace{1cm} Indicates no error. Any other value indicates an error or warning.
\item \texttt{ippStsNullPtrErr} \hspace{1cm} Indicates an error condition if any of the specified pointers is NULL.
\item \texttt{ippStsContextMatchErr} \hspace{1cm} Indicates an error condition if the context parameter does not match the operation.
\end{itemize}

CmpZero_BN
\textit{Checks the value of the input data field.}

Syntax
\begin{verbatim}
IppStatus ippsCmpZero_BN(const IppsBigNumState *b, Ipp32u *result);
\end{verbatim}

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\begin{itemize}
\item \texttt{b} \hspace{1cm} Integer big number of the data type \texttt{IppsBigNumState}.
\item \texttt{result} \hspace{1cm} Indicates whether the input integer big number is positive, negative, or zero.
\end{itemize}

Description
The function scans the data field of the input \texttt{const IppsBigNumState *b} and returns
\begin{itemize}
\item \texttt{IS\_ZERO} if the value held by \texttt{IppsBigNumState *b} is zero
\item \texttt{GREATER\_THAN\_ZERO} if the input is more than zero
\end{itemize}
• LESS_THAN_ZERO if the input is less than zero.

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.

Add_BN
Adds two integer big numbers.

Syntax
IppStatus ippsAdd_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *r);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

a First integer big number of the data type IppsBigNumState.
b Second integer big number of the data type IppsBigNumState.
r Addition result.

Description
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) ← (*a) + (*b).

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.
ippStsOutOfRangeErr Indicates an error condition if the size of r 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.

```c
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), bnuB);

    // define Big Number R
    int sizeR = max(sizeof(bnuA), sizeof(bnuB));
    IppsBigNumState* bnR = New_BN(1+sizeR/sizeof(Ipp32u));

    // R = A+B
    ippsAdd_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**

```c
IppStatus ippsSub_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState * r);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **a**
  - First integer big number of the data type IppsBigNumState.
- **b**
  - Second integer big number of the data type IppsBigNumState.
Description
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
- **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.
- **ippStsOutOfRangeErr**: Indicates an error condition if \(\text{IppsBigNumState} *r\) is smaller than the result data length.

**NOTE**
The function executes only under the condition that size of \(\text{IppsBigNumState} *r\) is not less than either the length of \(\text{IppsBigNumState} *a\) or that of \(\text{IppsBigNumState} *b\).

**Mul_BN**
*Multiplies two integer big numbers.*

Syntax
```c
IppStatus ippsMul_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState * r);
```

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
- **a**: Multiplicand of \(\text{IppsBigNumState}\).
- **b**: Multiplier of \(\text{IppsBigNumState}\).
- **r**: Multiplication result.

Description
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:
\(r \leftarrow a \times b\).
MAC_BN_I

Multiplies two integer big numbers and accumulates the result with the third integer big number.

Syntax

IppStatus ippsMAC_BN_I(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *r);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

a
Multiplicand of IppsBigNumState.

b
Multiplier of IppsBigNumState.

r
Multiplication result.

Description

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:

\[ r \leftarrow r + a \times b. \]

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.

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.
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 ippsDiv_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *q, IppsBigNumState *r);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Dividend of IppsBigNumState.</td>
</tr>
<tr>
<td>b</td>
<td>Divisor of IppsBigNumState.</td>
</tr>
<tr>
<td>q</td>
<td>Quotient of IppsBigNumState.</td>
</tr>
<tr>
<td>r</td>
<td>Remainder of IppsBigNumState.</td>
</tr>
</tbody>
</table>

Description
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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippStsNoErr</td>
<td>Indicates no error. Any other value indicates an error or warning.</td>
</tr>
<tr>
<td>ippStsNullPtrErr</td>
<td>Indicates an error condition if any of the specified pointers is NULL.</td>
</tr>
<tr>
<td>ippStsOutOfRangeErr</td>
<td>Indicates an error condition if IppsBigNumState<em>r is smaller than the length of IppsBigNumState</em>b or when the size of IppsBigNumState *q is smaller than the quotient result data length.</td>
</tr>
<tr>
<td>ippStsDivByZeroErr</td>
<td>Indicates an error condition if the zero divisor is attempted.</td>
</tr>
</tbody>
</table>
NOTE
The size of IppsBigNumState *q should not be less than (lengthof a) - (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);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
a
Integer big number of IppsBigNumState.
m
Modulus integer of IppsBigNumState.
r
Modular reduction result.

Description
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:
\[ r \leftarrow a \mod m \]

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.
ippStsOutOfRangeErr
Indicates an error condition if IppsBigNumState *r is smaller than the length of IppsBigNumState *m.
ippStsBadModulusErr
Indicates an error condition if the modulus IppsBigNumState *m 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**

```c
IppStatus ippsGcd_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState *g);
```

**Include Files**

ippcp.h

**Domain Dependencies**

*Headers:* ippcore.h
*Libraries:* ippcore.lib

**Parameters**

- `a` - First integer big number of IppsBigNumState.
- `b` - Second integer big number of IppsBigNumState.
- `g` - Greatest common divisor to `a` and `b`.

**Description**

The function computes the greatest common divisor (GCD) for two positive integer big numbers.

The following pseudocode represents this function:

\[
g \leftarrow \text{gcd} \left( a, b \right).
\]

**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.
- `ippStsOutOfRangeErr` - Indicates an error condition if IppsBigNumState *g is smaller than the length of IppsBigNumState *a or IppsBigNumState *b.

**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**

```c
IppStatus ippsModInv_BN(IppsBigNumState *e, IppsBigNumState *m, IppsBigNumState *d);
```

**Include Files**

ippcp.h
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
- $e$: Integer big number of IppsBigNumState.
- $m$: Modulus integer of IppsBigNumState.
- $d$: Multiplicative inverse.

Description
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 $d \times e \equiv 1 \pmod{m}$.

Return Values
- ippStsNoErr: Indicates no error. Any other value indicates an error or warning.
- ippStsBadArgErr: Indicates an error condition if $e$ is less than or equal to 0.
- ippStsNullPtrErr: Indicates an error condition if any of the specified pointers is NULL.
- ippStsBadModulusErr: 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.
- ippStsOutOfRangeErr: Indicates an error condition if IppsBigNumState $*d$ is smaller than the length of IppsBigNumState $*m$.

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.

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 $TR \mod n$.

For better results, functions included in the cryptography package use $R = 2^k$ 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 $n0$ 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 \( T = x^e \mod n \). Consider computing \( T = x^4 \mod n \), 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

\[
x = xR \mod n.
\]

Then compute the Montgomery reduction of

\[
XX
\]

using the function **MontMul** to generate

\[
T = xxR^{-1} \mod n.
\]

The Montgomery reduction of \( T^2 \mod n \) with respect to \( R \) is

\[
T^2 R^{-1} \mod n = (x^2 R^{-1})^2 R^{-1} \mod n = x^4 R \mod n.
\]

Further applying **MontMul** with this value and the value of 1 yields the desired result \( T = x^4 \mod n \).

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. Clean up secret data stored in the context.
8. Free the memory using an operating system memory free function, if needed.

**See Also**

**Data Security Considerations**
MontGetSize

*Gets the size of the IppsMontState context.*

**Syntax**

```c
IppStatus ippsMontGetSize(IppsExpMethod method, int length, int * size);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- `method`  
  Selected exponential method.

- `length`  
  Data field length for the modulus in Ipp32u chunks.

- `size`  
  Size of the buffer required for initialization.

**Description**

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**

- `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 `length` is less than or equal to 0.

MontInit

*Initializes the context and partitions the specified buffer space.*

**Syntax**

```c
IppStatus ippsMontInit(IppsExpMethod method, int length, IppsMontState * m);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib
Parameters

- **method**: Selected exponential method.
- **length**: Data field length for the modulus in Ipp32u chunks.
- **m**: Pointer to the context IppsMontState.

Description

The function initializes the *m* buffer as the IppsMontState context. The function then partitions the buffer using the selected modular exponential method in such a way as to carry up to \( \text{length} \times \text{sizeof}(\text{Ipp32u}) \)-bit big number modulus and execute various Montgomery modulus 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 \( \text{length} \) is less than or equal to 0.

See Also

Data Security Considerations

MontSet

Sets the input integer big number to a value and computes the Montgomery reduction index.

Syntax

```c
IppStatus ippsMontSet(const Ipp32u *n, int length, IppsMontState *m);
```

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **n**: Input big number modulus.
- **length**: The length of the modulus in Ipp32u chunks.
- **m**: Pointer to the context IppsMontState capturing the modulus and the least significant word of the multiplicative inverse \( Ni \).

Description

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 \( R^2 \equiv 1 \mod n \times Ni \).
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.

ippStsBadModulusErr
Indicates an error condition if the modulus is not a positive odd integer.

ippStsLengthErr
Indicates an error condition if \texttt{length} is less than or equal to 0.

ippStsOutOfRangeErr
Indicates an error condition if \texttt{length} is larger than \texttt{IppsMontState*} \texttt{m}.

MontGet
\textit{Extracts the big number modulus.}

Syntax
\begin{verbatim}
IppStatus ippsMontGet(Ipp32u *\textit{n}, int *\textit{length}, const IppsMontState *\textit{m});
\end{verbatim}

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
\begin{itemize}
\item \textit{m} \hspace{1cm} context \texttt{IppsMontState}.
\item \textit{n} \hspace{1cm} Modulus data field.
\item \textit{length} \hspace{1cm} Modulus data length in \texttt{Ipp32u} chunks.
\end{itemize}

Description
The function extracts the big number modulus from the input \texttt{IppsMontState *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.

MontForm
\textit{Converts input positive integer big number into Montgomery form.}

Syntax
\begin{verbatim}
IppStatus ippsMontForm(const IppsBigNumState* \textit{a}, IppsMontState* \textit{m}, IppsBigNumState* \textit{r});
\end{verbatim}
Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\(a\)  
Input integer big number within the range \([0, m - 1]\).

\(m\)  
Input big number modulus of \textit{IppsBigNumState}.

\(r\)  
Resulting Montgomery form \(r = a \cdot R \mod m\).

Description

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:

\[ r = a \cdot R \mod m. \]

Return Values

\texttt{ippStsNoErr}  
Indicates no error. Any other value indicates an error or warning.

\texttt{ippStsNullPtrErr}  
Indicates an error condition if any of the specified pointers is NULL.

\texttt{ippStsBadArgErr}  
Indicates an error condition if \(a\) is a negative integer.

\texttt{ippStsScaleRangeErr}  
Indicates an error condition if \(a\) is more than \(m\).

\texttt{ippStsOutOfRangeErr}  
Indicates an error condition if \textit{IppsBigNumState} \(*r\) is larger than \textit{IppsMontState} \(*m\).

\textbf{NOTE}

The size of \textit{IppsBigNumState} \(*r\) should not be less than the data length of the modulus \(m\).

Montmul

\textit{Computes Montgomery modular multiplication for positive integer big numbers of Montgomery form.}

Syntax

\texttt{IppStatus ippsMontMul(const IppsBigNumState *a, const IppsBigNumState *b, IppsMontState *m, IppsBigNumState *r);}  

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Parameters

- \(a\): Multiplicand within the range \([0, m - 1]\).
- \(b\): Multiplier within the range \([0, m - 1]\).
- \(m\): Modulus.
- \(r\): Montgomery multiplication result.

Description

The function computes the Montgomery modular multiplication for positive integer big numbers of Montgomery form with respect to the modulus \(\text{IppsMontState} \ast m\). As a result, \(\text{IppsBigNumState} \ast r\) holds the product.

The following pseudocode represents this function:

\[ r \leftarrow a \ast b \ast R^{-1} \mod m. \]

Return Values

- ippStsNoErr: Indicates no error. Any other value indicates an error or warning.
- ippStsBadArgErr: Indicates an error condition if \(a\) or \(b\) is a negative integer.
- ippStsNullPtrErr: Indicates an error condition if any of the specified pointers is NULL.
- ippStsScaleRangeErr: Indicates an error condition if \(a\) or \(b\) is more than \(m\).
- ippStsOutOfRangeErr: Indicates an error condition if \(\text{IppsBigNumState} \ast r\) is larger than \(\text{IppsMontState} \ast m\).

**NOTE**

The size of \(\text{IppsBigNumState} \ast r\) should not be less than the data length of the modulus \(m\).
Example of Using Montgomery Reduction Scheme Functions

Montgomery Multiplication

```c
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 multiplicant A
    Ipp32u bnuA = 12;
    IppsBigNumState* bnA = New_BN(1, &bnuA);
    // encode A into Montgomery form
    ippsMontForm(bnA, pMont, bnA);

    // define and init Big Number multiplicant A
    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 ippsMontExp(const IppsBigNumState *a, const IppsBigNumState *e, IppsMontState *m, IppsBigNumState *r);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- \(a\) Big number Montgomery integer within the range of \([0, m - 1]\).
- \(e\) Big number exponent.
- \(m\) Modulus.
- \(r\) Montgomery exponentiation result.

Description

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:

\[ r \leftarrow a^e \cdot R^{(e-1) \mod m} \]

Return Values

- ippStsNoErr Indicates no error. Any other value indicates an error or warning.
- ippStsBadArgErr Indicates an error condition if \(a\) or \(e\) is a negative integer.
- ippStsNullPtrErr Indicates an error condition if any of the specified pointers is NULL.
- ippStsScaleRangeErr Indicates an error condition if \(a\) or \(e\) is more than \(m\).
- ippStsOutOfRangeErr Indicates an error condition if IppsBigNumState *\(r\) is larger than IppsMontState *\(m\).

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.
This section describes functions that make 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. Clean up secret data stored in the context.
6. Free the memory allocated for the `IppsPRNGState` context by calling the operating system memory free service function.

See Also
Data Security Considerations

User’s Implementation of a Pseudorandom Number Generator

Both functions `ippsPRNGGen` and `ippsPRNGGen_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(_STDCALL *IppBitSupplier)(Ipp32u* pData, int nBits, void* pEbsParams);

Parameters

- `pData` Pointer to the output data.
- `nBits` 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

- `ippStsNoErr` Indicates no error. Any other value indicates an error or warning.
- `ippStsErr` Indicates an error condition.

`PRNGGetSize`

Gets the size of the `IppsPRNGState` context in bytes.
Syntax
IppStatus ippsPRNGGetSize(int *pSize);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pSize Pointer to the IppsPRNGState context size in bytes.

Description
The function gets the IppsPRNGState 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.

PRNGInit
*Initializes user-supplied memory as IppsPRNGState context for future use.*

Syntax
IppStatus ippsPRNGInit(int seedBits, IppsPRNGState* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
seedBits Size in bits for the seed value.
pCtx Pointer to the IppsPRNGState context being initialized.

Description
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 = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
• H0 = 0xC3D2E1F01032547698BADCFECDAB8967452301

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 seedBits is less than 1 or greater than 512.

See Also
Data Security Considerations

PRNGSetSeed
Sets up the seed value for the pseudorandom number generator.

Syntax
IppStatus ippsPRNGSetSeed(const IppsBigNumState* pSeed, IppsPRNGState* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSeed Pointer to the seed value being set up.
pCtx Pointer to the IppsPRNGState context.

Description
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

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.

NOTE
This function restarts the pseudorandom number generation process, which results in losing already generated pseudorandom numbers.
**PRNGGetSeed**

*Extracts the seed value of the pseudorandom number generator from the context structure.*

**Syntax**

```c
IppStatus ippsPRNGGetSeed(const IppsPRNGState* pCtx, IppsBigNumState* pSeed);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

- Headers: `ippcore.h`
- Libraries: `ippcore.lib`

**Parameters**

- `pCtx`: Pointer to the IppsPRNGState context.
- `pSeed`: Pointer to the seed value.

**Description**

The function extracts the seed value of the pseudorandom number generator from the IppsPRNGState context structure into a big number.

**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 `pSeed` is not a IppsBigNumState structure or `pCtx` is not a IppsPRNGState structure.
- `ippOutOfRangeErr`: Indicates an error condition if the length of the actual seed exceeds `pSeed`.

---

**PRNGSetAugment**

*Sets the initial state with the given input entropy for the pseudorandom number generation.*

**Syntax**

```c
IppStatus ippsPRNGSetAugment(const IppsBigNumState* pAugment, IppsPRNGState* pCtx);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

- Headers: `ippcore.h`
- Libraries: `ippcore.lib`
Parameters

\textbf{pAugment}

Pointer to the entropy augment value being set up.

\textbf{pCtx}

Pointer to the IppsPRNGState context.

Description

The function resets entropy augment value with the supplied value of the \textit{seedBits} bit length. The supplied big number should be created prior to the function call using the appropriate Big Number Arithmetic functions (see \textit{Example 5-1}).

Return Values

\textbf{ippStsNoErr}

Indicates no error. Any other value indicates an error or warning.

\textbf{ippStsNullPtrErr}

Indicates an error condition if any of the specified pointers is NULL.

\textbf{ippStsContextMatchErr}

Indicates an error condition if the context parameter does not match the operation.

PRNGSetModulus

\textit{Sets the initial state with the given input modulus for the pseudorandom number generation.}

Syntax

\texttt{IppStatus ippsPRNGSetModulus(const IppsBigNumState* pModulus, IppsPRNGState* pCtx);}
PRNGSetH0
Sets the initial state with the given input IV for the SHA-1 algorithm.

Syntax
IppStatus ippsPRNGSetH0(const IppsBigNumState* pH0, IppsPRNGState* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pH0 Pointer to the initial hash value being set up.
pCtx Pointer to the IppsPRNGState context.

Description
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 5-1).

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.

PRNGen
Generates a pseudorandom unsigned Big Number of the specified bitlength.

Syntax
IppStatus ippsPRNGen(Ipp32u* pRandBNU, int nBits, void* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib
Parameters

- **pRandBNU**
  - Pointer to the output pseudorandom unsigned integer big number.
- **nBits**
  - Number of the generated pseudorandom bit.
- **pCtx**
  - Pointer to the IppsPRNGState context.

Description

The function generates pseudorandom unsigned integer big number of the specified *nBits* length.

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.
- **ippStsLengthErr**
  - Indicates an error condition if *nBits* is less than 1.

**PRNGen_BN**

Generates a pseudorandom positive Big Number of the specified bitlength.

Syntax

```c
IppStatus ippsPRNGen_BN(IppsBigNumState* pRandBN, int nBits, void* pCtx);
```

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pRandBN**
  - Pointer to the output pseudorandom Big Number.
- **nBits**
  - Number of the generated pseudorandom bit.
- **pCtx**
  - Pointer to the IppsPRNGState context.

Description

The function generates pseudorandom positive Big Number of the specified *nBits* length.

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.
Indicates an error condition if the context parameter does not match the operation.
Indicates an error condition if \textit{nBits} is less than 1.

**Example of Using Pseudorandom Number Generation Functions**

**Find Pseudorandom Co-primes**

```c
void FindCoPrimes(void){
    int size;

    // define Pseudo Random Generator (default settings)
    ippStsContextMatchErr = ippStsContextMatchErr;
    IppsPRNGState* pPrng = (IppsPRNGState*)new Ipp8u [size] ;
    ippStsLengthErr;

    // 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; 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\textsuperscript{®} Integrated Performance Primitives (Intel\textsuperscript{®} IPP) functions for prime number generation.
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

\[
\lim_{x \to \infty} \frac{\pi(x)}{x / (\ln x)} = 1.
\]

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

\[
P_{k, t} \leq \frac{2^k}{2^t - 1} \left( 1 - \frac{1}{2^t} \right) \text{ for } t = 2, k \geq 88, \text{ or } 3 \leq t < 2, k \geq 21.
\]

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 \texttt{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 structured 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 \texttt{PrimeGetSize} to get the size required to configure the \texttt{IppsPrimeState} context.
2. Allocate memory through the operating system memory allocation function and configure the \texttt{IppsPrimeState} context by calling the function \texttt{PrimeInit}.
3. Generate a probable prime number of the specified bitlength by calling the function \texttt{PrimeGen_BN}. If the returned \texttt{IppStatus} is \texttt{ippStsInsufficientEntropy}, then change the parameters of the pseudorandom generator and call the function \texttt{PrimeGen_BN} again.
4. Clean up secret data stored in the context.
5. Free the memory allocated to the \texttt{IppsPrimeState} context by calling the operating system memory-free service function.

**See Also**

Data Security Considerations

**PrimeGetSize**

*Gets the size of the* \texttt{IppsPrimeState} *context in bytes.*

**Syntax**

\[
\text{IppStatus ippsPrimeGetSize(int } \text{nMaxBits, int* } \text{pSize);}\]

**Include Files**

\[\text{ippcp.h}\]
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\( n_{\text{MaxBits}} \)
Maximum length of the probable prime number in bits.

\( p_{\text{Size}} \)
Pointer to the IppsPrimeState context size in bytes.

Description
The function gets the IppsPrimeState context size in bytes and stores it in \( p_{\text{Size}} \).

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 \( n_{\text{MaxBits}} \) is less than 1.

PrimeInit

Initializes user-supplied memory as IppsPrimeState context for future use.

Syntax

IppStatus ippsPrimeInit(int \( n_{\text{MaxBits}} \), IppsPrimeState* \( p_{\text{Ctx}} \));

Include Files

ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\( n_{\text{MaxBits}} \)
Maximum length of the probable prime number in bits.

\( p_{\text{Ctx}} \)
Pointer to the IppsPrimeState context being initialized.

Description
The function initializes the memory pointed by \( p_{\text{Ctx}} \) as the IppsPrimeState 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.

ippStsLengthErr
Indicates an error condition if \( n_{\text{MaxBits}} \) is less than 1.
See Also
Data Security Considerations

PrimeGen_BN
Generates a random probable prime number of the specified bitlength.

Syntax
IppStatus ippsPrimeGen_BN(IppsBigNumState* pPrime, int nBits, int nTrials,
IppsPrimeState* pCtx, IppBitSupplier rndFunc, void* pRndParam);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pPrime            Big number to store the generated number in.
nBits             Target bitlength for the desired probable prime number.
nTrials           Security parameter specified for the Miller-Rabin probable primality.
pCtx              Pointer to the IppsPrimeState context.
rndFunc           Specified Random Generator.
pRndParam          Pointer to the Random Generator context.

Description
The function employs the rndFuncRandom Generator specified by the user to generate a random probable prime number of the nBits length and stores the generated probable prime number in the pPrime big number. The generated probable prime number is further validated by the Miller-Rabin primality test scheme with the specified security parameter nTrials.

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 nBits is less than 1.
ippStsContextMatchErr Indicates an error condition if the context parameter does not match the operation.
ippStsBadArgErr    Indicates an error condition if nTrials is less than 1.
ippStsOutOfRangeErr Indicates an error condition if nBits > nMaxBits (see PrimeGetSize and PrimeInit)
ippStsInsufficientEntropy Indicates a warning condition if prime generation fails due to poor choice of entropy.
PrimeTest_BN

Tests the given big number for being a probable prime.

Syntax

IppStatus ippsPrimeTest_BN(const IppsBigNumState* pPrime, int nTrials, Ipp32u* pResult, IppsPrimeState* pCtx, IppBitSupplier rndFunc, void* pRndParam);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pPrime The big number to test.
nTrials Security parameter specified for the Miller-Rabin probable primality.
pResult Pointer to the result of the primality test.
pCtx Pointer to the IppsPrimeState context.
rndFunc Specified Random Generator.
pRndParam Pointer to the Random Generator context.

Description

The function uses the Miller-Rabin probabilistic primality test scheme with the given security parameter to test whether the given big number 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 whether the input probable prime passes the Miller-Rabin test.

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.
ippStsBadArgErr Indicates an error condition if nTrials 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);
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

-nBits
Target bitlength for the desired probable prime number.

-nTrials
Security parameter specified for the Miller-Rabin probable primality.

-pCtx
Pointer to the IppsPrimeState context.

-rndFunc
Specified Random Generator.

-pRndParam
Pointer to the Random Generator context.

Description
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

-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 nBits is less than 1.

-ippStsContextMatchErr
Indicates an error condition if the context parameter does not match the operation.

-ippStsBadArgErr
Indicates an error condition if nTrials is less than 1.

-ippStsOutOfRangeErr
Indicates an error condition if nBits > nMaxBits (see PrimeGetSize and PrimeInit)

-ippStsInsufficientEntropy
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 ippsPrimeTest(int nTrials, Ipp32u *pResult, IppsPrimeState* pCtx, IppBitSupplier rndFunc, void* pRndParam);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
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 \texttt{rndFunc} Random Generator. The function sets up the \texttt{*pResult} as \texttt{IS_PRIME} or \texttt{IS_COMPOSITE} to show if the input probable prime passes the Miller-Rabin test.

**Return Values**

- \texttt{ippStsNoErr} Indicates no error. Any other value indicates an error or warning.
- \texttt{ippStsNullPtrErr} Indicates an error condition if any of the specified pointers is NULL.
- \texttt{ippStsContextMatchErr} Indicates an error condition if the context parameter does not match the operation.
- \texttt{ippStsBadArgErr} Indicates an error condition if \texttt{nTrials} is less than 1.

**PrimeSet**

*Sets the Big Number for primality testing.*

**Syntax**

\begin{verbatim}
IppStatus ippsPrimeSet(const Ipp32u* pBNU, int nBits, IppsPrimeState* pCtx);
\end{verbatim}

**Include Files**

\begin{verbatim}
ippcp.h
\end{verbatim}

**Domain Dependencies**

- **Headers:** ippcore.h
- **Libraries:** ippcore.lib

**Parameters**

- \texttt{pBNU} Pointer to the unsigned integer big number.
- \texttt{nBits} Unsigned integer big number length in bits.
- \texttt{pCtx} Pointer to the IppsPrimeState context.

**Description**

The function sets a probable prime number and its length for the probabilistic primality test.
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 \( nBits \) is less than 1.

*ippStsContextMatchErr*  
Indicates an error condition if the context parameter does not match the operation.

*ippStsOutOfRangeErr*  
Indicates an error condition if \( nBits \) is too large to fit \( pCtx \).

**PrimeSet_BN**

*Sets the Big Number for primality testing.*

**Syntax**

\[
\text{IppStatus ippsPrimeSet_BN(const IppsBigNumState* } pBN, \text{ IppsPrimeState* } pCtx); 
\]

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h  
Libraries: ippcore.lib

**Parameters**

- \( pBN \)  
  Pointer to the Big Number context.

- \( pCtx \)  
  Pointer to the IppsPrimeState context.

**Description**

The function sets the Big Number for probabilistic primality test.

**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.

*ippStsOutOfRangeErr*  
Indicates an error condition if the Big Number is too large to fit \( pCtx \).

**PrimeGet**

*Extracts the probable prime unsigned integer big number.*

**Syntax**

\[
\text{IppStatus ippsPrimeGet(Ipp32u* } pBNU, \text{ int *pSize, const IppsPrimeState* } pCtx); 
\]
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

\texttt{pBN} \quad \text{Pointer to the unsigned integer big number.}

\texttt{pSize} \quad \text{Pointer to the length of the unsigned integer big number.}

\texttt{pCtx} \quad \text{Pointer to the IppsPrimeState context.}

Description
The function extracts the probable prime number from \texttt{*pCtx} context and stores it into the specified unsigned integer big number.

Return Values

\texttt{ippStsNoErr} \quad \text{Indicates no error. Any other value indicates an error or warning.}

\texttt{ippStsNullPtrErr} \quad \text{Indicates an error condition if any of the specified pointers is NULL.}

\texttt{ippStsContextMatchErr} \quad \text{Indicates an error condition if the context parameter does not match the operation.}

\textbf{PrimeGet\_BN}

\textit{Extracts the probable prime positive Big Number.}

Syntax

\texttt{IppStatus ippsPrimeGet\_BN(IppsBigNumState* pBN, const IppsPrimeState *pCtx);}
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.

**ippStsOutOfRangeErr** 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

```c
int PrimeGen_sample(void){   int error = 0;

    int ctxSize;
    // define 256-bit Prime Generator
    int maxBitSize = 256;
    ippsPrimeGetSize(256, &ctxSize);
    IppsPrimeState* pPrimeG = (IppsPrimeState*)( new Ipp8u [ctxSize] );
    ippsPrimeInit(256, pPrimeG);

    // define Pseudo Random Generator (default settings)
    ippsPRNGGetSize(&ctxSize);
    IppsPRNGState* pRand = (IppsPRNGState*)(new Ipp8u [ctxSize] );
    ippsPRNGInit(160, pRand);

    do {
        Ipp32u result;

        // test primality of the value (known in advance)
        BigNumber P1("0xDB7C2ABF62E35E668076BEAD208B");
        ippsPrimeTest_BN(P1, 50, &result, pPrimeG, ippsPRNGen, pRand);
        error = IPP_IS_PRIME!=result;
        if(error) {
            cout <<"Primality of the known prime isn't confirmed\n"
                  break;
        } else cout <<"Primality of the known prime is confirmed\n";

        // generate 256-bit prime
        BigNumber P(0, 256/8);
        while( ippStsNoErr != ippsPrimeGen_BN(P, 256, 50, pPrimeG, ippsPRNGen, pRand) ) ;
        // and test it
        ippsPrimeTest_BN(P, 50, &result, pPrimeG, ippsPRNGen, pRand);
        error = IPP_IS_PRIME!=result;
        if(error) {
            cout <<"Primality of the generated number isn't confirmed\n"
                  break;
        } else cout <<"Primality of the generated number is confirmed\n";
    } while(0);

    delete [] (Ipp8u*)pRand;
    delete [] (Ipp8u*)pPrimeG;

    return error;
}
```
rsa

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. This set of primitives offers a flexible user interface that enables scalability of the RSA crypto key size with the limit of up to 4096 bits.

According to [PKCS 1.2.1], a de facto standard for RSA implementations, a pair of keys (public and private) defines forward and inverse transforms of text (or operations on a public and secret key). Mathematical expressions for the forward and inverse transforms are similar. If \( x \) is plain text and \( y \) is the corresponding ciphertext, the mathematical expressions are as follows:

- \( y = x^e \mod n \) for the forward transform, or encryption
- \( x = y^d \mod n \) for the inverse transform, or decryption

In these expressions, \( e \) is the public exponent, \( d \) is the private exponent, and \( n \) is the RSA modulus. To enable direct and inverse transforms, a mathematical relationship exists between these values.

The \( (n,e) \) pair is called the public key. With the known modulus \( n \), the public or private exponent determines whether the RSA cryptosystem is public or private. Intel IPP supports these, interrelated, representations of the private key:

- **Private key type 1** is the \( (n,d) \) pair.
- **Private key type 2** is the \( (p,q,d_P,d_Q,q_{Inv}) \) quintuple (for details, see [PKCS 1.2.1]).

This representation speeds computations by using the Chinese Remainder Theorem (CRT).

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
You can use the primitives to build an RSA cryptographic system with the supplied randomized seed and stimulus. The function **RSA_GenerateKeys** generates key components for the desired RSA cryptographic system.

**RSA Primitives** and RSA-based schemes (**RSA-OAEP Scheme Functions** and **RSA-SSA Scheme Functions**) use IppsRSAPublicKeyState or IppsRSAPrivateKeyState context, which is initialized in a call to the **RSA_InitPublicKey**, **RSA_InitPrivateKeyType1**, or **RSA_InitPrivateKeyType2** function, as an operational vehicle carrying the RSA public or private keys.

**RSAGetSizePublicKey**, **RSAGetSizePrivateKeyType1**, **RSAGetSizePrivateKeyType2**

*Get the size of the* IppsRSAPublicKeyState or IppsRSAPrivateKeyState context.

**Syntax**

```c
IppStatus ippsRSA_GetSizePublicKey(int rsaModulusBitSize, int publicExpBitSize, int* pKeySize);
IppStatus ippsRSA_GetSizePrivateKeyType1(int rsaModulusBitSize, int privateExpBitSize, int* pKeySize);
```
IppStatus ippRSA_GetSizePrivateKeyType2(int factorPBitSize, int factorQBitSize, int* pKeySize);

**Include Files**
ippcp.h

**Domain Dependencies**
Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- `rsaModulusBitSize`: Length of the RSA system in bits (that is, the length of the composite RSA modulus n in bits).
- `publicExpBitSize`: Length of the RSA public exponent in bits (that is, the length of the e component of the RSA public key).
- `privateExpBitSize`: Length of the RSA private exponent in bits (that is, the length of the d component of the RSA private key type 1).
- `factorPBitSize, factorQBitSize`: Length in bits of the p and q factors of the RSA modulus n = p*q.
- `pKeySize`: Pointer to the IppsRSAPublicKeyState context size in bytes.

**Description**

These functions get the size of the IppsRSAPublicKeyState or IppsRSAPrivateKeyState context in bytes and stores it in *pKeySize. Call RSA_GetSizePublicKey to establish an RSA cryptosystem for encryption (or signature verification) operations. Call RSA_GetSizePrivateKeyType1 or RSA_GetSizePrivateKeyType2 to establish an RSA cryptosystem for decryption (or signature generation) operations. The choice between these two functions depends on the representation of the private key to be used.

**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.
- **ippStsNotSupportedModeErr**: Indicates an error condition if `rsaModulusBitSize < 32, rsaModulusBitSize > 4096, factorPBitSize + factorQBitSize < 32, factorPBitSize + factorQBitSize > 4096, factorPBitSize < 0, or factorQBitSize < 0.`
- **ippStsBadArgErr**: Indicates an error condition if `publicExpBitSize < 0 or publicExpBitSize > rsaModulusBitSize.`
- **ippStsBadArgErr**: For RSA_GetSizePublicKey, indicates an error condition if `publicExpBitSize < 0 or publicExpBitSize > rsaModulusBitSize.`
- **ippStsBadArgErr**: For RSA_GetSizePrivateKeyType1, indicates and error condition if `privateExpBitSize < 0 or privateExpBitSize > rsaModulusBitSize.`
- **ippStsBadArgErr**: For RSA_GetSizePrivateKeyType2, indicates and error condition if `factorPBitSize < 0, factorPBitSize < 0, or factorPBitSize < factorQBitSize.`
RSA_InitPublicKey, RSA_InitPrivateKeyType1, RSA_InitPrivateKeyType2

Initialize user-supplied memory as the IppsRSAPublicKeyState or IppsRSAPrivateKeyState context for future use.

Syntax

IppStatus ippsRSA_InitPublicKey(int rsaModulusBitSize, int publicExpBitSize, IppsRSAPublicKeyState* pKey, int keyCtxSize);
IppStatus ippsRSA_InitPrivateKeyType1(int rsaModulusBitSize, int privateExpBitSize, IppsRSAPrivateKeyState* pKey, int keyCtxSize);
IppStatus ippsRSA_InitPrivateKeyType2(int factorPBitSize, int FactorQBitSize, IppsRSAPrivateKeyState* pKey, int keyCtxSize);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

rsaModulusBitSize  Length of the RSA system in bits (that is, the length of the composite RSA modulus n in bits).
publicExpBitSize  Length of the RSA public exponent in bits (that is, the length of the e component of the RSA public key).
privateExpBitSize  Length of the RSA private exponent in bits (that is, the length of the d component of the type 1 RSA private key).
factorPBitSize, FactorQBitSize  Length in bits of the p and q factors of the RSA modulus n = p*q.
pKey  Pointer to the IppsRSAPublicKeyState or IppsRSAPrivateKeyState context being initialized. The context depends on the function. Available size in bytes of the memory buffer being initialized.
keyCtxSize

Description

These functions initialize the memory pointed by pKey as the IppsRSAPublicKeyState or IppsRSAPrivateKeyState context, depending on the function. To determine the size of the memory buffer, call the appropriate RSAGetSizePublicKey, RSAGetSizePrivateKeyType1, RSAGetSizePrivateKeyType2 function prior to calling any of these functions.

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.
ippStsNotSupportedModeErr  Indicates an error condition if rsaModulusBitSize < 32 or rsaModulusBitSize > 4096, factorPBitSize < 16 or factorPBitSize > 4096, or factorQBitSize < 16 or factorQBitSize > 4096.
Indicates an error condition if \( publicExpBitSize > rsaModulusBitSize \) or \( privateExpBitSize > rsaModulusBitSize \).

Indicates an error condition if the allocated memory is insufficient for the operation.

**See Also**

RSAGetSizePublicKey, RSAGetSizePrivateKeyType1, RSAGetSizePrivateKeyType2

**Data Security Considerations**

**RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2**

*Set up an RSA key in the existing RSA key context.*

**Syntax**

```c
IppStatus ippsRSA_SetPublicKey(const IppsBigNumState* pModulus, const IppsBigNumState* pPublicExp, IppsRSAPublicKeyState* pKey);
IppStatus ippsRSA_SetPrivateKeyType1(const IppsBigNumState* pModulus, const IppsBigNumState* pPrivateExp, IppsRSAPrivateKeyState* pKey);
IppStatus ippsRSA_SetPrivateKeyType2(const IppsBigNumState* pFactorP, const IppsBigNumState* pFactorQ, const IppsBigNumState* pCrtExpP, const IppsBigNumState* pCrtExpQ, const IppsBigNumState* pInverseQ, IppsRSAPrivateKeyState* pKey);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- `pModulus`:
  
  The composite RSA modulus \( n \).

- `pPublicExp`:
  
  The \( e \) component of the RSA public key.

- `pPrivateExp`:
  
  The \( d \) component of the type 1 RSA private key.

- `pFactorP`, `pFactorQ`:
  
  The \( p \) and \( q \) factors of the RSA modulus \( n = p \times q \).

- `pCrtExpP`, `pCrtExpQ`:
  
  The \( dP \) and \( dQ \) components of the quintuple \((p,q,dP,dQ,qInv)\), which defines a type 2 private key.

- `pInverseQ`:
  
  The \( qInv \) component of the quintuple \((p,q,dP,dQ,qInv)\).

- `pKey`:
  
  Pointer to the IppsRSAPublicKeyState or IppsRSAPrivateKeyState context.

**Description**

The `RSA_SetPublicKey` function sets up the RSA public key \((n,e)\) in the IppsRSAPublicKeyState context, that is, copies the \( n \) and \( e \) components supplied by the user into the context.

The `RSA_SetPrivateKeyType1` function sets up the RSA type 1 private key \((n,d)\) in the IppsRSAPrivateKeyState context, that is, copies the \( n \) and \( d \) components supplied by the user into the context.
The RSA_SetPrivateKeyType2 function sets up the RSA type 2 private key \((p,q,dP,dQ,qInv)\) in the IppsRSAPrivateKeyState context, that is, copies user-supplied \(p\) and \(q\) factors of the RSA composite modulus into the context, computes the rest of the key components, and copies them into the context:

- \(dP = q \mod (p-1)\)
- \(dQ = p \mod (q-1)\)
- \(qInv = 1/q \mod p\)

**Return Values**

- **ippStsNoErr**
  Indicates no error. Any other value indicates an error or warning.
- **ippStsNullPtrErr**
  Indicates an error condition if any of the specified pointers is NULL.
- **ippStsContextMatchErr**
  Indicates an error condition if any of the context parameters does not match the operation.
- **ippStsSizeErr**
  Indicates an error condition if the bit length of a key component specified by the \(pModulus\), \(pPublicExp\), \(pPrivateExp\), \(pFactorP\), or \(pFactorQ\) pointer exceeds the bit length specified at the initialization.
- **ippStsOutOfRangeErr**
  Indicates an error condition if any key component is not positive.

**RSA_GetPublicKey, RSA_GetPrivateKeyType1, RSA_GetPrivateKeyType2**

Extracts key components from an RSA key context.

**Syntax**

```c
IppStatus ippsRSA_GetPublicKey(IppsBigNumState* pModulus, IppsBigNumState* pPublicExp, const IppsRSAPublicKeyState* pKey);

IppStatus ippsRSA_GetPrivateKeyType1(IppsBigNumState* pModulus, IppsBigNumState* pPrivateExp, const IppsRSAPrivateKeyState* pKey);

IppStatus ippsRSA_GetPrivateKeyType2(IppsBigNumState* pFactorP, IppsBigNumState* pFactorQ, IppsBigNumState* pCrtExpP, IppsBigNumState* pCrtExpQ, IppsBigNumState* pInverseQ, const IppsRSAPrivateKeyState* pKey);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- **pModulus**
  The composite RSA modulus \(n\).
- **pPublicExp**
  The \(e\) component of the RSA public key.
- **pPrivateExp**
  The \(d\) component of the type 1 RSA private key.
- **pFactorP, pFactorQ**
  The \(p\) and \(q\) factors of the RSA modulus \(n = p*q\).
- **pCrtExpP, pCrtExpQ**
  The \(dP\) and \(dQ\) components of the quintuple \((p,q,dP,dQ,qInv)\).
The *qInv* component of the quintuple \((p,q,dP,dQ,qInv)\).

**pKey**

Pointer to the **IppsRSAPublicKeyState** or **IppsRSAPrivateKeyState** context.

**Description**

The **RSA_GetPublicKey** function extracts components of the RSA public key \((n,e)\) from the **IppsRSAPublicKeyState** context. The **RSA_GetPrivateKeyType1** and **RSA_GetPrivateKeyType2** functions extract components of the RSA private key of the respective type from the **IppsRSAPrivateKeyState** context.

To extract key components selectively, set pointers to the key components that do not need to be extracted to **NULL**.

**Return Values**

- **ippStsNoErr**: Indicates no error. Any other value indicates an error or warning.
- **ippStsNullPtrErr**: Indicates an error condition if any of the specified pointers is **NULL**.
- **ippStsContextMatchErr**: Indicates an error condition if any of the context parameters does not match the operation.
- **ippStsSizeErr**: Indicates an error condition if the bit length of any specified key component is not sufficient to hold the value.
- **ippStsIncompleteContextErr**: Indicates an error condition if the public or private key is not set up.

**NOTE**

While you can set up the public key or type 1 private key in a call to **RSA_SetPublicKey** or **RSA_SetPrivateKeyType1**, respectively, you can set up the type 2 private key in a call to either **RSA_SetPrivateKeyType2** or **RSA_GenerateKeys**.

**See Also**

**RSA_SetPublicKey**, **RSA_SetPrivateKeyType1**, **RSA_SetPrivateKeyType2**

**RSA_GenerateKeys**

**RSA_GetBufferSizePublicKey**, **RSA_GetBufferSizePrivateKey**

Get the size of a temporary scratch buffer for future use in RSA operations.

**Syntax**

```
IppStatus ippsRSA_GetBufferSizePublicKey(int* pBufferSize, const IppsRSAPublicKeyState* pKey);
IppStatus ippsRSA_GetBufferSizePrivateKey(int* pBufferSize, const IppsRSAPrivateKeyState* pKey);
```

**Include Files**

```
ippcp.h
```

**Domain Dependencies**

Headers: **ippcore.h**
**Parameters**

\[
\begin{align*}
\text{pBufferSize} & \quad \text{Pointer to the size of a temporary buffer.} \\
\text{pKey} & \quad \text{Pointer to the RSA key context.}
\end{align*}
\]

**Description**

These functions get the size of a temporary buffer for future use in public- or private-key RSA operations, respectively.

**Return Values**

\[
\begin{align*}
\text{ippStsNoErr} & \quad \text{Indicates no error. Any other value indicates an error or warning.} \\
\text{ippStsNullPtrErr} & \quad \text{Indicates an error condition if any of the specified pointers is NULL.} \\
\text{ippStsContextMatchErr} & \quad \text{Indicates an error condition if any of the context parameters does not match the operation.} \\
\text{ippStsIncompleteContextErr} & \quad \text{For \texttt{RSA\_GetBufferSizePublicKey}, indicates an error condition if the public key is not set up.} \\
& \quad \text{For \texttt{RSA\_GetBufferSizePrivateKeyType1}, indicates an error condition if the type 1 private key is not set up.}
\end{align*}
\]

**NOTE**

You can set up the public key or type 1 private key in a call to \texttt{RSA\_SetPublicKey} or \texttt{RSA\_SetPrivateKeyType1}, respectively. For the \texttt{RSA\_GetBufferSizePrivateKeyType2} function, it suffices to initialize the context for the key in a call to \texttt{RSA\_InitPrivateKeyType2}.

**See Also**

\texttt{RSA\_SetPublicKey, RSA\_SetPrivateKeyType1, RSA\_SetPrivateKeyType2}  
\texttt{RSA\_InitPublicKey, RSA\_InitPrivateKeyType1, RSA\_InitPrivateKeyType2}

**RSA\_GenerateKeys**

Generates key components for the desired RSA cryptographic system.

**Syntax**

\[
\text{IppStatus ippsRSA\_GenerateKeys(const IppsBigNumState* pSrcPublicExp, IppsBigNumState* pModulus, IppsBigNumState* pPublicExp, IppsBigNumState* pPrivateExp, IppsRSAPrivateKeyState* pPrivateKeyType2, Ipp8u* pScratchBuffer, int nTrials, IppsPrimeState* pPrimeGen, IppBitSupplier rndFunc, void* pRndParam);}
\]

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h  
Libraries: ippcore.lib
Parameters

- **pSrcPublicExp**: Pointer to the IppsBigNumState context of the initial value for searching an RSA public exponent.
- **pModulus**: Pointer to the generated RSA modulus.
- **pPublicExp**: Pointer to the generated RSA public exponent.
- **pPrivateExp**: Pointer to the generated RSA private exponent.
- **pPrivateKeyType2**: Pointer to the generated RSA private key type 2.
- **pScratchBuffer**: Pointer to the temporary buffer of size not less than returned by the RSA_GetBufferSizePrivateKey function.
- **nTrials**: Security parameter specified for the Miller-Rabin test for probable primality.
- **pPrimeGen**: Pointer to the prime number generator.
- **rndFunc**: Pseudorandom number generator.
- **pRndParam**: Pointer to the context of the pseudorandom number generator.

Description

This function generates public and private keys of the desired RSA cryptographic system. This function sequentially performs the following computations:

1. Generates random probable prime numbers \( p \) and \( q \) using the specified pseudorandom number generator \( \text{rndFunc} \).
2. Computes the RSA composite modulus \( n = (p \times q) \).
3. Based on the generated \( p \) and \( q \) factors, computes all the other CRT-related RSA components: \( dP = d \mod (p-1) \), \( dQ = p \mod (q-1) \) and \( qInv = 1/q \mod p \).

To generate RSA keys using the RSA_GenerateKeys function, call it in the following sequence of steps:

1. Establish the pseudorandom number generator and prime number generator.
2. Define the RSA private key type 2 in successive calls to the RSA_GetSizePrivateKeyType2 and RSA_InitPrivateKeyType2 functions with desired values of \( \text{factorPBitSize} \) and \( \text{factorQBitSize} \) parameters.
3. Allocate a temporary buffer of a suitable size.
4. Set up the initial value of the public exponent \( pSrcPublicExp \).
5. Call RSA_GenerateKeys.
   - If RSA_GenerateKeys returns IppNoErr, the key pair is generated.
   - If RSA_GenerateKeys returns ippStsInsufficientEntropy, repeat step 5.

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.
Indicates an error condition if the bit length of any key component specified by \( pModulus \), \( pPublicExp \) or \( pPrivateExp \) is not sufficient to hold the value or the prime number generator, specified by \( pPrimeGen \), is not sufficient to generate suitable values.

Indicates an error condition if the initial value for searching the public exponent, specified by \( pSrcPublicExp \), is not positive.

Indicates an error condition in cases not explicitly mentioned above.

Indicates a warning condition if the prime number generation fails due to a poor choice of entropy.

See Also
RSA_InitPublicKey, RSA_InitPrivateKeyType1, RSA_InitPrivateKeyType2
RSAValidateKeys
Pseudorandom Number Generation Functions

RSA_ValidateKeys
Validates key components of the RSA cryptographic system.

Syntax

\[
\text{IppStatus ippsRSA_ValidateKeys(int* pResult, const IppsRSAPublicKeyState* pPublicKey, const IppsRSAPrivateKeyState* pPrivateKeyType2, const IppsRSAPrivateKeyState* pPrivateKeyType1, Ipp8u* pScratchBuffer, int nTrials, IppsPrimeState* pPrimeGen, IppBitSupplier rndFunc, void* pRndParam);}\]

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pResult</td>
<td>Pointer to the result of validation.</td>
</tr>
<tr>
<td>pPublicKey</td>
<td>Pointer to the RSA public key.</td>
</tr>
<tr>
<td>pPrivateKeyType2</td>
<td>Pointer to the RSA private key type 2.</td>
</tr>
<tr>
<td>pPrivateKeyType1</td>
<td>Pointer to the RSA private key type 1. This parameter is optional and can have the value of NULL.</td>
</tr>
<tr>
<td>pScratchBuffer</td>
<td>Pointer to the temporary buffer of size not less than returned by the ( \text{RSA_GetBufferSizePrivateKey} ) function.</td>
</tr>
<tr>
<td>nTrials</td>
<td>Security parameter specified for the Miller-Rabin test for probable primality.</td>
</tr>
<tr>
<td>pPrimeGen</td>
<td>Pointer to the prime number generator.</td>
</tr>
<tr>
<td>rndFunc</td>
<td>Pseudorandom number generator.</td>
</tr>
<tr>
<td>pRndParam</td>
<td>Pointer to the context of the pseudorandom number generator.</td>
</tr>
</tbody>
</table>
Description
The function validates key components of the RSA cryptographic system and stores the result of the validation procedure in *pResult.

The meanings of values of *pResult are as follows:

- **IS_VALID_KEY**: The RSA key pair is valid.
- **IS_INVALID_KEY**: The RSA key is not valid.

The key pair is valid under the following conditions:

- The \( p \) and \( q \) factors are prime.
- The type 2 private key meets these conditions:
  - \( e*dP = 1 \pmod{p-1} \) and \( e*dQ = 1 \pmod{q-1} \)
  - \( q*qInv = 1 \pmod{p} \)
- If the pPrivateKeyType1 parameter is not NULL, the type 1 private key meets the condition \( e*d = 1 \pmod{(p-1)(q-1)} \).

Validation of the public and type 1 private key pair requires type 2 private key.

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.
- **ippStsSizeErr**: Indicates an error condition if the prime number generator, specified by pPrimeGen, is not sufficient to generate suitable values.
- **ippStsIncompleteContextErr**: Indicates an error condition if the public or private key is not set up.
- **ippStsBadArgErr**: Indicates an error condition if any of the RSA keys *pPublicKey, *pPrivateKeyType2, or, optional, *pPrivateKeyType1 is not properly set up or generated.

See Also
RSA_GenerateKeys

RSA Primitives
The functions described in this section refer to RSA primitives.

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 **RSAGetSizePublicKey** to get the size required to configure IppsRSSAPublicKeyState context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the **RSA_InitPublicKey** function to initialize the context.
3. Call **RSA_SetPublicKey** to set up RSA public key \((n, e)\).
4. Call the **RSA_GetBufferSizePublicKey** function to get the size of a temporary buffer.
5. Invoke the **RSA_Encrypt** function with the established RSA public key to encode the plaintext into the respective ciphertext.
6. Clean up secret data stored in the context.
7. Free the memory allocated for the IppsRSAPublicKeyState 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 GetSizePrivateKeyType1 or RSAGetSizePrivateKeyType2 to get the size required to configure IppsRSAPrivateKeyState context.
2. Ensure that the required memory space is properly allocated. With the allocated memory, call the InitPrivateKeyType1 or RSA_InitPrivateKeyType2 function to initialize the context.
3. Call the RSA_GetBufferSizePrivateKey function to get the size of a temporary buffer.
4. Establish the RSA private key by means of either the RSA_GenerateKeys function or by the key setup function RSA_SetPrivateKeyType1 or RSA_SetPrivateKeyType2. The RSA_GenerateKeys function can generate both type 1 and type 2 private keys, while the choice of the key setup function depends on the representation of the private key you are using.
5. Invoke the RSA_Decrypt function with the established RSA public key to decode the ciphertext into the respective plaintext.
6. Clean up secret data stored in the context.
7. Free the memory allocated for the IppsRSAPrivateKeyState context by calling the operating system memory free service function.

See Also
Data Security Considerations

RSA_Encrypt
Performs the RSA encryption operation.

Syntax
IppStatus ippsRSA_Encrypt(const IppsBigNumState* pPtxt, IppsBigNumState* pCtxt, const IppsRSAPublicKeyState* pKey, Ipp8u* pScratchBuffer);

Include Files
ippetcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pPtxt Pointer to the IppsBigNumState context of the plaintext.
pCtxt Pointer to the IppsBigNumState context of the ciphertext.
pKey Pointer to the IppsRSAPublicKeyState context.
pScratchBuffer Pointer to the temporary buffer of size not less than returned by the RSA_GetBufferSizePublicKey function.

Description
The function performs the RSA encryption operation, that is, the RSA operation on a public key.

Return Values
ippStsNoErr Indicates no error. Any other value indicates an error or warning.
Indicates an error condition if any of the specified pointers is NULL.

Indicates an error condition if the context parameter does not match the operation.

Indicates an error condition if the public key is not set up.

NOTE
You can set up the public key in a call to RSA_SetPublicKey.

Indicates an error condition if the RSA context has not been properly set up for the operation.

Indicates an error condition if the big number specified by \( pPtxt \) is not positive or greater than the RSA modulus.

Indicates an error condition if the big number specified by \( pCtxt \) is not sufficient to hold the result.

See Also
RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2
RSA_Decrypt
Functions for Building RSA System

RSA_Decrypt
Performs the RSA decryption operation.

Syntax
IppStatus ippsRSA_Decrypt(const IppsBigNumState* \( pCtxt \), IppsBigNumState* \( pPtxt \), const IppsRSAPrivateKeyState* \( pKey \), Ipp8u* \( pScratchBuffer \));

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
\( pCtxt \) Pointer to the IppsBigNumState context of the ciphertext.
\( pPtxt \) Pointer to the IppsBigNumState context of the plaintext.
\( pKey \) Pointer to the IppsRSAPrivateKeyState context.
\( pScratchBuffer \) Pointer to the scratch buffer of size not less than returned by the RSA_GetBufferSizePrivateKey function.

Description
The function performs the RSA encryption operation, that is, the RSA operation on a private key.
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 private key is not set up.

NOTE
While you can set up the type 1 private key in a call to RSA_SetPrivateKeyType1, you can set up the type 2 private key in a call to either RSA_SetPrivateKeyType2 or RSA_GenerateKeys.

ippStsOutOfRangeErr Indicates an error condition if the big number specified by pCtxt is not positive or greater than the RSA modulus.

ippStsSizeErr Indicates an error condition if the big number specified by pPtxt is not sufficient to hold the result.

See Also
RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2
RSA_GenerateKeys
RSA_Encrypt
Functions for Building RSA System

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

```c
// P prime factor
BigNumber P("0xEECFAE81B1B93C908810B10A1B5600199EB9F44AEF4F4DA493B81A9E3D84F632"
            "124EF02365D1E3B7E28FAP7EA040A2DSB252176459D1F397541BA2A58FB6599");
// Q prime factor
BigNumber Q("0xC97FB1F027F453F6341233EAAAD1D9353F6C42D0866B1D05A0F2035028B9D86"
            "9840B41666B2E92E40DA3B4324045CFCE3352524D0416A5A441E700AF461503");
// P's CRT exponent
BigNumber dP("0x54494CA63EBA0337E424023FC69A5AEB07DDDC0183A40AC9B54B051F2B13E"
            "D9490975E87414FF59C1F7692E9A2E202B38FC910474174AC93C1F67C981");
// Q's CRT exponent
BigNumber dQ("0x471E0290FF0AF0750351B7F878864CA961ADBD3A8A7E9991C50556A94C3146A7"
            "F9803F86F8AE342E931FD8AE47A2202D1B99A945849807FE39F9245A989636A3D");
// CRT coefficient
BigNumber invQ("0xB06C4FDBB6301198D265BDAE9423B380F271F7343538509307FCD39E2119F"
            "C98632154F5883B167A967BF402B49E9E20F9656E698EA3666EDEB25798309F7");
// rsa modulus N = P*Q
BigNumber N("0xBBF82F090682CE9C2338AC2B29A871F7368D07EED41043A440D66E70454F51F"
            "B8DBAAAF03C2AB61EA48CEEB6FCD4876ED52D6D0E1EC4619719D8A5BB8807E"
            "AFB8803ADFCE737723EE684B7D93A2584EE6A49D0D0953748834B245459854E"
            "E0AAB12D761A51F527A9A41F6C1687FE2537298CA2A8F5946F85F091DBDCB");
// private exponent
```
BigNumber    D("0xA5DAFC5341FAF289C4B988DB30C1CF83F31251E0668B42784813801579641B2"
"9410B3C7998D6BC465745E5C392669D6870DA2082939E37FDCB82EC93EDAC9"
"7654E25C953A6D656AF68C56C092CD38DC3BEF5D20A93"
"9926ED4F74A13EDDFBE1A1CECC4894AF9428C2B7B8883FE4463A4BC85B1CB3C1");

// public exponent
BigNumber    E("0x11");

int RSA_sample(void)
{
    int keyCtxSize;

    // (bit) size of key components
    int bitsN = N.BitSize();
    int bitsE = E.BitSize();
    int bitsP = P.BitSize();
    int bitsQ = Q.BitSize();

    // define and setup public key
    ippsRSA_GetSizePublicKey(bitsN, bitsE, &keyCtxSize);
    IppsRSAPublicKeyState* pPub = (IppsRSAPublicKeyState*)( new Ipp8u [keyCtxSize] );
    ippsRSA_InitPublicKey(bitsN, bitsE, pPub, keyCtxSize);
    ippsRSA_SetPublicKey(N, E, pPub);

    // define and setup (type2) private key
    ippsRSA_GetSizePrivateKeyType2(bitsP, bitsQ, &keyCtxSize);
    IppsRSAPrivateKeyState* pPrv = (IppsRSAPrivateKeyState*)( new Ipp8u [keyCtxSize] );
    ippsRSA_InitPrivateKeyType2(bitsP, bitsQ, pPrv, keyCtxSize);
    ippsRSA_SetPrivateKeyType2(P, Q, dP, dQ, invQ, pPrv);

    // allocate scratch buffer
    int buffSizePublic;
    ippsRSA_GetBufferSizePublicKey(&buffSizePublic, pPub);
    int buffSizePrivate;
    ippsRSA_GetBufferSizePrivateKey(&buffSizePrivate, pPrv);
    int buffSize = max(buffSizePublic, buffSizePrivate);
    Ipp8u* scratchBuffer = NULL;
    scratchBuffer = new Ipp8u [buffSize];

    // error flag
    int error = 0;

do {
    // validate keys
    //
    // random generator
    IppsPRNGState* pRand = newPRNG();
    // prime generator
    IppsPrimeState* pPrimeG = newPrimeGen(P.BitSize());

    int validateRes = IPP_IS_INVALID;
    ippsRSA.ValidateKeys( &validateRes,
                            pPub, pPrv, NULL, scratchBuffer,
                            10, pPrimeG, ippsPRNGen, pRand);

    // delete generators
    deletePrimeGen(pPrimeG);
    deletePRNG(pRand);

    //
if(IPP_IS_VALID!=validateRes) {
    cout <<"validation fail" << endl;
    error = 1;
    break;
}

// known plain- and ciper-texts
BigNumber kat_PT("0x00EB7A19ACE93006350E329504B45E2CA82310B26DCD87D5C68F1EEA8F55267"
    "C31B2E8BB4251F84D7E0B2C04626F5AFF93EDCFB25C9C2B3FF8AE10E839A2DD"
    "4CDCFE4FF4772B84A1B7C1362BAAD29AB48D2869D5024121435811591BE392F9"
    "82FB3E87D095AEB40448D972F3AC14F7BC275195281CE32D2F1B76D4D353E2D");
BigNumber kat_CT("0x1253E04DC0A5397BB44A7AB87E9BF2A039A33D1E996FC82A94CCD30074C95DF7"
    "63722017069E5268DA5D1C0B4F872CF653C11DF82314A67968DFEAEE28DEF04BB"
    "6D4B1C31D65A188705782B06EB96A0242CA2F490FE92FEF5C9C1405BB48"
    "DA9536AD8700C84FC9130ADEA74E550D51A74DD58D50DE96838D603E9555");

// encrypt  message
BigNumber ct(0, N.DwordSize());
ippsRSA_Encrypt(kat_PT, ct, pPub, scratchBuffer);
if(ct!=kat_CT) {
    cout <<"encryption fail" << endl;
    error = 1;
    break;
}

// decrypt message
BigNumber rt(0, N.DwordSize());
ippsRSA_Decrypt(kat_CT, rt, pPrv, scratchBuffer);
if(rt!=kat_PT) {
    cout <<"decryption fail" << endl;
    error = 1;
    break;
}
} while(0);

delete [] scratchBuffer;
delete [] (Ipp8u*) pPub;
// remove sensitive data before release
ippsRSA_InitPrivateKeyType2(bitsP, bitsQ, pPrv, keyCtxSize);
delete [] (Ipp8u*) pPrv;
return error==0;

RSA Encryption Schemes

RSA-OAEP Scheme Functions
This subsection describes functions implementing RSA-OAEP encryption scheme, specified in [PKCS 1.2.1].
RSAEncrypt_OAEP  
Carries out the RSA-OAEP encryption scheme.

Syntax

```c
IppStatus ippsRSAEncrypt_OAEP(const Ipp8u* pSrc, int srcLen, const Ipp8u* pLabel, int labLen, const Ipp8u* pSeed, Ipp8u* pDst, const IppsRSAPublicKeyState* pKey, IppHashAlgId hashAlg, Ipp8u* pBuffer);
```

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- `pSrc`: Pointer to the octet message to be encrypted.
- `srcLen`: Length of the message to be encrypted.
- `pLabel`: Pointer to the optional label to be associated with the message.
- `labLen`: Length of the optional label.
- `pSeed`: Pointer to the random octet string of length `hashLen`, where `hashLen` is the length (in octets) of the hash function output.
- `pDst`: Pointer to the output octet ciphertext string.
- `pKey`: Pointer to the properly initialized `IppsRSAPublicKeyState` context.
- `hashAlg`: ID of the hash algorithm used. For details, see table Supported Hash Algorithms.
- `pBuffer`: Pointer to a temporary buffer of size not less than returned by the `RSA_GetBufferSizePublicKey` function.

Description

The function carries out the RSA-OAEP encryption scheme, defined in [PKCS 1.2.1]. The length of the encrypted message is equal to the size of the RSA modulus `n`.

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 public key is not set up.
- `ippStsLengthErr`: Indicates an error condition if the any input/output length parameters are inconsistent with one another.

**NOTE**
You can set up the public key in a call to `RSA_SetPublicKey`.


if the hashAlg parameter does not match any value of IppHashAlgId listed in table Supported Hash Algorithms.

See Also
RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2
RSADecrypt_OAEP

RSADecrypt_OAEP
Carries out the RSA-OAEP decryption scheme.

Syntax
IppStatus ippsRSADecrypt_OAEP(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen,
Ipp8u* pDst, int* pDstLen, const IppsRSA_PrivateKeyState* pKey, IppHashAlgId hashAlg,
Ipp8u* pBuffer);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSrc</td>
<td>Pointer to the octet ciphertext to be decrypted.</td>
</tr>
<tr>
<td>pLabel</td>
<td>Pointer to the optional label to be associated with the message.</td>
</tr>
<tr>
<td>labLen</td>
<td>Length of the optional label.</td>
</tr>
<tr>
<td>pDst</td>
<td>Pointer to the output octet plaintext message.</td>
</tr>
<tr>
<td>pDstLen</td>
<td>Pointer to the length of the decrypted message.</td>
</tr>
<tr>
<td>pKey</td>
<td>Pointer to the properly initialized IppsRSA_PrivateKeyState context.</td>
</tr>
<tr>
<td>hashAlg</td>
<td>ID of the hash algorithm used. For details, see table Supported Hash Algorithms.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>Pointer to a temporary buffer of size not less than returned by the RSA_GetBufferSizePrivateKey function.</td>
</tr>
</tbody>
</table>

Description
The function carries out the RSA-OAEP decryption scheme defined in [PKCS 1.2.1]. The *pDstLen parameter returns the length of the decrypted message.

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippStsNoErr</td>
<td>Indicates no error. Any other value indicates an error or warning.</td>
</tr>
<tr>
<td>ippStsNullPtrErr</td>
<td>Indicates an error condition if any of the specified pointers is NULL.</td>
</tr>
<tr>
<td>ippStsContextMatchErr</td>
<td>Indicates an error condition if the context parameter does not match the operation.</td>
</tr>
<tr>
<td>ippStsIncompleteContextErr</td>
<td>Indicates an error condition if the private key is not set up.</td>
</tr>
</tbody>
</table>
NOTE
While you can set up the type 1 private key in a call to RSA_SetPrivateKeyType1, you can set up the type 2 private key in a call to either RSA_SetPrivateKeyType2 or RSA_GenerateKeys.

See Also
RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2
RSAEncrypt_OAEP
RSA_GenerateKeys

PKCS V1.5 Encryption Scheme Functions
This subsection describes functions implementing encryption schemes defined in version 1.5 of the PKCS#1 standard ([PKCS 1.2.1]).

RSAEncrypt_PKCSv15
Performs RSA-OAEP encryption using the RSA-OAEP scheme as defined in the v1.5 version of the PKCS#1 standard.

Syntax
IppStatus ippsRSAEncrypt_PKCSv15 (const Ipp8u* pSrc, int srcLen, const Ipp8u* pRandPS, Ipp8u* pDst, const IppsRSAPublicKeyState* pKey, Ipp8u* pBuffer);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pSrc
Length (in bytes) of the message. The message can be empty, that is, srcLen==0.

srcLen

pRandPS
Pointer to the non-zero octet padding string. pRandPS can be NULL. In this case, the function applies the padding string of 0xFF bytes.

pDst
Pointer to the output message.

pKey
Pointer to the properly initialized IppsRSAPublicKeyState context.

pBuffer
Pointer to a buffer of size not less than returned by the RSA_GetBufferSizePublicKey function.
Description
The function performs encryption using the RSA-OAEP scheme according to the v1.5 version of the PKCS#1 standard, defined in [PKCS 1.2.1]. The length of the encrypted message is equal to size of the RSA modulus $n$.

If RSAEncrypt_PKCSv15 receives a non-zero $pRandPS$ pointer, the function assumes that the length of the padding string is at least $k-srcLen-3$ 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
- ippStsNoErr: Indicates no error. Any other value indicates an error or warning.
- ippStsNullPtrErr: Indicates an error condition if any of the specified pointers other than $pRandPS$ is NULL.
- ippStsContextMatchErr: Indicates an error condition if the RSA context parameter does not match the operation.
- ippStsIncompleteContextErr: Indicates an error condition if the public key is not set up.
- ippStsSizeErr: Indicates an error condition if any input/output length parameters are inconsistent with one another.

**NOTE**
You can set up the public key in a call to RSA_SetPublicKey.

See Also
- RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2
- RSADecrypt_PKCSv15

RSADecrypt_PKCSv15
Performs RSA-OAEP decryption using the RSA-OAEP scheme as defined in the v1.5 version of the PKCS#1 standard.

Syntax
IppStatus ippsRSADecrypt_PKCSv15 (const Ipp8u* pSrc, Ipp8u* pDst, int* pDstLen, const IppsRSAPrivateKeyState* pKey, Ipp8u* pBuffer);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib
Parameters

- **pSrc**: Pointer to the input octet message to be decrypted.
- **pDst**: Pointer to the output message.
- **pDstLen**: Pointer to the length (in bytes) of the decrypted message.
- **pKey**: Pointer to the properly initialized `IppsRSAPrivateKeyState` context.
- **pBuffer**: Pointer to a temporary buffer of size not less than returned by the `RSA_GetBufferSizePrivateKey` function.

Description

The function performs decryption using the RSA-OAEP scheme according to the v1.5 version of the PKCS#1 standard, defined in [PKCS 1.2.1]. The `*pDstLen` parameter returns the length of the decrypted message.

**NOTE**

If an empty message is encrypted by the `RSAEncrypt_PKCSv15` function, `RSADecrypt_PKCSv15` returns an empty string, that is, `*pDstLen==0`.

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 RSA context parameter does not match the operation.
- **ippStsIncompleteContextErr**: Indicates an error condition if the private key is not set up.
- **ippStsSizeErr**: Indicates an error condition if any input/output length parameters are inconsistent with one another.

**NOTE**

While you can set up the type 1 private key in a call to `RSA_SetPrivateKeyType1`, you can set up the type 2 private key in a call to either `RSA_SetPrivateKeyType2` or `RSA_GenerateKeys`.

See Also

- `RSA_SetPublicKey`, `RSA_SetPrivateKeyType1`, `RSA_SetPrivateKeyType2`
- `RSA_GenerateKeys`
- `RSAEncrypt_PKCSv15`

RSA Signature Schemes

**RSA-SSA Scheme Functions**

This subsection describes functions implementing RSASSA-PSS_5 signature scheme with appendix [PKCS 1.2.1].
To invoke **RSASign_PSS** or **RSAVerify_PSS** primitive, supply the **IppsRSAPrivateKeyState** and/or **IppsRSAPublicKeyState** context initialized by a suitable function (see **RSA_InitPublicKey**, **RSA_InitPrivateKeyType1**, or **RSA_InitPrivateKeyType2** for details).

**RSASign_PSS**
Carries out the **RSASSA-PSS** signature generation scheme.

**Syntax**

```c
IppStatus ippsRSASign_PSS(const Ipp8u* pMsg, int msgLen, const Ipp8u* pSalt, int saltLen, Ipp8u* pSign, const IppsRSAPrivateKeyState* pPrivateKey, const IppsRSAPublicKeyState* pPublicKeyOpt, IppHashAlgId hashAlg, Ipp8u* pBuffer);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- **pMsg**: Pointer to the octet message to be signed.
- **msgLen**: Length of the input *pMsg message in octets.
- **pSalt**: Pointer to the random octet salt string.
- **saltLen**: Length of the salt string in octets.
- **pSign**: Pointer to the output octet signature.
- **pPrivateKey**: Pointer to the properly initialized **IppsRSAPrivateKeyState** context.
- **pPublicKeyOpt**: Pointer to the properly initialized optional **IppsRSAPublicKeyState** context.
- **hashAlg**: Identifier of the hash algorithm. For details, see table Supported Hash Algorithms.
- **pBuffer**: Pointer to a temporary buffer of size not less than returned by each of the functions **RSA_GetBufferSizePrivateKey** and **RSA_GetBufferSizePublicKeyKey**.

**Description**

The function generates the message signature according to the **RSASSA-PSS** scheme defined in [PKCS 1.2.1] using the hash algorithm defined by the **hashAlg** parameter.

If you are using an RSA private key type 2 to generate the signature, you can use the optional *pPublicKeyOpt parameter to mitigate Fault Attack. If you are using an RSA private key type 1 or sure that Fault Attack is not applicable, pPublicKeyOpt can be NULL. Passing the NULL value to the pPublicKeyOpt parameter saves computation time.

**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.
Indicates an error condition if any of the context parameters does not match the operation.

Indicates an error condition if the public or private key is not set up.

Indicates an error condition if the value of saltLen is negative or any input/output length parameters are inconsistent with one another together (see [PKCS 1.2.1] for details).

Indicates an error condition if the hashAlg parameter does not match any value of IppHashAlgId listed in table Supported Hash Algorithms.

See Also
RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2
RSA_GenerateKeys
RSAVerify_PSS

RSAVerify_PSS
Carries out the RSA-SSA signature verification scheme.

Syntax
IppStatus ippsRSAVerify_PSS(const Ipp8u* pMsg, int msgLen, const Ipp8u* pSign, int* pIsSignValid, const IppsRSAPublicKeyState* pKey, IppHashAlgId hashAlg, Ipp8u* pBuffer);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pMsg Pointer to the octet message that has been signed.
msgLen Length in octets of the pMsg message.
pSign Pointer to the octet signature string to be verified.
pIsSignValid Pointer to the verification result.
pKey Pointer to the properly initialized IppsRSAPublicKeyState context.
hashAlg Identifier of the hash algorithm. For details, see table Supported Hash Algorithms.
pBuffer Pointer to the scratch buffer of size not less than returned by the RSA_GetBufferSizePublicKey function.

Description
The function carries out the RSASSA-PSS signature verification scheme defined in [PKCS 1.2.1]. RSAVerify_PSS verifies the signature generated by the RSASign_PSS function called with the same hashAlg parameter.
### 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 public key is not set up.
- **ippStsNotSupportedModeErr**: Indicates an error condition if the `hashAlg` parameter does not match any value of `IppHashAlgId` listed in table [Supported Hash Algorithms](#).

### See Also

- [RSA_SetPublicKey](#)
- [RSA_SetPrivateKeyType1](#)
- [RSA_SetPrivateKeyType2](#)

### PKCS V1.5 Signature Scheme Functions

This subsection describes functions implementing the RSASSA-PKCS1-v1_5 signature scheme with appendix [PKCS 1.2.1](#).

#### RSASign_PKCS1v15

*Carries out the RSA-SSA signature generation scheme of PKCS#1 v1.5.*

#### Syntax

```c
IppStatus ippsRSASign_PKCS1v15(const Ipp8u* pMsg, int msgLen, Ipp8u* pSign, const IppsRSAPrivateKeyState* pPrivateKey, const IppsRSAPublicKeyState* pPublicKeyOpt, IppHashAlgId hashAlg, Ipp8u* pBuffer);
```

#### Include Files

```c
ippcp.h
```

#### Domain Dependencies

- **Headers**: ippcore.h
- **Libraries**: ippcore.lib

#### Parameters

- **pMsg**: Pointer to the message to be signed.
- **msgLen**: Length of the message `pMsg` in octets.
- **pSign**: Pointer to the output octet signature.
- **pPrivateKey**: Pointer to the properly initialized `IppsRSAPrivateKeyState` context.
- **pPublicKeyOpt**: Pointer to the properly initialized optional `IppsRSAPublicKeyState` context.
- **hashAlg**: Identifier of the hash algorithm used. For details, see table [Supported Hash Algorithms](#).
**pBuffer**

Pointer to a temporary buffer of size not less than returned by each of the functions `RSA_GetBufferSizePrivateKey` and `RSA_GetBufferSizePublicKeyKey`.

**Description**

The function computes the message digest specified by the `hashAlg` parameter and generates the signature according to the RSASSA-PKCS1-v1_5 scheme defined in [PKCS 1.2.1].

If you are using an RSA private key type 2 to generate the signature, you can use the optional `pPublicKeyOpt` parameter to mitigate Fault Attack. If you are using an RSA private key type 1 or sure that Fault Attack is not applicable, `pPublicKeyOpt` can be NULL. Passing the NULL value to the `pPublicKeyOpt` parameter saves computation time.

**Important**

The length of the signature being generated equals the length of the RSA modulus, supplied with the `IppsRSAPrivateKeyState` context. Make sure that `pSign` points to a buffer of a sufficient length.

**Return Values**

- **ippStsNoErr**: Indicates no error. Any other value indicates an error or warning.
- **ippStsNullPtrErr**: Indicates an error condition if any of the specified pointers is NULL.
- **ippStsContextMatchErr**: Indicates an error condition if any of the context parameters does not match the operation.
- **ippStsIncompleteContextErr**: Indicates an error condition if the public or private key is not set up.

**NOTE**

While you can set up the public key or type 1 private key in a call to `RSA_SetPublicKey` or `RSA_SetPrivateKeyType1`, respectively, you can set up the type 2 private key in a call to either `RSA_SetPrivateKeyType2` or `RSA_GenerateKeys`.

- **ippStsLengthErr**: Indicates an error condition if any input/output length parameters are inconsistent with one another.
- **ippStsSizeErr**: Indicates an error condition if the length of the RSA modulus is too small (see details in [PKCS 1.2.1]).
- **ippStsNotSupportedModeErr**: Indicates an error condition if the `hashAlg` parameter does not match any value of `IppHashAlgId` listed in table Supported Hash Algorithms.

**See Also**

- `RSA_SetPublicKey`, `RSA_SetPrivateKeyType1`, `RSA_SetPrivateKeyType2`
- `RSA_GenerateKeys`
- `RSAVerify_PKCS1v15`

**RSAVerify_PKCS1v15**

Carries out the RSA-SSA signature verification scheme of PKCS#1 v1.5.
Syntax

IppStatus ippsRSAVerify_PKCS1v15(const Ipp8u* pMsg, int msgLen, const Ipp8u* pSign, int* pIsSignValid, const IppsRSAPublicKeyState* pKey, IppHashAlgId hashAlg, Ipp8u* pBuffer);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pMsg          Pointer to the message that has been signed.
msgLen        Length of the message *pMsg in octets.
pSign         Pointer to the signature string to be verified.
pIsSignValid  Pointer to the verification result.
pKey          Pointer to the properly initialized IppsRSAPublicKeyState context.
hashAlg       Identifier of the hash algorithm. For details, see table Supported Hash Algorithms.
pBuffer       Pointer to a temporary buffer of size not less than returned by the RSA_GetBufferSizePublicKey function.

Description

The function verifies the signature generated by the RSASign_PKCS1v15 function that uses the same hashAlg parameter against the input message, as defined [PKCS 1.2.1].

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 public key is not set up.

NOTE
You can set up the public key in a call to RSA_SetPublicKey.

ippStsLengthErr Indicates an error condition if any input/output length parameters are inconsistent with one another.
ippStsNotSupportedModeErr Indicates an error condition if the hashAlg parameter does not match any value of IppHashAlgId listed in table Supported Hash Algorithms.
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].

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 function.
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. Clean up secret data stored in the context.
6. 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.

See Also
Data Security Considerations

DLPGetSize

*Gets the size of the IppsDLPState context.*

**Syntax**

IppStatus ippsDLPGetSize(int peBits, int reBits, int *pSize);

**Include Files**

ippcp.h

**Domain Dependencies**

*Headers:* ippcore.h

*Libraries:* ippcore.lib

**Parameters**

*peBits*  
Bitsize of the GF($p$) element (that is, the length of the DL-based cryptosystem in bits)

*reBits*  
Bitsize of the multiplicative subgroup GF($r$).

*pSize*  
Pointer to the IppsDLPState context size in bytes.
Description
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

- 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 \( peBits \leq reBits \).

DLPInit

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

Syntax

IppStatus ippsDLPInit(int peBits, int reBits, IppsDLPState* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- peBits: Bitsize of the GF(p) element (that is, the length of the DL-based cryptosystem in bits)
- reBits: Bitsize of the multiplicative subgroup GF(r).
- pCtx: Pointer to the IppsDLPState context being initialized.

Description

The function initializes the memory pointed by pCtx as the IppsDLPState 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.
- ippStsSizeErr: Indicates an error condition if \( peBits \leq reBits \).

See Also

Data Security Considerations
**DLPPack, DLPUnpack**

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

**Syntax**

```c
IppStatus ippsDLPPack (const IppsDLPState* pCtx, Ipp8u* pBuffer);
IppStatus ippsDLPUnpack (const Ipp8u* pBuffer, IppsDLPState* pCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- `pCtx`  
  Pointer to the IppsDLPState context.

- `pBuffer`  
  Pointer to the user-defined buffer.

**Description**

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**

- `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.

**DLPSet**

*Sets up domain parameters of the DL-based cryptosystem over GF(p).*

**Syntax**

```c
IppStatus ippsDLPSet(const IppsBigNumState* pP, const IppsBigNumState* pQ, const IppsBigNumState* pG, IppsDLPState* pCtx);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib
Parameters

- \( pP \): Pointer to the characteristic \( p \) of the prime finite field \( GF(p) \).
- \( pQ \): Pointer to the characteristic \( q \) of the multiplicative subgroup \( GF(q) \).
- \( pG \): Pointer to the generator \( G \) of the multiplicative subgroup \( GF(r) \).
- \( pCtx \): Pointer to the cryptosystem context.

Description

The function sets up DL-based cryptosystem domain parameters into the cryptosystem 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.
- ippStsContextMatchErr: Indicates an error condition if the context parameter does not match the operation.
- ippStsRangeErr: Indicates an error condition if any of the Big Numbers specified by \( pP \), \( pQ \), and \( pG \) is too big to be stored in the \( IppsDLPState \) context.

DLPGet

Retrieves domain parameters of the DL-based cryptosystem over \( GF(p) \).

Syntax

\[
\text{IppStatus ippsDLPGet(IppsBigNumState* } pP, \ \text{IppsBigNumState* } pQ, \ \text{IppsBigNumState* } pG, \ \text{IppsDLPState* } pCtx); 
\]

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- \( pP \): Pointer to the characteristic \( p \) of the prime finite field \( GF(p) \).
- \( pQ \): Pointer to the characteristic \( q \) of the multiplicative subgroup \( GF(q) \).
- \( pG \): Pointer to the generator \( G \) of the multiplicative subgroup \( GF(r) \).
- \( pCtx \): Pointer to the cryptosystem context.

Description

The function retrieves DL-based cryptosystem domain parameters into the cryptosystem 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.
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.
ippStsRangeErr     Indicates an error condition if any of the Big Numbers specified by \( p_F, p_R, \) and \( p_G \) 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, IppDLPKeyTag tag, IppsDLPState* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- \( pDP \)  
  Pointer to the domain parameter value to be set.
- \( tag \)  
  Tag specifying the desired domain parameter.
- \( pCtx \)  
  Pointer to the cryptosystem context.

Description

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

- ippStsNoErr  
  Indicates no error. Any other value indicates an error or warning.
Indicates an error condition if any of the specified pointers is NULL.

Indicates an error condition if the context parameter does not match the operation.

Indicates an error condition if the Big Number specified by pDP is too big to be stored in the IppsDLPState context.

Indicates an error condition if some of the function parameters are invalid:
  Big Number specified by pDP is negative
  Domain parameter specified by tag does not match the IppsDLPState context.

DLPGetDP
Retrieves a particular domain parameter of the DL-based cryptosystem over GF(p).

Syntax
IppStatus ippsDLPGetDP(IppsBigNumState* pDP, IppDLPKeyTag tag, const IppsDLPState* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pDP  Pointer to the output Big Number context.
tag  Tag specifying the domain parameter to be retrieved.
pCtx  Pointer to the cryptosystem context.

Description
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
ippStsNoErr  Indicates no error. Any other value indicates an error or warning.
Indicates an error condition if any of the specified pointers is NULL.
Indicates an error condition if the context parameter does not match the operation.
Indicates an error condition if the cryptosystem context has not been properly set up.
Indicates an error condition if the Big Number specified by \( p_{DP} \) is too small for the DL parameter.
Indicates an error condition if the domain parameter specified by the tag does not match the IppsDLPState context.

**DLPGenKeyPair**
Generates a private key and computes public keys of the DL-based cryptosystem over GF(\( p \)).

**Syntax**

\[
\text{IppStatus ippsDLPGenKeyPair(IppsBigNumState}\ast p_{Private}, \text{IppsBigNumState}\ast p_{Public}, \\
\text{IppsDLPState}\ast p_{Ctx}, \text{IppBitSupplier} \text{ \( rndFunc \), void}\ast p_{RndParam});
\]

**Include Files**
ippcp.h

**Domain Dependencies**
Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- \( p_{Private} \)  
  Pointer to the private key \( privKey \).
- \( p_{Public} \)  
  Pointer to the public key \( pubKey \).
- \( p_{Ctx} \)  
  Pointer to the cryptosystem context.
- \( rndFunc \)  
  Specified Random Generator.
- \( p_{RndParam} \)  
  Pointer to the Random Generator context.

**Description**
The function generates a private key \( privKey \) and computes a public key \( pubKey \) of the DL-based cryptosystem. The function employs specified \( \text{\texttt{rndFunc}} \) 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**

- \( \text{ippStsNoErr} \)  
  Indicates no error. Any other value indicates an error or warning.
- \( \text{ippStsNullPtrErr} \)  
  Indicates an error condition if any of the specified pointers is NULL.
- \( \text{ippStsContextMatchErr} \)  
  Indicates an error condition if the context parameter does not match the operation.
Indicates an error condition if the cryptosystem context has not been properly set up.

Indicates an error condition if any of the Big Numbers specified by \( p_{Private} \) and \( p_{Public} \) 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**

\[
\text{IppStatus ippsDLPPublicKey(const IppsBigNumState* } p_{Private}, \text{ IppsBigNumState* } p_{Public}, \text{ IppsDLPState* } p_{Ctx});
\]

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- \( p_{Private} \): Pointer to the input private key \( privKey \).
- \( p_{Public} \): Pointer to the output public key \( pubKey \).
- \( p_{Ctx} \): Pointer to the cryptosystem context.

**Description**

The function computes a public key \( pubKey \) of the DL-based cryptosystem.

**Return Values**

- \( \text{ippStsNoErr} \): Indicates no error. Any other value indicates an error or warning.
- \( \text{ippStsNullPtrErr} \): Indicates an error condition if any of the specified pointers is NULL.
- \( \text{ippStsContextMatchErr} \): Indicates an error condition if the context parameter does not match the operation.
- \( \text{ippStsIncompleteContextErr} \): Indicates an error condition if the cryptosystem context has not been properly set up.
- \( \text{ippStsInvalidPrivateKey} \): Indicates an error condition if the \( privKey \) has an illegal value.
- \( \text{ippStsRangeErr} \): Indicates an error condition if Big Number specified by \( p_{Public} \) is too small for the DL public key.

**DLPValidateKeyPair**

*Validates private and public keys of the DL-based cryptosystem over GF\((p)\).*
Syntax
IppStatus ippsDLPValidateKeyPair(const IppsBigNumState* pPrivate, const IppsBigNumState* pPublic, IppDLResult* pResult, IppsDLPState* pCtx);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

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
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≤ (P - 1)) is false.
ippDLInvalidKeyPair public != G^private (mod P).

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.

DLPSetKeyPair
Sets private and/or public keys of the DL-based cryptosystem over GF(p).

Syntax
IppStatus ippsDLPSetKeyPair(const IppsBigNumState* pPrivate, const IppsBigNumState* pPublic, IppsDLPState* pCtx);
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pPrivate
Pointer to the input private key *privKey*.
pPublic
Pointer to the output public key *pubKey*.
pCtx
Pointer to the cryptosystem context.

Description
The function stores the private key *privKey* and public key *pubKey* in the cryptosystem 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.
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.

DLPGenerateDSA
Generates domain parameters of the DL-based cryptosystem over GF(p) to use DSA.

Syntax
IppStatus ippsDLPGenerateDSA(const IppsBigNumState* pSeedIn, int nTrials, IppsDLPState* pCtx, IppsBigNumState* pSeedOut, int* pCounter, IppBitSupplier rndFunc, void* pRndParam);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pSeedIn
Pointer to the input *Seed*.
nTrials
Security parameter specified for the Miller-Rabin probable primality.
pCtx
Pointer to the cryptosystem context.
**Description**

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**

- **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.
- **ippStsSizeErr**: Indicates an error condition if: peBits < 512, peBits is not divided by 64, reBits != 160.
- **ippStsRangeErr**: Indicates an error condition if: bitsize of the input Seed value is less than 160, bitsize of the input Seed value is greater than peBits, not enough space to store the output Seed value (if requested).
- **ippStsBadArgErr**: Indicates an error condition if nTrials < 1.
- **ippStsInsuffucientEntropy**: 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**

```c
IppStatus ippsDLPValidateDSA(int nTrials, IppDLResult* pResult, IppsDLPState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- **nTrials**: Security parameter specified for the Miller-Rabin probable primality.
- **pResult**: Pointer to the validation result.
- **pCtx**: Pointer to the cryptosystem context.
**Description**

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:

- **ippDLValid**: Validation has passed successfully.
- **ippDLBaseIsEven**: $P$ is even.
- **ippDLOrderIsEven**: $R$ is even.
- **ippDLOrderIsEven**: $P \leq 2^{peBits-1}$ or $P \geq 2^{peBits}$.
- **ippDLOrderIsEven**: $R \leq 2^{reBits-1}$ or $R \geq 2^{reBits}$.
- **ippDLOrderIsEven**: $P$ is not a prime.
- **ippDLOrderIsEven**: $R$ is not a prime.
- **ippDLOrderIsEven**: $R$ is not divisible by $(P - 1)$.
- **ippDLOrderIsEven**: $(1 < G < (P - 1))$ is false or $G^R \not= 1 \pmod{P}$.

To ensure that both $p$ and $r$ are primes, the function applies $nTrials$-round Miller-Rabin primality test. Test data for primality test is provided by the specified `rndFunc` Random Generator.

**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.
- **ippStsBadArgErr**: Indicates an error condition if $nTrials < 1$.

**DLPSignDSA**

*Performs the DSA digital signature signing operation.*

**Syntax**

```c
IppStatus ippsDLPSignDSA(const IppsBigNumState* pMsg, const IppsBigNumState* pPrivate, IppsBigNumState* pSignR, IppsBigNumState* pSignS, IppsDLPState* pCtx);
```

**Include Files**

- ippcp.h

**Domain Dependencies**

- **Headers**: ippcore.h
- **Libraries**: ippcore.lib
Parameters

- `pMsg`: Pointer to the message representation `msgRep` to be signed.
- `pPrivate`: Pointer to the signer's private key `privKey`.
- `pSignR`: Pointer to the `r`-component of the signature.
- `pSignS`: Pointer to the `s`-component of the signature.
- `pCtx`: Pointer to the cryptosystem context.

Description

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

- `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.
- `ippStsMessageErr`: Indicates an error condition if the value of `msgRep` is greater than the multiplicative subgroup characteristic (`q`).
- `ippStsInvalidPrivateKey`: Indicates an error condition if an illegal value has been assigned to `privKey`.
- `ippStsRangeErr`: Indicates an error condition if any of the signature components has not enough space.

DLPVerifyDSA

*Verifies the input DSA digital signature.*

Syntax

```c
IppStatus ippsDLPVerifyDSA(const IppsBigNumState* pMsg, const IppsBigNumState* pSignR, const IppsBigNumState* pSignS, IppDLResult* pResult, IppsDLPState* pCtx);
```

Include Files

`ippcp.h`

Domain Dependencies

Headers: `ippcore.h`
Libraries: `ippcore.lib`

Parameters

- `pSignR`: Pointer to the signature `r`-component to be verified.
**Description**

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 5-9** 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**

- **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.
- **ippStsMessageErr** Indicates an error condition if the value of msgRep is greater than the multiplicative subgroup characteristic (q).

**Example of Using Discrete-logarithm Based Primitive Functions**

**Use of DLPSignDSA and DLPVerifyDSA**

```c
//
// known domain parameters
//
static const int M = 512; // DSA system bitsize
static const int L = 160; // DSA order bitsize

static BigNumber P("0x8DF2A49492276AA3D25759BB06869CBEAC0D83AFB8D0CF7" \\
"CBB8324F0D7882E5D0762FC5B7210EAF2C9ADAC32AB7AAC" \\
"49693DFBF83724C2EC0736EE31C80291");

static BigNumber Q("0xC773218C737EC8EE993B4F2DED30F48EDACE915F");

static BigNumber G("0x626D027839EA0A13413163A55B4CB500299D5522956CEFCB" \\
"38FF10F399C2C2E71C9DE5FAA2FBF58E5B79521925C9C" \\
"C42E9F6F464B0B88CC572AF53E6D78802");

//
// known DSA regular key pair
//```
static
BigNumber X("0x2070B3223DBA372FDE1C0FFC7B2E3B498B260614");

static
BigNumber Y("0x19131871D75B1612A819F29D70D1B0D7346F7AA77BB62A85 \
98FD6C5675DA9D212D3A36EF1672EF660B8C7C255CC0EC74 \
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);

    // remove actual keys from context and release resource
    ippsDLPInit(M, L, 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);

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pSeedIn**
  - Pointer to the input Seed.
  - Security parameter specified for the Miller-Rabin probable primality.

- **nTrials**
  - Pointer to the cryptosystem context.

- **pCtx**
  - Pointer to the output Seed value (if requested).

- **pSeedOut**
  - Pointer to the counter value (if requested).

- **pCounter**
  - Specified Random Generator.

- **rndFunc**
  - Pointer to the Random Generator context.

**Description**

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**

- **ippStsNoErr**
  - Indicates no error. Any other value indicates an error or warning.

- **ippStsNullFptrErr**
  - 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.

- **ippStsSizeErr**
  - Indicates an error condition if: $peBits < 512$ or $reBits < 160$, $peBits$ is not divided by 256.

- **ippStsRangeErr**
  - Indicates an error condition if: bitsize of the input Seed value is less than $reBits$, not enough space to store the output Seed value (if requested).
indicating an error condition if \( n\text{Trials} < 1 \).

\text{ippStsInsufficientEntropy}\hspace{1em}\text{Indicates a warning condition if prime generation fails due to a poor choice of the entropy.}

\textbf{DLPValidateDH}

Validates domain parameters of the DL-based cryptosystem over GF(\( p \)) to use the DH Agreement scheme.

\textbf{Syntax}

\begin{verbatim}
IppStatus ippsDLPValidateDH(int nTrials, IppDLResult* pResult, IppsDLPState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
\end{verbatim}

\textbf{Include Files}

ippcp.h

\textbf{Domain Dependencies}

headers: ippcore.h

Libraries: ippcore.lib

\textbf{Parameters}

\begin{itemize}
  \item \textit{nTrials}\hspace{1em}Security parameter specified for the Miller-Rabin probable primality.
  \item \textit{pResult}\hspace{1em}Pointer to the validation result.
  \item \textit{pCtx}\hspace{1em}Pointer to the cryptosystem context.
  \item \textit{rndFunc}\hspace{1em}Specified Random Generator.
  \item \textit{pRndParam}\hspace{1em}Pointer to the Random Generator context.
\end{itemize}

\textbf{Description}

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:

\begin{itemize}
  \item \text{ippDLValid}\hspace{1em}\text{Validation has passed successfully.}
  \item \text{ippDLBaseIsEven}\hspace{1em}\text{\( P \) is even.}
  \item \text{ippDLOrderIsEven}\hspace{1em}\text{\( R \) is even.}
  \item \text{ippDLoInvalidBaseRange}\hspace{1em}\( P \leq 2^{peBits-1}\) or \( P \geq 2^{peBits} \).
  \item \text{ippDLoInvalidOrderRange}\hspace{1em}\( R \leq 2^{reBits-1}\) or \( R \geq 2^{reBits} \).
  \item \text{ippDLCCompositeBase}\hspace{1em}\text{\( P \) is not a prime.}
  \item \text{ippDLCCompositeOrder}\hspace{1em}\text{\( R \) is not a prime.}
  \item \text{ippDLoInvalidCofactor}\hspace{1em}\text{\( R \) is not divisible by (\( P -1 \)).}
  \item \text{ippDLoInvalidGenerator}\hspace{1em}(1 < G < (P -1)) is false or \( G^R \neq 1 \) (mod \( P \)).
\end{itemize}

To ensure that both \( p \) and \( r \) are primes, the function applies \textit{nTrials}-round Miller-Rabin primality test. Test data for primality test is provided by the specified \textit{rndFunc} Random Generator.
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.

ippStsBadArgErr  Indicates an error condition if nTrials < 1.

DLPSharedSecretDH

Computes a shared field element by using the Diffie-Hellman scheme.

Syntax

IppStatus ippsDLPSharedSecretDH(const IppsBigNumState* pPrivateA, const IppsBigNumState* pPublicB, IppsBigNumState* pShare, IppsDLPState* pCtx);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pPrivateA  Pointer to your own private key privateKeyA.

pPublicB  Pointer to the public key pubKeyB belonging to the other party.

pShare  Pointer to the shared secret element Share.

pCtx  Pointer to the cryptosystem context.

Description

The function computes a shared secret element \( FG(p) \) \( pubKeyB^{privateKeyA}(modp) \).

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.

ippStsRangeErr  Indicates an error condition if Share does not have enough space.
DLGetResultString
For DL-based cryptosystems, returns the character string corresponding to code that represents the result of validation.

Syntax
const char* ippsDLGetResultString(IppDLResult code);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
code
The code of the validation result.

Description
For DL-based cryptosystems, the function returns the character string corresponding to code that represents the result of validation.

Return Values
Possible values of code and the corresponding character strings are as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>&quot;Unknown DL result&quot;</td>
</tr>
<tr>
<td>ippDLValid</td>
<td>&quot;Validation passed successfully&quot;</td>
</tr>
<tr>
<td>ippDLEven</td>
<td>&quot;Base is even&quot;</td>
</tr>
<tr>
<td>ippDLOdd</td>
<td>&quot;Order is even&quot;</td>
</tr>
<tr>
<td>ippDLInvalidBase</td>
<td>&quot;Invalid Base (P) range&quot;</td>
</tr>
<tr>
<td>ippDLInvalidOrder</td>
<td>&quot;Invalid Order (R) range&quot;</td>
</tr>
<tr>
<td>ippDLComposite</td>
<td>&quot;Composite Base (P)&quot;</td>
</tr>
<tr>
<td>ippDLComposite</td>
<td>&quot;Composite Order (R)&quot;</td>
</tr>
<tr>
<td>ippDLInvalidCofactor</td>
<td>&quot;R does not divide (P -1)&quot;</td>
</tr>
<tr>
<td>ippDLInvalidG</td>
<td>&quot;1 != G^R (mod P)&quot;</td>
</tr>
<tr>
<td>ippDLInvalidPrivate</td>
<td>&quot;Invalid Private Key&quot;</td>
</tr>
<tr>
<td>ippDLInvalidPublic</td>
<td>&quot;Invalid Public Key&quot;</td>
</tr>
<tr>
<td>ippDLInvalidKey</td>
<td>&quot;Invalid Key Pair&quot;</td>
</tr>
<tr>
<td>ippDLInvalidSignature</td>
<td>&quot;Invalid Signature&quot;</td>
</tr>
</tbody>
</table>

See Also
DLPValidateDH
DLPValidateDSA
DLPValidateKeyPair
Elliptic Curve Cryptography Functions

Cryptography Intel® Integrated Performance Primitives (Intel® IPP) offers functions allowing for different operations with an elliptic curve defined over a prime finite field $\text{GF}(p)$.

The functions are based on standards [IEEE P1363A], [SEC1], [ANSI], and [SM2].

Cryptography for Intel IPP supports some elliptic curves with fixed parameters, the so-called standard or recommended curves. These parameters are chosen so that they provide a sufficient level of security and enable efficient implementation.

The `IppECCType` enumerator lists standard elliptic curves supported. You can select a particular type in a call to `ECCPSetStd`.

The table below associates each value of `IppECCType` with parameters of the elliptic curve and provides a reference to the appropriate specification.

<table>
<thead>
<tr>
<th>Value of <code>IppECCType</code></th>
<th>Name of the Curve</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippECarbitrary</td>
<td>Not applicable</td>
<td>No reference because of arbitrary parameters.</td>
</tr>
<tr>
<td>ippECstd112r1</td>
<td>secp112r1</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd112r2</td>
<td>secp112r2</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd128r1</td>
<td>secp128r1</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd128r2</td>
<td>secp128r2</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd160r1</td>
<td>secp160r1</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd160r2</td>
<td>secp160r2</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd192r1</td>
<td>secp192r1</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd224r1</td>
<td>secp224r1</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd256r1</td>
<td>secp256r1</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd384r1</td>
<td>secp384r1</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstd521r1</td>
<td>secp521r1</td>
<td>[SEC2]</td>
</tr>
<tr>
<td>ippECstdSM2</td>
<td>SM2</td>
<td>[SM2]</td>
</tr>
</tbody>
</table>

For more information on parameters recommended for the functions, see [SEC2] and [SM2].

Functions Based on $\text{GF}(p)$

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:

- Setting up operations: `ECCPSet` sets up elliptic curve domain parameters. `ECCPSetKeyPair` sets a pair of public and private keys for the given cryptosystem.
- 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.
- Validation operations: **ECCPValidate** checks validity of the elliptic curve domain parameters. **ECCPValidateKeyPair** validates correctness of the public and private keys.
- Generation operations: **ECCPGenKeyPair** generates a private key and computes a public key for the given elliptic cryptosystem.
- 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**

```c
IppStatus ippsECCPGetSize(int feBitSize, int *pSize);
```

**Include Files**

```c
ippcp.h
```

**Domain Dependencies**

Headers: **ippcore.h**

Libraries: **ippcore.lib**

**Parameters**

- **feBitSize**: Size (in bits) of the field element.
- **pSize**: Pointer to the size (in bytes) of the context.

**Description**

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**

- **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 2.

**ECCPGetSizeStd**

*Gets the size of the IppsECCPState context for a standard elliptic curve.*

**Syntax**

```c
IppStatus ippsECCPGetSizeStd128r1(int* pSize);
IppStatus ippsECCPGetSizeStd128r2(int* pSize);
```
IppStatus ippsECCPGetSizeStd192r1(int* pSize);
IppStatus ippsECCPGetSizeStd224r1(int* pSize);
IppStatus ippsECCPGetSizeStd256r1(int* pSize);
IppStatus ippsECCPGetSizeStd384r1(int* pSize);
IppStatus ippsECCPGetSizeStd521r1(int* pSize);
IppStatus ippsECCPGetSizeStdSM2(int* pSize);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

$pSize$  Pointer to the size (in bytes) of the IppsECCPState context for a standard elliptic curve.

Description
Each of these functions computes the size of the context in bytes for the elliptic curve cryptosystem based on a specific standard elliptic curve. For a list of these curves, see table Standard Elliptic Curves.

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.

ECCPInit
Initializes the context for the elliptic curve cryptosystem over GF(p).

Syntax
IppStatus ippsECCPInit(int feBitSize, IppsECCPState* pECC);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

$feBitSize$  Size (in bits) of a field element.
$pECC$  Pointer to the cryptosystem context.
Description

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

- **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 2.

See Also

Data Security Considerations

**ECCPInitStd**

*Initializes the context for the cryptosystem based on a standard elliptic curve.*

Syntax

```
IppStatus ippsECCPInitStd128r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd128r2(IppsECCPState* pECC);
IppStatus ippsECCPInitStd192r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd192r2(IppsECCPState* pECC);
IppStatus ippsECCPInitStd224r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd224r2(IppsECCPState* pECC);
IppStatus ippsECCPInitStd256r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd256r2(IppsECCPState* pECC);
IppStatus ippsECCPInitStd384r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd384r2(IppsECCPState* pECC);
IppStatus ippsECCPInitStd521r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd521r2(IppsECCPState* pECC);
IppStatus ippsECCPInitStdSM2(IppsECCPState* pECC);
```

Include Files

`ippcp.h`

Domain Dependencies

Headers: `ippcore.h`

Libraries: `ippcore.lib`

Parameters

- **pECC**
  
  Pointer to the cryptosystem context based on a standard elliptic curve.

Description

Each of these functions initializes the context of the elliptic curve cryptosystem based on a specific standard elliptic curve. For a list of these curves, see table **Standard Elliptic Curves**.
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.

See Also

Data Security Considerations

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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pPrime Pointer to the characteristic \( p \) of the prime finite field GF(p).
pA Pointer to the coefficient \( A \) of the equation defining the elliptic curve.
pB Pointer to the coefficient \( B \) of the equation defining the elliptic curve.
pGX Pointer to the \( x \)-coordinate of the elliptic curve base point.
pGY Pointer to the \( y \)-coordinate of the elliptic curve base point.
pOrder Pointer to the order of the elliptic curve base point.
cofactor Cofactor.
pECC Pointer to the context of the cryptosystem.

Description

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:
  \( y^2 = x^3 + A \cdot x + B \mod p \).
- \( pGX, pGY \) 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:
The domain parameters are set in the cryptosystem context which must be already created by the `ECCPGetSize` and `ECCPInit` functions.

**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 `pPrime`, `pA`, `pB`, `pGX`, `pGY`, `pOrder`, and `pECC` is not valid.

- **ippStsRangeErr**
  Indicates an error condition if one of the parameters pointed by `pPrime`, `pA`, `pB`, `pGX`, `pGY`, and `pOrder` cannot embed the `feBitSize` bits length or the value of `cofactor` is less than 1.

**ECCPSetStd**

*Sets up a recommended set of domain parameters for an elliptic curve over \( \text{GF}(p) \).*

**Syntax**

```
IppStatus ippsECCPSetStd128r1(IppsECCPState* pECC);
IppStatus ippsECCPSetStd128r2(IppsECCPState* pECC);
IppStatus ippsECCPSetStd192r1(IppsECCPState* pECC);
IppStatus ippsECCPSetStd224r1(IppsECCPState* pECC);
IppStatus ippsECCPSetStd256r1(IppsECCPState* pECC);
IppStatus ippsECCPSetStd384r1(IppsECCPState* pECC);
IppStatus ippsECCPSetStd521r1(IppsECCPState* pECC);
IppStatus ippsECCPSetStdSM2(IppsECCPState* pECC);
IppStatus ippsECCPSetStd(IppECCType flag, IppsECCPState* pECC);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib

**Parameters**

- **flag**
  Set specifier.

- **pECC**
  Pointer to the cryptosystem context.

**Description**

Each of the `ECCPSetStd` functions sets a recommended set of domain parameters for an elliptic curve over a prime finite field \( \text{GF}(p) \).

**Functions with One Parameter**
All the functions but the last one set domain parameters for standard elliptic curves, listed in table Standard Elliptic Curves. Before a call to each of these functions, create the cryptosystem context by calling the appropriate ECCPGetSizeStd and ECCPInitStd functions.

**Function with Two Parameters**

For the last function, the value of the parameter flag defines the set of domain parameters. Possible values of flag are as follows:

- IppECCPStd112r1: For the cryptosystem context where feBitSize == 112
- IppECCPStd112r2: For the cryptosystem context where feBitSize == 112
- IppECCPStd128r1: For the cryptosystem context where feBitSize == 128
- IppECCPStd128r2: For the cryptosystem context where feBitSize == 128
- IppECCPStd160r1: For the cryptosystem context where feBitSize == 160
- IppECCPStd160r2: For the cryptosystem context where feBitSize == 160
- IppECCPStd192r1: For the cryptosystem context where feBitSize == 192
- IppECCPStd224r1: For the cryptosystem context where feBitSize == 224
- IppECCPStd256r1: For the cryptosystem context where feBitSize == 256
- IppECCPStd384r1: For the cryptosystem context where feBitSize == 384
- IppECCPStd521r1: For the cryptosystem context where feBitSize == 521.

For more information on parameter values for the recommended elliptic curves, see [SEC2].

Before a call to this function, create the cryptosystem context by calling the ECCPGetSize and ECCPInit functions. The value of feBitSize is applied when these functions are called and predetermines the choice of the flag value.

**Return Values**

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippStsNoErr</td>
<td>Indicates no error. Any other value indicates an error or warning.</td>
</tr>
<tr>
<td>ippStsNullPtrErr</td>
<td>Indicates an error condition if any of the specified pointers is NULL.</td>
</tr>
<tr>
<td>ippStsContextMatchErr</td>
<td>Indicates an error condition if the cryptosystem context is not valid.</td>
</tr>
<tr>
<td>ippStsECCInvalidFlagErr</td>
<td>Indicates an error condition if the value of the parameter flag is not valid.</td>
</tr>
</tbody>
</table>

**ECCPGet**

*Retrieves elliptic curve domain parameters over GF(p).*

**Syntax**

```c
IppStatus ippsECCPGet(IppsBigNumState* pPrime, IppsBigNumState* pA, IppsBigNumState* pB, IppsBigNumState* pGx, IppsBigNumState* pGY, IppsBigNumState* pOrder, int* cofactor, IppsECCPState* pECC);
```

**Include Files**

ippcp.h
Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pPrime
  Pointer to the characteristic $p$ of the prime finite field GF($p$).
pA
  Pointer to the coefficient $A$ of the equation defining the elliptic curve.
pB
  Pointer to the coefficient $B$ of the equation defining the elliptic curve.
pGX
  Pointer to the $x$-coordinate of the elliptic curve base point.
pGY
  Pointer to the $y$-coordinate of the elliptic curve base point.
pOrder
  Pointer to the order $n$ of the elliptic curve base point.
cofactor
  Pointer to the cofactor $h$.
pECC
  Pointer to the context of the cryptosystem.

Description
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
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 pPrime, pA, pB, pGX, pGY, pOrder, or pECC is not valid.
ippStsRangeErr
  Indicates an error condition if the memory size of one of the parameters pointed by pPrime, pA, pB, pGX, pGY, pOrder, and pECC is less than the value of feBitSize in the ECCPInit function.

ECCPGetOrderBitSize
Retrieves order size of the elliptic curve base point over GF($p$) in bits.

Syntax
IppStatus ippsECCPGetOrderBitSize(int* pBitSize, IppsECCPState* pECC);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pBitSize

Pointer to the size of the base point (in bits).

pECC

Pointer to the cryptosystem context.

Description

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 \( p\text{BitsSize} \). The elliptic curve domain parameters must be hitherto defined by one of the functions: \( \text{ECCPSet} \) or \( \text{ECCPSetStd} \).

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 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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

nTrials

A number of attempts made to check the number for primality.

pResult

Pointer to the result received upon the check of the elliptic curve domain parameters.

pECC

Pointer to the cryptosystem context.

rndFunc

Specified Random Generator.

pRndParam

Pointer to Random Generator context.

Description

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 \( p\text{Result} \).
Elliptic curve domain parameters must be hitherto defined by one of the functions: \texttt{ECCPSet} or \texttt{ECCPSetStd}. The purpose of the parameters \texttt{rndFunc}, \texttt{pRndParam}, and \texttt{nTrials} is analogous to that of the parameters \texttt{rndFunc}, \texttt{pRndParam}, and \texttt{nTrials} in the \texttt{PrimeTest} function.

The result of the elliptic curve domain parameters check can take one of the following values:

- \texttt{ippECValid}: The parameters are valid.
- \texttt{ippECCompositeBase}: The prime finite field characteristic \( p \) is a composite number.
- \texttt{ippECIsNotAG}: 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.
- \texttt{ippECPIntIsNotValid}: The base point \( G \) is not on the elliptic curve.
- \texttt{ippECCompositeOrder}: The order \( n \) of the base point \( G \) is a composite number.
- \texttt{ippECInvalidOrder}: 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.
- \texttt{ippECIsWeakSSSA}: The order \( n \) of the base point \( G \) is equal to the finite field characteristic \( p \).
- \texttt{ippECIsWeakMOV}: The curve is excluded because it is subject to the MOV reduction attack.

\textbf{Return Values}

- \texttt{ippStsNoErr}: Indicates no error. Any other value indicates an error or warning.
- \texttt{ippStsNullPtrErr}: Indicates an error condition if any of the specified pointers is NULL.
- \texttt{ippStsContextMatchErr}: Indicates an error condition if one of the contexts pointed by \( c \) or \( pECC \) is not valid.
- \texttt{ippStsBadArgErr}: Indicates an error condition if the memory size of the parameter \texttt{seed} is less than five words (32 bytes in each) or the value of the parameter \texttt{nTrials} is less than 1.

\textbf{ECCPPointGetSize}

\textit{Gets the size of the IppsECCPPoint context in bytes for a point on the elliptic curve point defined over \textit{GF}(\textit{p})}.

\textbf{Syntax}

\begin{verbatim}
IppStatus ippsECCPPointGetSize(int feBitSize, int* pSize);
\end{verbatim}

\textbf{Include Files}

ippcp.h

\textbf{Domain Dependencies}

\textit{Headers}: ippcore.h

\textit{Libraries}: ippcore.lib
Parameters

feBitSize  Size (in bits) of the field element.
pSize  Pointer to the context size.

Description

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 IppsECCPPoint intended for storing the information about a point on the elliptic curve defined over GF(p).

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 2.

ECCPPointInit

Initializes the context for a point on the elliptic curve defined over GF(p).

Syntax

IppStatus ippsECCPPointInit(int feBitSize, IppsECCPPointState* pPoint);

Include Files

ippcp.h

Domain Dependencies

Headers:  ippcore.h
Libraries:  ippcore.lib

Parameters

feBitSize  Size (in bits) of the field element.
pPoint  Pointer to the context of the elliptic curve point.

Description

The function initializes the context for a point on the elliptic curve defined over a finite field GF(p).

Context is a structure IppsECCPPointState intended for storing the information about a point on the elliptic curve defined over GF(p).

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 2.

See Also
Data Security Considerations

ECCPSetPoint
Sets coordinates of a point on the elliptic curve defined over GF(p).

Syntax
IppStatus ippsECCPSetPoint(const IppsBigNumState* pX, const IppsBigNumState* pY, IppsECCPPointState* pPoint, IppsECCPState* pECC);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters
pX Pointer to the x-coordinate of the point on the elliptic curve.
pY Pointer to the y-coordinate of the point on the elliptic curve.
pPoint Pointer to the context of the elliptic curve point.
pECC Pointer to the context of the elliptic cryptosystem.

Description
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
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 pX, pY, pPoint, or pECC is not valid.

ECCPSetPointAtInfinity
Sets the point at infinity.

Syntax
IppStatus ippsECCPSetPointAtInfinity(IppsECCPPointState* pPoint, IppsECCPState* pECC);
Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pPoint  Pointer to the context of the elliptic curve point.
pECC    Pointer to the context of the elliptic cryptosystem.

Description
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

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 pPoint or pECC 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 IppsECCPPointState* pPoint, IppsECCPState* pECC);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pX  Pointer to the x-coordinate of the point on the elliptic curve.
pY  Pointer to the y-coordinate of the point on the elliptic curve.
pPoint  Pointer to the context of the elliptic curve point.
pECC  Pointer to the context of the elliptic cryptosystem.
Description

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: \texttt{ECCPSet} or \texttt{ECCPSetStd}.

Return Values

- \texttt{ippStsNoErr} \hspace{1cm} Indicates no error. Any other value indicates an error or warning.
- \texttt{ippStsNullPtrErr} \hspace{1cm} Indicates an error condition if any of the specified pointers is NULL.
- \texttt{ippStsContextMatchErr} \hspace{1cm} 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

\[
\text{IppStatus ippsECCPCheckPoint(const IppsECCPPointState* } pP, \text{ IppECResult* } pResult, \text{ IppsECCPState* } pECC);\]

Include Files

\texttt{ippcp.h}

Domain Dependencies

Headers: \texttt{ippcore.h}
Libraries: \texttt{ippcore.lib}

Parameters

- \( pP \) \hspace{1cm} Pointer to the elliptic curve point.
- \( pResult \) \hspace{1cm} Pointer to the result of the check.
- \( pECC \) \hspace{1cm} Pointer to the context of the elliptic cryptosystem.

Description

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: \texttt{ECCPSet} or \texttt{ECCPSetStd}.

The result of the check for the correctness of the point can take one of the following values:

- \texttt{ippECValid} \hspace{1cm} Point is on the elliptic curve.
- \texttt{ippECPointIsNotValid} \hspace{1cm} Point is not on the elliptic curve and is not the point at infinity.
- \texttt{ippECPointIsAtInfinite} \hspace{1cm} 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 IppsECCPPointState* pP, const IppsECCPPointState* pQ, IppECResult* pResult, IppsECCPState* pECC);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pP Pointer to the elliptic curve point P.

pQ Pointer to the elliptic curve point Q.

pResult Pointer to the comparison result of two points: P and Q.

pECC Pointer to the context of the elliptic cryptosystem.

Description

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: ECCSet or ECCPSetStd.

The comparison result of two points P and Q can take one of the following values:

ippECPointIsEqual Points P and Q are equal.

ippECPointIsNotEqual Points P and Q are different.

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.
ECCPNegativePoint
Finds an elliptic curve point which is an additive inverse for the given point over GF(p).

Syntax
IppStatus ippsECCPNegativePoint(const IppsECCPointState* pP, IppsECCPointState* pR, IppsECCPState* pECC);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pP Pointer to the elliptic curve point P.
pR Pointer to the elliptic curve point R.
pECC Pointer to the context of the elliptic cryptosystem.

Description
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

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, pR, or pECC is not valid.

ECCPAddPoint
Computes the addition of two elliptic curve points over GF(p).

Syntax
IppStatus ippsECCPAddPoint(const IppsECCPointState* pP, const IppsECCPointState* pQ, IppsECCPointState* pR, IppsECCPState* pECC);

Include Files
ippcp.h

Domain Dependencies
Headers: ippcore.h
Parameters

- **pP**: Pointer to the elliptic curve point $P$.
- **pQ**: Pointer to the elliptic curve point $Q$.
- **pR**: Pointer to the elliptic curve point $R$.
- **pECC**: Pointer to the context of the elliptic cryptosystem.

Description

The function calculates the addition of two elliptic curve points $P$ and $Q$ over a finite field $\text{GF}(p)$ 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

- **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 $pP$, $pQ$, $pR$, or $pECC$ is not valid.

ECCPMulPointScalar

Performs scalar multiplication of a point on the elliptic curve defined over $\text{GF}(p)$.

Syntax

```c
IppStatus ippsECCPMulPointScalar(const IppsECCPPointState* pP, const IppsBigNumState* pK, IppsECCPPointState* pR, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

Domain Dependencies

Headers: `ippcore.h`
Libraries: `ippcore.lib`

Parameters

- **pP**: Pointer to the elliptic curve point $P$.
- **pK**: Pointer to the scalar $K$.
- **pR**: Pointer to the elliptic curve point $R$.
- **pECC**: Pointer to the context of the elliptic cryptosystem.

Description

The function performs the $K$ scalar multiplication of an elliptic curve point $P$ over $\text{GF}(p)$ 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**

- **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`, `pK`, `pR`, or `pECC` is not valid.

**ECCPGenKeyPair**

*Generates a private key and computes public keys of the elliptic cryptosystem over GF(p).*

**Syntax**

```c
IppStatus ippsECCPGenKeyPair(IppsBigNumState* pPrivate, IppsECCPPointState* pPublic, IppsECCPState* pECC, IppBitSupplier rndFunc, void* pRndParam);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

*Headers:* `ippcore.h`

*Libraries:* `ippcore.lib`

**Parameters**

- **pPrivate**
  Pointer to the private key `privKey`.

- **pPublic**
  Pointer to the public key `pubKey`.

- **pECC**
  Pointer to the context of the elliptic cryptosystem.

- **rndFunc**
  Specified Random Generator.

- **pRndParam**
  Pointer to the Random Generator context.

**Description**

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 `pubKey = privKey · 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**

- 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 pPrivate, pPublic, or pECC is not valid.
- ippStsSizeErr: Indicates an error condition if the memory size of the parameter privKey pointed by pPrivate 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**

```c
IppStatus ippsECCPPublicKey(const IppsBigNumState* pPrivate, IppsECCPPointState* pPublic, IppsECCPState* pECC);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- pPrivate: Pointer to the private key privKey.
- pPublic: Pointer to the public key pubKey.
- pECC: Pointer to the context of the elliptic cryptosystem.

**Description**

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 pubKey = privKey · 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**

- 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.
Indicates an error condition if one of the contexts pointed by pPrivate, pPublic, or pECC is not valid.

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**

```c
IppStatus ippsECCPValidateKeyPair(const IppsBigNumState* pPrivate, const IppsECCPPointState* pPublic, IppECResult* pResult, IppsECCPState* pECC);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pPrivate**: Pointer to the private key privKey.
- **pPublic**: Pointer to the public key pubKey.
- **pResult**: Pointer to the validation result.
- **pECC**: Pointer to the context of the elliptic cryptosystem.

**Description**

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 · 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:

- **ippECValid**: Keys are valid.
- **ippECInvalidKeyPair**: Keys are not valid because privKey · G≠pubKey
- **ippECInvalidPrivateKey**: Key privKey falls outside the range of [1, n-1].
- **ippECPointIsAtInfinite**: Key pubKey is the point at infinity.
- **ippECInvalidPublicKey**: Key pubKey is not valid because n · pubKey≠O, where O is the point at infinity.

**Return Values**

- **ippStsNoErr**: Indicates no error. Any other value indicates an error or warning.
ECCPSetKeyPair

Sets private and/or public keys of the elliptic cryptosystem over GF(p).

Syntax

IppStatus ippsECCPSetKeyPair(const IppsBigNumState* pPrivate, const IppsECCPPointState* pPublic, IppBool regular, IppsECCPState* pECC);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pPrivate  Pointer to the private key *privKey*.
pPublic   Pointer to the public key *pubKey*.
regular   Key status flag.
pECC      Pointer to the context of the elliptic cryptosystem.

Description

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 · G*, where G is the base point of the elliptic curve.

The two possible values of the parameter *regular* define the key timeliness status:

ippiTrue   Keys are regular.
ippiFalse  Keys are ephemeral.

The elliptic curve domain parameters must be hitherto defined by one of the functions: ECCSet or ECCPSetStd.

Return Values

ippiStsNoErr  Indicates no error. Any other value indicates an error or warning.
ippiStsNullPtrErr  Indicates an error condition if any of the specified pointers is NULL.
ippiStsContextMatchErr  Indicates an error condition if one of the contexts pointed by *pPrivate, pPublic, or pECC* is not valid.
ECCPSharedSecretDH
Computes a shared secret field element by using the
Diffie-Hellman scheme.

Syntax

IppStatus ippsECCPSharedSecretDH(const IppsBigNumState* pPrivate, const
IppsECCPPointState* pPublic, IppsBigNumState* pShare, IppsECCPState* pECC);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pPrivate Pointer to your own public key pubKey.
pPublic Pointer to the public key pubKey.
pShare Pointer to the secret number bnShare.
pECC Pointer to the context of the elliptic cryptosystem.

Description

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 ·
   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 · 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 · pubKeyB =
   privKeyA · privKeyB · 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 · pubKeyA =
   privKeyB · privKeyA · G.

Because the following equation is true privKeyA · privKeyB · G = privKeyB · privKeyA · 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

- 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 pPublic, pPShare, or pECC is not valid.
- ippStsRangeErr: Indicates an error condition if the memory size of bnShare pointed by pShare is less than the value of feBitSize in the function ECCPInit.
- ippStsShareKeyErr: 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);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

pPrivate: Pointer to your own public key pubKey.
pPublic: Pointer to the public key pubKey.
pShare: Pointer to the secret number bnShare.
pECC: Pointer to the context of the elliptic cryptosystem.

Description

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 · 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 · 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 · privKeyA · pubKeyB = h · privKeyA · privKeyB · G, where h is the elliptic curve cofactor.
4. Bob gets Alice's public key and calculates the secret point \( \text{share}_B \). When calculating, he uses his own private key and Alice's public key and applies the following formula: \( \text{share}_B = h \cdot \text{privKey}_B \cdot \text{pubKey}_A = h \cdot \text{privKey}_B \cdot \text{privKey}_A \cdot G \), 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: \text{ECCPSet} or \text{ECCPSetStd}.

**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 \( pPublic, pShare, \) or \( pECC \) is not valid.
- **ippStsRangeErr**: Indicates an error condition if the memory size of \( bnShare \) pointed by \( pShare \) is less than the value of \( feBitSize \) in the function \text{ECCPInit}.
- **ippStsShareKeyErr**: 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**

\[
\text{IppStatus ippsECCPSignDSA(const IppsBigNumState* } \text{pMsgDigest, const IppsBigNumState* } \text{pPrivate, IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCPState* pECC);}
\]

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h
Libraries: ippcore.lib

**Parameters**

- **pMsgDigest**: Pointer to the message digest \( msg \) to be digitally signed, that is, to be encrypted with a private key.
- **pPrivate**: Pointer to the signer's regular private key.
- **pSignX**: Pointer to the integer \( r \) of the digital signature.
- **pSignY**: Pointer to the integer \( s \) of the digital signature.
- **pECC**: Pointer to the context of the elliptic cryptosystem.

**Description**

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:

- $\text{regPrivKey}$: Regular private key.
- $\text{ephPrivKey}$: Ephemeral private key.
- $\text{ephPubKey}$: Ephemeral public key.

The keys can be generated and set up by the functions $\text{ECCPGenKeyPair}$ and $\text{ECCPSetKeyPair}$ with only requirement that the key $\text{regPrivKey}$ be different from the key $\text{ephPrivKey}$.

The elliptic curve domain parameters must be hitherto defined by one of the functions: $\text{ECCPSet}$ or $\text{ECCPSetStd}$.

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 $p\text{MsgDigest}$, $p\text{SignX}$, $p\text{SignY}$, or $p\text{ECC}$ is not valid.
- ippStsMessageErr: Indicates an error condition if the value of $msg$ pointed by $p\text{MsgDigest}$ falls outside the range of $[1, n-1]$ where $n$ is the order of the elliptic curve base point $G$.
- ippStsRangeErr: Indicates an error condition if one of the parameters pointed by $p\text{SignX}$ or $p\text{SignY}$ has a less memory size than the order $n$ of the elliptic curve base point $G$.
- ippStsEphemeralKeyErr: Indicates an error condition if the values of the ephemeral keys $\text{ephPrivKey}$ and $\text{ephPubKey}$ are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

**See Also**

Signing/Verification Using the Elliptic Curve Cryptography Functions over a Prime Finite Field

**ECCPVerifyDSA**

*Verifies authenticity of the digital signature over a message digest (ECDSA).*

**Syntax**

```c
IppStatus ippsECCPVerifyDSA(const IppsBigNumState* pMsgDigest, const IppsBigNumState* pSignX, const IppsBigNumState* pSignY, IppECResult* pResult, IppsECCPState* pECC);
```

**Include Files**

ippcp.h

**Domain Dependencies**

Headers: ippcore.h

Libraries: ippcore.lib
Parameters

- `pMsgDigest`: Pointer to the message digest `msg`.
- `pSignX`: Pointer to the integer `r` of the digital signature.
- `pSignY`: Pointer to the integer `s` of the digital signature.
- `pResult`: Pointer to the digital signature verification result.
- `pECC`: Pointer to the context of the elliptic cryptosystem.

Description

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 base point `G`.

See Also

- Signing/Verification Using the Elliptic Curve Cryptography Functions over a Prime Finite Field

**ECCPSignNR**

*Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).*

Syntax

```c
IppStatus ippsECCPSignNR(const IppsBigNumState* pMsgDigest, const IppsBigNumState* pPrivate, IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCPState* pECC);
```
Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pMsgDigest**: Pointer to the message digest \( msg \).
- **pPrivate**: Pointer to the private key \( privKey \).
- **pSignX**: Pointer to the integer \( r \) of the digital signature.
- **pSignY**: Pointer to the integer \( s \) of the digital signature.
- **pECC**: Pointer to the context of the elliptic cryptosystem.

Description

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:

- **regPrivKey**: Regular private key.
- **ephPrivKey**: Ephemeral private key.
- **ephPubKey**: 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

- **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 \( G \).
- **ippStsRangeErr**: Indicates an error condition if one of the parameters pointed by **pSignX** or **pSignY** has a less memory size than the order \( n \) of the elliptic curve base point \( G \).
Indicates an error condition if the values of the ephemeral keys `ephPrivKey` and `ephPubKey` are not valid. (Either $r = 0$ or $s = 0$ 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**

```c
IppStatus ippsECCPVerifyNR(const IppsBigNumState* pMsgDigest, const IppsBigNumState* pSignX, const IppsBigNumState* pSignY, IppECResult* pResult, IppsECCPState* pECC);
```

**Include Files**

`ippcp.h`

**Domain Dependencies**

**Headers:** `ippcore.h`

**Libraries:** `ippcore.lib`

**Parameters**

- `pMsgDigest`: Pointer to the message digest `msg`.
- `pSignX`: Pointer to the integer $r$ of the digital signature.
- `pSignY`: Pointer to the integer $s$ of the digital signature.
- `pResult`: Pointer to the digital signature verification result.
- `pECC`: Pointer to the context of the elliptic cryptosystem.

**Description**

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:

- `ippECValid`: The digital signature is valid.
- `ippECInvalidSignature`: 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

- 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.

ECCPSignSM2

*Computes a digital signature over a message digest using the SM2 scheme.*

Syntax

```c
IppStatus ippsECCPSignSM2(const IppsBigNumState* pMsgDigest, const IppsBigNumState* pRegPrivate, const IppsBigNumState* pEphPrivate, IppsBigNumState* pSignR, IppsBigNumState* pSignS, IppsECCPState* pECC);
```

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

- **pMsgDigest**: Pointer to the message digest msg.
- **pRegPrivate**: Pointer to the regular private key regPrivKey.
- **pEphPrivate**: Pointer to the ephemeral private key ephPrivKey.
- **pSignR**: Pointer to the integer r of the digital signature.
- **pSignS**: Pointer to the integer s of the digital signature.
- **pECC**: Pointer to the context of the elliptic cryptosystem.

Description

The function computes two big numbers r and s that form the digital signature over a message digest msg.

The digital signature is computed using the SM2 scheme [SM2]. The scheme requires that the following cryptosystem keys are set up by the message sender:

- **regPrivKey**: Regular private key.
- **ephPrivKey**: Ephemeral private key.
- **ephPubKey**: Ephemeral public key.

You can generate and set up the keys by calling the ECCGenKeyPair and ECCSetKeyPair functions with the only requirement that the key regPrivKey is different from the key ephPrivKey.
Before calling \texttt{ECCPSignSM2}, set up the domain parameters of the elliptic curve in the \texttt{*pECC} context by calling one of the functions: \texttt{ECCPSet} or \texttt{ECCPSetStdSM2}.

\textbf{Return Values}

\begin{itemize}
  \item \texttt{ippStsNoErr} \quad Indicates no error. Any other value indicates an error or warning.
  \item \texttt{ippStsNullPtrErr} \quad Indicates an error condition if any of the specified pointers is NULL.
  \item \texttt{ippStsContextMatchErr} \quad Indicates an error condition if one of the specified contexts is not valid.
  \item \texttt{ippStsMessageErr} \quad Indicates an error condition if the value of \texttt{msg} pointed by \texttt{pMsgDigest} falls outside the range of \([1, n-1]\) where \(n\) is the order of the elliptic curve base point \(G\).
  \item \texttt{ippStsRangeErr} \quad Indicates an error condition if one of the parameters pointed by \texttt{pSignR} or \texttt{pSignS} has a smaller memory size than the order \(n\) of the elliptic curve base point \(G\).
  \item \texttt{ippStsEphemeralKeyErr} \quad Indicates an error condition if the values of the ephemeral keys \texttt{ephPrivKey} and \texttt{ephPubKey} are not valid. (Either \(r = 0\) or \(s = 0\) is received as a result of the digital signature calculation).
\end{itemize}

\textbf{ECCPVerifySM2}

\textit{Verifies authenticity of a digital signature over a message digest using the SM2 scheme.}

\textbf{Syntax}

\begin{verbatim}
IppStatus ippsECCPVerifySM2(const IppsBigNumState* pMsgDigest, const IppsECCPPointState* pRegPublic, const IppsBigNumState* pSignR, const IppsBigNumState* pSignS, IppECResult* pResult, IppsECCPState* pECC);
\end{verbatim}

\textbf{Include Files}

ippcp.h

\textbf{Domain Dependencies}

\textbf{Headers:} ippcore.h

\textbf{Libraries:} ippcore.lib

\textbf{Parameters}

\begin{itemize}
  \item \texttt{pMsgDigest} \quad Pointer to the message digest \texttt{msg}.
  \item \texttt{pRegPublic} \quad Pointer to the message sender's regular private key \texttt{regPubKey}.
  \item \texttt{pSignR} \quad Pointer to the integer \(r\) of the digital signature.
  \item \texttt{pSignS} \quad Pointer to the integer \(s\) of the digital signature.
  \item \texttt{pResult} \quad Pointer to the digital signature verification result.
  \item \texttt{pECC} \quad Pointer to the context of the elliptic cryptosystem.
\end{itemize}

\textbf{Description}

The function verifies authenticity of the digital signature, represented as integer big numbers \(r\) and \(s\), over a message digest \texttt{msg}. The digital signature over the message digest \texttt{msg} must be computed using the SM2 scheme [SM2] by to the \texttt{ECCPSignSM2} function.
The scheme requires the following cryptosystem key set up by the message sender:

\textit{regPubKey} \hspace{1cm} \text{Message sender's regular private key.}

You can generate and set up the key in a call to the \texttt{ECCPGenKeyPair} function.

The result of the digital signature verification can take one of these values:

\begin{itemize}
  \item \texttt{ippECValid} \hspace{1cm} \text{The digital signature is valid.}
  \item \texttt{ippECInvalidSignature} \hspace{1cm} \text{The digital signature is not valid.}
\end{itemize}

Before calling \texttt{ECCPVerifySM2}, set up the domain parameters of the elliptic curve in the \*p\texttt{ECC} context by calling one of the functions: \texttt{ECCSet} or \texttt{ECCPSetStdSM2}.

\section*{Return Values}

\begin{itemize}
  \item \texttt{ippStsNoErr} \hspace{1cm} \text{Indicates no error. Any other value indicates an error or warning.}
  \item \texttt{ippStsNullPtrErr} \hspace{1cm} \text{Indicates an error condition if any of the specified pointers is NULL.}
  \item \texttt{ippStsContextMatchErr} \hspace{1cm} \text{Indicates an error condition if one of the specified contexts is not valid.}
  \item \texttt{ippStsMessageErr} \hspace{1cm} \text{Indicates an error condition if the value of \textit{msg} pointed by \texttt{pMsgDigest} falls outside the range of \([1, n-1]\) where \(n\) is the order of the elliptic curve base point \(G\).}
\end{itemize}

\section*{Signing/Verification Using the Elliptic Curve Cryptography Functions over a Prime Finite Field}

\section*{Use of \texttt{ECCPSignDSA, ECCPVerifyDSA}}

\begin{verbatim}
#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(IppECCPStd256r1, 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)
{

}
\end{verbatim}
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(ECP)
    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];
ippsPRNGen(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
IppsECCPPointState* regPublic = newECP_256_Point();
IppsECCPPointState* ephPublic = newECP_256_Point();

    // generate regular & ephemeral key pairs, should be different each other
ippsECCPGenKeyPair(regPrivate, regPublic, pECP, ippsPRNGen, pRandGen);
ippsECCPGenKeyPair(ephPrivate, ephPublic, pECP, ippsPRNGen, pRandGen);

    // signature
    //
    // set ephemeral key pair
ippsECCPSetKeyPair(ephPrivate, ephPublic, ippFalse, pECP);
    // compute signature
IppsBigNumState* signX = newBN(ordSize, 0);
IppsBigNumState* signY = newBN(ordSize, 0);
ippsECCPSignDSA(pMsg, regPrivate, signX, signY, pECP);

// verification
//
ippsECCPSetKeyPair(NULL, regPublic, ippTrue, pECP);
IppECResult eccResult;
ippsECCPVerifyDSA(pMsg, signX, signY, &eccResult, pECP);
if(ippECValid == eccResult)
    cout << "signature verification passed" << endl;
else
    cout << "signature verification 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;
}

ECCGetResultString

For elliptic curve cryptosystems, returns the character string corresponding to code that represents the result of validation.

Syntax

const char* ippsECCGetResultString(IppECResult code);

Include Files

ippcp.h

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Parameters

code  
The code of the validation result.

Description

For elliptic curve cryptosystems, returns the character string corresponding to code that represents the result of validation.

Return Values

Possible values of code and the corresponding character strings are as follows:

default  "Unknown ECC result"
ippECValid
ippECCompositeBase
ippECComplicatedBase
ippECIsZeroDiscriminant
ippECCompositeOrder
ippECInvalidOrder
ippECIsWeakMOV
ippECIsWeakSSSA
ippECIsSupersingular
ippECInvalidPrivateKey
ippECInvalidPublicKey
ippECInvalidKeyPair
ippECPointOutOfGroup
ippECPointAtInfinite
ippECPointIsNotValid
ippECPointIsEqual
ippECPointIsNotEqual
ippECInvalidSignature

"Validation passed successfully"
"Finite Field produced by Composite"
"Too many non-zero terms in the polynomial"
"Zero discriminant"
"Composite Base Point order"
"Composite Base Point order"
"EC cover by MOV Reduction Test"
"EC cover by SS-SA Reduction Test"
"EC is supersingular curve"
"Invalid Private Key"
"Invalid Public Key"
"Invalid Key Pair"
"Point is out of group"
"Point at infinity"
"Invalid EC Point"
"Points are equal"
"Points are different"
"Invalid Signature"

See Also
ECCPValidate
ECCPValidateKeyPair
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

```c
const IppLibraryVersion* ippcpGetLibVersion(void);
```

Include Files

```c
ippcp.h
```

Domain Dependencies

Headers: ippcore.h
Libraries: ippcore.lib

Description

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:

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

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

- **majorBuild**
  - is the number of builds for the (major.minor) version.

- **build**
  - is the total number of Intel IPP builds.

- **Name**
  - is the name of the current library version.

- **Version**
  - is the version string.

- **BuildDate**
  - 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.lib", Version = "7.0 build 205.68", BuildDate = "Jul 20 2011".
```
Example

The code example below shows how to use the function ippcpGetLibVersion.

```c
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_l.lib 7.0 build 205.68

Classes and Functions Used in Examples

This section presents source code of functions and classes used in Example "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 (xsample_bignum.h) declaring the BigNumber class are presented below:

```c
#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 constants
    static const BigNumber& Zero();
    static const BigNumber& One();
    static const BigNumber& Two();

```

268
// arithmetic operators probably need
BigNumber& operator = (const BigNumber& bn);
BigNumber& operator += (const BigNumber& bn);
BigNumber& operator -= (const BigNumber& bn);
BigNumber& operator *= (Ipp32u n);
BigNumber& operator *= (const BigNumber& bn);
BigNumber& operator /= (const BigNumber& bn);
BigNumber& 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
BigNumber Modulo(const BigNumber& a) const;
BigNumber ModAdd(const BigNumber& a, const BigNumber& b) const;
BigNumber ModSub(const BigNumber& a, const BigNumber& b) const;
BigNumber ModMul(const BigNumber& a, const BigNumber& b) const;
BigNumber InverseAdd(const BigNumber& a) const;
BigNumber 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.

```cpp
#include "xsample_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)
{
    IppsBigNumSGN bnSgn;
    int bnBitLen;
    Ipp32u* bnData;
    ippRef_BN(&bnSgn, &bnBitLen, &bnData, pBN);
    create(bnData, BITSIZE_WORD(bnBitLen), bnSgn);
}

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 = (int)(strlen(s) + 7)/8;
} else {
    base = 10;
    dataLen = (int)(strlen(s) + 9)/10;
}

create(0, dataLen);
*(this) = Zero();
while(*s) {
    char tmp[2] = {s[0],0};
    Ipp32u digit = (Ipp32u)strcspn(HexDigitList, tmp);
    *(this) = (*this) * base + BigNumber(digit);
    s++;
}

if(neg)   
    *(this) = Zero() - *(this);

BigNumber::BigNumber(const BigNumber& bn)
{
    IppsBigNumSGN bnSgn;
    int bnBitLen;
    Ipp32u* bnData;
    ippsRef_BN(&bnSgn, &bnBitLen, &bnData, bn);
    create(bnData, BITSIZE_WORD(bnBitLen), bnSgn);
}

// 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
//
BigInt& BigNumber::operator = (const BigNumber& bn)
{
    if(this != &bn) {  // prevent self copy
        IppsBigNumSGN bnSgn;
        int bnBitLen;
        Ipp32u* bnData;
        ippsRef_BN(&bnSgn, &bnBitLen, &bnData, bn);

        delete (Ipp8u*)m_pBN;
        create(bnData, BITSIZE_WORD(bnBitLen), bnSgn);
    }
    return *this;
}

BigInt& BigNumber::operator += (const BigNumber& bn)
{
    int aBitLen;
    ippsRef_BN(NULL, &aBitLen, NULL, *this);
    int bBitLen;
    ippsRef_BN(NULL, &bBitLen, NULL, bn);
    int rBitLen = IPP_MAX(aBitLen, bBitLen) + 1;

    BigNumber result(0, BITSIZE_WORD(rBitLen));
    ippsAdd_BN(*this, bn, result);
    *this = result;
    return *this;
}

BigInt& BigNumber::operator -= (const BigNumber& bn)
{
    int aBitLen;
    ippsRef_BN(NULL, &aBitLen, NULL, *this);
    int bBitLen;
    ippsRef_BN(NULL, &bBitLen, NULL, bn);
    int rBitLen = IPP_MAX(aBitLen, bBitLen);

    BigNumber result(0, BITSIZE_WORD(rBitLen));
    ippsSub_BN(*this, bn, result);
    *this = result;
    return *this;
}

BigInt& BigNumber::operator *= (const BigNumber& bn)
{
    int aBitLen;
    ippsRef_BN(NULL, &aBitLen, NULL, *this);
    int bBitLen;
    ippsRef_BN(NULL, &bBitLen, NULL, bn);
    int rBitLen = aBitLen + bBitLen;

    BigNumber result(0, BITSIZE_WORD(rBitLen));
    ippsMul_BN(*this, bn, result);
BigNumber& BigNumber::operator *= (Ipp32u n)
{
    int aBitLen;
    ippsRef_BN(NULL, &aBitLen, NULL, *this);
    BigNumber result(0, BITSIZE_WORD(aBitLen+32));
    BigNumber bn(n);
    ippsMul_BN(*this, 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);
}
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
{  
    Ipp32u* bnData;
    ippsRef_BN(NULL, NULL, &bnData, *this);
    return bnData[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 = (int)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;
            }
        }
    }
    return msb;
}
Functions for Creation of Cryptographic Contexts

The section presents source code for creation of some cryptographic contexts.
Declarations

Contents of the header file (xsample_cpobjs.h) declaring functions for creation of some cryptographic contexts are presented below:

```c
#ifndef _CPOBJS_H_
#define _CPOBJS_H_

// create new of some ippCP '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);

IppsPrimeState* newPrimeGen(int seedBitsize=160);
void deletePrimeGen(IppsPrimeState* pPrime);

IppsRSAState* newRSA(int lenN, int lenP, IppRSAKeyType 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.

```c
#include "xsample_cpobjs.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;
    ippBigNumGetSize(len, &size);
    IppsBigNumState* pBN = (IppsBigNumState*)( new Ipp8u [size] );
    ippBigNumInit(len, pBN);
    if(pData)
        ippSet_BN(IppsBigNumPOS, len, pData, pBN);
    return pBN;
}
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; }

// Prime Generator context
//
IppsPrimeState* newPrimeGen(int maxBits)
{
    int size;
    ippsPrimeGetSize(maxBits, &size);
    IppsPrimeState* pCtx = (IppsPrimeState*)( new Ipp8u [size] );
    ippsPrimeInit(maxBits, pCtx);
    return pCtx;
}
void deletePrimeGen(IppsPrimeState* pPrimeG)
{ delete [] (Ipp8u*)pPrimeG; }

// 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; }
This appendix contains the table that lists the Cryptography functions removed from Intel IPP 9.0. If an application created with the previous versions calls a function listed here, then the source code must be modified. The table also specifies the corresponding Intel IPP 9.0 functions or workaround to replace the removed functions.

<table>
<thead>
<tr>
<th>Removed from 9.0</th>
<th>Substitution or Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>ippsARCFive128DecryptCBC</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128DecryptCFB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128DecryptCTR</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128DecryptECB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128DecryptOFB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128EncryptCBC</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128EncryptCFB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128EncryptCTR</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128EncryptECB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128EncryptOFB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128GetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128Init</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128Pack</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive128Unpack</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64DecryptCBC</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64DecryptCFB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64DecryptCTR</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64DecryptECB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64DecryptOFB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64EncryptCBC</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64EncryptCFB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64EncryptCTR</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64EncryptECB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64EncryptOFB</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64GetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64Init</td>
<td>N/A</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>ippsARCFive64Pack</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsARCFive64Unpack</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsCMACRijndael128Final</td>
<td>ippsAES_CMACFinal</td>
</tr>
<tr>
<td>ippsCMACRijndael128GetSize</td>
<td>ippsAES_CMACGetSize</td>
</tr>
<tr>
<td>ippsCMACRijndael128Init</td>
<td>ippsAES_CMACInit</td>
</tr>
<tr>
<td>ippsCMACRijndael128MessageDigest</td>
<td>ippsAES_CMACGetTag</td>
</tr>
<tr>
<td>ippsCMACRijndael128Update</td>
<td>ippsAES_CMACUpdate</td>
</tr>
<tr>
<td>ippsCMACSafeRijndael128Init</td>
<td>ippsAES_CMACInit</td>
</tr>
<tr>
<td>ippsDAARijndael128Final</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael128GetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael128Init</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael128MessageDigest</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael128Update</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael192Final</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael192GetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael192Init</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael192MessageDigest</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael192Update</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael256Final</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael256GetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael256Init</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael256MessageDigest</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAARijndael256Update</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAADAEvalRijndael128Init</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAATDESFinal</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAATDESGetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAATDESInit</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAATDESSharedDigest</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsDAATDESSharedUpdate</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBAddPoint</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBCheckPoint</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBComparePoint</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBGenKeyPair</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBGet</td>
<td>N/A</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>ippsECCBGetOrderBitSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBGetPoint</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBGetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBInit</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBMulPointScalar</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBNegativePoint</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBPointGetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBPointInit</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBPublicKey</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSet</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSetKeyPair</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSetPoint</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSetPointAtInfinity</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSetStd</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSharedSecretDH</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSharedSecretDHC</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSignDSA</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBSignNR</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBValidate</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBValidateKeyPair</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBVerifyDSA</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsECCBVerifyNR</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsHMACMD5Duplicate</td>
<td>ippsHMAC_Duplicate</td>
</tr>
<tr>
<td>ippsHMACMD5Final</td>
<td>ippsHMAC_Final</td>
</tr>
<tr>
<td>ippsHMACMD5GetSize</td>
<td>ippsHMAC_GetSize</td>
</tr>
<tr>
<td>ippsHMACMD5GetTag</td>
<td>ippsHMAC_GetTag</td>
</tr>
<tr>
<td>ippsHMACMD5Init</td>
<td>ippsHMAC_Init</td>
</tr>
<tr>
<td>ippsHMACMD5MessageDigest</td>
<td>ippsHMAC_Message</td>
</tr>
<tr>
<td>ippsHMACMD5Pack</td>
<td>ippsHMAC_Pack</td>
</tr>
<tr>
<td>ippsHMACMD5Unpack</td>
<td>ippsHMAC_Unpack</td>
</tr>
<tr>
<td>ippsHMACMD5Update</td>
<td>ippsHMAC_Update</td>
</tr>
<tr>
<td>ippsHMACSHA1Duplicate</td>
<td>ippsHMAC_Duplicate</td>
</tr>
<tr>
<td>ippsHMACSHA1Final</td>
<td>ippsHMAC_Final</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>ippsHMACSHA1GetSize</td>
<td>ippsHMACGetSize</td>
</tr>
<tr>
<td>ippsHMACSHA1GetTag</td>
<td>ippsHMACGetTag</td>
</tr>
<tr>
<td>ippsHMACSHA1Init</td>
<td>ippsHMAC_Init</td>
</tr>
<tr>
<td>ippsHMACSHA1MessageDigest</td>
<td>ippsHMAC_Message</td>
</tr>
<tr>
<td>ippsHMACSHA1Pack</td>
<td>ippsHMAC_Pack</td>
</tr>
<tr>
<td>ippsHMACSHA1Unpack</td>
<td>ippsHMAC_Unpack</td>
</tr>
<tr>
<td>ippsHMACSHA1Update</td>
<td>ippsHMAC_Update</td>
</tr>
<tr>
<td>ippsHMACSHA224Duplicate</td>
<td>ippsHMAC_Duplicate</td>
</tr>
<tr>
<td>ippsHMACSHA224Final</td>
<td>ippsHMAC_Final</td>
</tr>
<tr>
<td>ippsHMACSHA224GetSize</td>
<td>ippsHMACGetSize</td>
</tr>
<tr>
<td>ippsHMACSHA224GetTag</td>
<td>ippsHMACGetTag</td>
</tr>
<tr>
<td>ippsHMACSHA224Init</td>
<td>ippsHMAC_Init</td>
</tr>
<tr>
<td>ippsHMACSHA224MessageDigest</td>
<td>ippsHMAC_Message</td>
</tr>
<tr>
<td>ippsHMACSHA224Pack</td>
<td>ippsHMAC_Pack</td>
</tr>
<tr>
<td>ippsHMACSHA224Unpack</td>
<td>ippsHMAC_Unpack</td>
</tr>
<tr>
<td>ippsHMACSHA224Update</td>
<td>ippsHMAC_Update</td>
</tr>
<tr>
<td>ippsHMACSHA256Duplicate</td>
<td>ippsHMAC_Duplicate</td>
</tr>
<tr>
<td>ippsHMACSHA256Final</td>
<td>ippsHMAC_Final</td>
</tr>
<tr>
<td>ippsHMACSHA256GetSize</td>
<td>ippsHMACGetSize</td>
</tr>
<tr>
<td>ippsHMACSHA256GetTag</td>
<td>ippsHMACGetTag</td>
</tr>
<tr>
<td>ippsHMACSHA256Init</td>
<td>ippsHMAC_Init</td>
</tr>
<tr>
<td>ippsHMACSHA256MessageDigest</td>
<td>ippsHMAC_Message</td>
</tr>
<tr>
<td>ippsHMACSHA256Pack</td>
<td>ippsHMAC_Pack</td>
</tr>
<tr>
<td>ippsHMACSHA256Unpack</td>
<td>ippsHMAC_Unpack</td>
</tr>
<tr>
<td>ippsHMACSHA256Update</td>
<td>ippsHMAC_Update</td>
</tr>
<tr>
<td>ippsHMACSHA384Duplicate</td>
<td>ippsHMAC_Duplicate</td>
</tr>
<tr>
<td>ippsHMACSHA384Final</td>
<td>ippsHMAC_Final</td>
</tr>
<tr>
<td>ippsHMACSHA384GetSize</td>
<td>ippsHMACGetSize</td>
</tr>
<tr>
<td>ippsHMACSHA384GetTag</td>
<td>ippsHMACGetTag</td>
</tr>
<tr>
<td>ippsHMACSHA384Init</td>
<td>ippsHMAC_Init</td>
</tr>
<tr>
<td>ippsHMACSHA384MessageDigest</td>
<td>ippsHMAC_Message</td>
</tr>
<tr>
<td>ippsHMACSHA384Pack</td>
<td>ippsHMAC_Pack</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>ippsHMAC_SHA384Unpack</td>
<td>ippsHMAC_Unpack</td>
</tr>
<tr>
<td>ippsHMAC_SHA384Update</td>
<td>ippsHMAC_Update</td>
</tr>
<tr>
<td>ippsHMAC_SHA512Duplicate</td>
<td>ippsHMAC_Duplicate</td>
</tr>
<tr>
<td>ippsHMAC_SHA512Final</td>
<td>ippsHMAC_Final</td>
</tr>
<tr>
<td>ippsHMAC_SHA512GetSize</td>
<td>ippsHMAC_GetSize</td>
</tr>
<tr>
<td>ippsHMAC_SHA512GetTag</td>
<td>ippsHMAC_GetTag</td>
</tr>
<tr>
<td>ippsHMAC_SHA512Init</td>
<td>ippsHMAC_Init</td>
</tr>
<tr>
<td>ippsHMAC_SHA512MessageDigest</td>
<td>ippsHMAC_Message</td>
</tr>
<tr>
<td>ippsHMAC_SHA512Pack</td>
<td>ippsHMAC_Pack</td>
</tr>
<tr>
<td>ippsHMAC_SHA512Unpack</td>
<td>ippsHMAC_Unpack</td>
</tr>
<tr>
<td>ippsHMAC_SHA512Update</td>
<td>ippsHMAC_Update</td>
</tr>
<tr>
<td>ippsMGF_MD5</td>
<td>ippsMGF</td>
</tr>
<tr>
<td>ippsMGF_SHA1</td>
<td>ippsMGF</td>
</tr>
<tr>
<td>ippsMGF_SHA224</td>
<td>ippsMGF</td>
</tr>
<tr>
<td>ippsMGF_SHA256</td>
<td>ippsMGF</td>
</tr>
<tr>
<td>ippsMGF_SHA384</td>
<td>ippsMGF</td>
</tr>
<tr>
<td>ippsMGF_SHA512</td>
<td>ippsMGF</td>
</tr>
<tr>
<td>ippsRSADecrypt</td>
<td>ippsRSA_Decrypt</td>
</tr>
<tr>
<td>ippsRSAEncrypt</td>
<td>ippsRSA_Encrypt</td>
</tr>
<tr>
<td>ippsRSAGenerate</td>
<td>ippsRSA_GenerateKeys</td>
</tr>
<tr>
<td>ippsRSAGetKey</td>
<td>ippsRSA_GetPublicKey,</td>
</tr>
<tr>
<td></td>
<td>ippsRSA_GetPrivateKeyType1,</td>
</tr>
<tr>
<td></td>
<td>ippsRSA_GetPrivateKeyType2</td>
</tr>
<tr>
<td>ippsRSAGetSize</td>
<td>ippsRSA_GetSizePublicKey</td>
</tr>
<tr>
<td>ippsRSAInit</td>
<td>ippsRSA_InitPublicKey</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt</td>
<td>ippsRSAOAEPDecrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_MD5</td>
<td>ippsRSAOAEPDecrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA1</td>
<td>ippsRSAOAEPDecrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA224</td>
<td>ippsRSAOAEPDecrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA256</td>
<td>ippsRSAOAEPDecrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA384</td>
<td>ippsRSAOAEPDecrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA512</td>
<td>ippsRSAOAEPDecrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPEncrypt</td>
<td>ippsRSAOAEPEncrypt</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_MD5</td>
<td>ippsRSAEncrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA1</td>
<td>ippsRSAEncrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA224</td>
<td>ippsRSAEncrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA256</td>
<td>ippsRSAEncrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA384</td>
<td>ippsRSAEncrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAOAEPDecrypt_SHA512</td>
<td>ippsRSAEncrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSAAsPack</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsRSASSASign</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_MD5</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_MD5_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA1</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA1_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA224</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA224_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA256</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA256_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA384</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA384_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSASign_SHA512</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_MD5</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_MD5_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA1</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA1_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA224</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA224_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA256</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA256_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA384</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA384_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA512</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA512_PKCSv15</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA512</td>
<td>ippsRSAEncrypt_PSS</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
<td>----------------------------------------------------------------</td>
</tr>
<tr>
<td>ippsRSASSAVerify_SHA512_PKCSv15</td>
<td>ippsRSAVerify_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSASSA_PKCS1v15_Sign</td>
<td>ippsRSASign_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSASSA_PKCS1v15_Verify</td>
<td>ippsRSAVerify_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSASSA_PSS_Sign</td>
<td>ippsRSASign_PSS</td>
</tr>
<tr>
<td>ippsRSASSA_PSS_Verify</td>
<td>ippsRSAVerify_PSS</td>
</tr>
<tr>
<td>ippsRSASetKey</td>
<td>ippsRSA_SetPrivateKeyType1, ippsRSA_SetPrivateKeyType2</td>
</tr>
<tr>
<td>ippsRSAUnpack</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsRSAValidate</td>
<td>ippsRSAValidateKeys</td>
</tr>
<tr>
<td>ippsRSA_Decrypt_PKCSv15</td>
<td>ippsRSA_decrypt_PKCSv15</td>
</tr>
<tr>
<td>ippsRSA_Encrypt_PKCSv15</td>
<td>ippsRSA_encrypt_PKCSv15</td>
</tr>
<tr>
<td>ippsRSA_OAEPDecrypt</td>
<td>ippsRSAdecrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPDecrypt_MD5</td>
<td>ippsRSA_decrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPDecrypt_SHA1</td>
<td>ippsRSA_decrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPDecrypt_SHA224</td>
<td>ippsRSA_decrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPDecrypt_SHA256</td>
<td>ippsRSA_decrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPDecrypt_SHA384</td>
<td>ippsRSA_decrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPDecrypt_SHA512</td>
<td>ippsRSA_decrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPEncrypt</td>
<td>ippsRSAEncrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPEncrypt_MD5</td>
<td>ippsRSA_encrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPEncrypt_SHA1</td>
<td>ippsRSA_encrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPEncrypt_SHA224</td>
<td>ippsRSA_encrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPEncrypt_SHA256</td>
<td>ippsRSA_encrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPEncrypt_SHA384</td>
<td>ippsRSA_encrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_OAEPEncrypt_SHA512</td>
<td>ippsRSA_encrypt_OAEP</td>
</tr>
<tr>
<td>ippsRSA_SSASign</td>
<td>ippsRSASign_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSASign_MD5</td>
<td>ippsRSASign_MD5</td>
</tr>
<tr>
<td>ippsRSA_SSASign_MD5_PKCSv15</td>
<td>ippsRSASign_PKSv15</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA1</td>
<td>ippsRSASign_SHA1</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA1_PKCSv15</td>
<td>ippsRSASign_SHA1</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA224</td>
<td>ippsRSASign_SHA224</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA224_PKCSv15</td>
<td>ippsRSASign_PKCSv15</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA256</td>
<td>ippsRSASign_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA256_PKCSv15</td>
<td>ippsRSASign_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA384</td>
<td>ippsRSASign_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA384_PKCSv15</td>
<td>ippsRSASign_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA512</td>
<td>ippsRSASign_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSASign_SHA512_PKCSv15</td>
<td>ippsRSASign_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify</td>
<td>ippsRSAVerify_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_MD5</td>
<td>ippsRSAVerify_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_MD5_PKCSv15</td>
<td>ippsRSAVerify_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA1</td>
<td>ippsRSAVerify_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA1_PKCSv15</td>
<td>ippsRSAVerify_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA224</td>
<td>ippsRSAVerify_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA224_PKCSv15</td>
<td>ippsRSAVerify_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA256</td>
<td>ippsRSAVerify_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA256_PKCSv15</td>
<td>ippsRSAVerify_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA384</td>
<td>ippsRSAVerify_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA384_PKCSv15</td>
<td>ippsRSAVerify_PKCS1v15</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA512</td>
<td>ippsRSAVerify_PSS</td>
</tr>
<tr>
<td>ippsRSA_SSAVerify_SHA512_PKCSv15</td>
<td>ippsRSAVerify_PKCS1v15</td>
</tr>
<tr>
<td>ippsRijndael128CCMDecrypt</td>
<td>ippsAES_CCMDecrypt</td>
</tr>
<tr>
<td>ippsRijndael128CCMDecryptMessage</td>
<td>ippsAES_CCMDecrypt</td>
</tr>
<tr>
<td>ippsRijndael128CCMEncrypt</td>
<td>ippsAES_CCMEncrypt</td>
</tr>
<tr>
<td>ippsRijndael128CCMEncryptMessage</td>
<td>ippsAES_CCMEncrypt</td>
</tr>
<tr>
<td>ippsRijndael128CCMGetSize</td>
<td>ippsAES_CCMGetSize</td>
</tr>
<tr>
<td>ippsRijndael128CCMGetTag</td>
<td>ippsAES_CCMGetTag</td>
</tr>
<tr>
<td>ippsRijndael128CCMInit</td>
<td>ippsAES_CCMInit</td>
</tr>
<tr>
<td>ippsRijndael128CCMMessageLen</td>
<td>ippsAES_CCMMessageLen</td>
</tr>
<tr>
<td>ippsRijndael128CCMStart</td>
<td>ippsAES_CCMStart</td>
</tr>
<tr>
<td>ippsRijndael128CCMTagLen</td>
<td>ippsAES_CCMTagLen</td>
</tr>
<tr>
<td>ippsRijndael128DecryptCBC</td>
<td>ippsAESDecryptCBC</td>
</tr>
<tr>
<td>ippsRijndael128DecryptCFB</td>
<td>ippsAESDecryptCFB</td>
</tr>
<tr>
<td>ippsRijndael128DecryptCTR</td>
<td>ippsAESDecryptCTR</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>ippsRijndael128DecryptECB</td>
<td>ippsAESDecryptECB</td>
</tr>
<tr>
<td>ippsRijndael128DecryptOFB</td>
<td>ippsAESDecryptOFB</td>
</tr>
<tr>
<td>ippsRijndael128EncryptCBC</td>
<td>ippsAESEncryptCBC</td>
</tr>
<tr>
<td>ippsRijndael128EncryptCFB</td>
<td>ippsAESEncryptCFB</td>
</tr>
<tr>
<td>ippsRijndael128EncryptCTR</td>
<td>ippsAESEncryptCTR</td>
</tr>
<tr>
<td>ippsRijndael128EncryptECB</td>
<td>ippsAESEncryptECB</td>
</tr>
<tr>
<td>ippsRijndael128EncryptOFB</td>
<td>ippsAESEncryptOFB</td>
</tr>
<tr>
<td>ippsRijndael128GCMDecrypt</td>
<td>ippsAES_GCMDecrypt</td>
</tr>
<tr>
<td>ippsRijndael128GCMEncrypt</td>
<td>ippsAES_GCMEncrypt</td>
</tr>
<tr>
<td>ippsRijndael128GCMGetSizeManaged</td>
<td>ippsAES_GCMGetSize</td>
</tr>
<tr>
<td>ippsRijndael128GCMGetTag</td>
<td>ippsAES_GCMGetTag</td>
</tr>
<tr>
<td>ippsRijndael128GCMInitManaged</td>
<td>ippsAES_GCMInit</td>
</tr>
<tr>
<td>ippsRijndael128GCMProcessAAD</td>
<td>ippsAES_GCMProcessAAD</td>
</tr>
<tr>
<td>ippsRijndael128GCMProcessIV</td>
<td>ippsAES_GCMProcessIV</td>
</tr>
<tr>
<td>ippsRijndael128GCMReset</td>
<td>ippsAES_GCMReset</td>
</tr>
<tr>
<td>ippsRijndael128GCMStart</td>
<td>ippsAES_GCMStart</td>
</tr>
<tr>
<td>ippsRijndael128GetSize</td>
<td>ippsAESGetSize</td>
</tr>
<tr>
<td>ippsRijndael128Init</td>
<td>ippsAESInit</td>
</tr>
<tr>
<td>ippsRijndael128Pack</td>
<td>ippsAESPack</td>
</tr>
<tr>
<td>ippsRijndael128SetKey</td>
<td>ippsAESSetKey</td>
</tr>
<tr>
<td>ippsRijndael128Unpack</td>
<td>ippsAESUnpack</td>
</tr>
<tr>
<td>ippsRijndael192DecryptCBC</td>
<td>ippsAESDecryptCBC</td>
</tr>
<tr>
<td>ippsRijndael192DecryptCFB</td>
<td>ippsAESDecryptCFB</td>
</tr>
<tr>
<td>ippsRijndael192DecryptCTR</td>
<td>ippsAESDecryptCTR</td>
</tr>
<tr>
<td>ippsRijndael192DecryptECB</td>
<td>ippsAESDecryptECB</td>
</tr>
<tr>
<td>ippsRijndael192DecryptOFB</td>
<td>ippsAESDecryptOFB</td>
</tr>
<tr>
<td>ippsRijndael192EncryptCBC</td>
<td>ippsAESEncryptCBC</td>
</tr>
<tr>
<td>ippsRijndael192EncryptCFB</td>
<td>ippsAESEncryptCFB</td>
</tr>
<tr>
<td>ippsRijndael192EncryptCTR</td>
<td>ippsAESEncryptCTR</td>
</tr>
<tr>
<td>ippsRijndael192EncryptECB</td>
<td>ippsAESEncryptECB</td>
</tr>
<tr>
<td>ippsRijndael192EncryptOFB</td>
<td>ippsAESEncryptOFB</td>
</tr>
<tr>
<td>ippsRijndael192GetSize</td>
<td>ippsAESGetSize</td>
</tr>
<tr>
<td>Removed from 9.0</td>
<td>Substitution or Workaround</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>ippsRijndael192Init</td>
<td>ippsAESInit</td>
</tr>
<tr>
<td>ippsRijndael192Pack</td>
<td>ippsAESPack</td>
</tr>
<tr>
<td>ippsRijndael192Unpack</td>
<td>ippsAESUnpack</td>
</tr>
<tr>
<td>ippsRijndael256DecryptCBC</td>
<td>ippsAESDecryptCBC</td>
</tr>
<tr>
<td>ippsRijndael256DecryptCFB</td>
<td>ippsAESDecryptOFB</td>
</tr>
<tr>
<td>ippsRijndael256DecryptCTR</td>
<td>ippsAESDecryptCTR</td>
</tr>
<tr>
<td>ippsRijndael256DecryptECB</td>
<td>ippsAESDecryptECB</td>
</tr>
<tr>
<td>ippsRijndael256DecryptOFB</td>
<td>ippsAESDecryptCFB</td>
</tr>
<tr>
<td>ippsRijndael256EncryptCBC</td>
<td>ippsAESEncryptCBC</td>
</tr>
<tr>
<td>ippsRijndael256EncryptCFB</td>
<td>ippsAESEncryptOFB</td>
</tr>
<tr>
<td>ippsRijndael256EncryptCTR</td>
<td>ippsAESEncryptCTR</td>
</tr>
<tr>
<td>ippsRijndael256EncryptECB</td>
<td>ippsAESEncryptECB</td>
</tr>
<tr>
<td>ippsRijndael256EncryptOFB</td>
<td>ippsAESEncryptCFB</td>
</tr>
<tr>
<td>ippsRijndael256GetSize</td>
<td>ippsAESGetSize</td>
</tr>
<tr>
<td>ippsRijndael256Init</td>
<td>ippsAESInit</td>
</tr>
<tr>
<td>ippsRijndael256Pack</td>
<td>ippsAESPack</td>
</tr>
<tr>
<td>ippsRijndael256Unpack</td>
<td>ippsAESUnpack</td>
</tr>
<tr>
<td>ippsSafeRijndael128Init</td>
<td>ippsAESInit</td>
</tr>
<tr>
<td>ippsXCBCRijndael128Final</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsXCBCRijndael128GetSize</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsXCBCRijndael128GetTag</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsXCBCRijndael128Init</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsXCBCRijndael128MessageTag</td>
<td>N/A</td>
</tr>
<tr>
<td>ippsXCBCRijndael128Update</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Bibliography

This bibliography provides a list of publications that might be helpful to you in using cryptography functions of Intel IPP.


C


Index

A

AES-CCM Functions
AES_CCMDecrypt42
AES_CCMEncrypt41
AES_CCMGetSize39
AES_CCMGetTag43
AES_CCMInit40
AES_CCMMessageLen43
AES_CCMStart41
AES_CCMTagLen44

AES-GCM Functions
AES_GCMDecrypt51
AES_GCMEncrypt50
AES_GCMGetSize46
AES_GCMGetTag51
AES_GCMInit46
AES_GCMProcessAAD49
AES_GCMProcessIV49
AES_GCMReset48
AES_GCMStart47

ARCFour Functions
ARCFourCheckKey78
ARCFourDecrypt81
ARCFourEncrypt80
ARCFourGetSize78
ARCFourInit79
ARCFourPack79
ARCFourReset82
ARCFourUnpack79

ARCFour stream cipher77

B

Big Number Arithmetic145
Big Number Arithmetic Functions
Add_BN155
BigNumGetSize145
BigNumInit146
Cmp_BN153
CmpZero_BN154
Div_BN159
ExtGet_BN151
Gcd_BN161
Get_BN150
GetOctString_BN152
GetSize_BN149
MAC_BN1158
Mod_BN160
ModInv_BN161
Mul_BN157
Ref_BN151
Set_BN147
SetOctString_BN148
Sub_BN156

C

CMAC133
CMAC Functions
AES_CMACFinal142
AES_CMACGetSize140
AES_CMACInit140

D

Discrete Logarithm Based Functions
DLGetResultString231
DLPGenerateDH228
DLPGenerateDSA222
DLPGenKeyPair219
DLPGet216
DLPGetDP218
DLPGetSize213
DLPInit214
DLPack215
DLPublicKey220
DLPSet215
DLPSetDP217
DLPSetKeyPair221
DLPSharedSecretDH230
DLPSignDSA224
DLPUnpack215
DLPValidateDH229
DLPValidateDSA223
DLPValidateKeyPair220
DLPValidateKeyPair220
DLPVerifyDSA225
ECCGetResultString264

E

Elliptic Curve Cryptographic Functions
ECCPAddPoint247
ECCPCheckPoint245
ECCPComparePoint246
ECCPGenKeyPair249
ECCPGet238
ECCPGetOrderBitSize239
ECCPGetPoint244
ECCPGetSize233
ECCPGetSizeStd233
ECPInit234
ECPInitStd235
ECPMullPointScalar248
ECCPNegativePoint247
ECCPPoint241
ECCPPointInit242
ECCPPublicKey250
ECPSet236
ECPSetKeyPair252
ECPSetPoint243
ECPSetPointAtInfinity243
ECPSetStd237
ECCPSharedSecretDH253
ECCPSharedSecretDHC254
ECCPSignDSA255
ECCPSignNR257
ECCPSignSM2260
ECPValidate240
ECPValidateKeyPair251
ECPVerifyDSA256
ECPVerifyNR259
ECCPVerifySM2261
encryption, decryption, and encryption (E-D-E) sequence52
Enter index keyword

F
font conventions19

G
GetLibVersion267

H
Hash and SHA Algorithms
HashDuplicate86
HashFinal88
HashGetSize84
HashGetTag88
HashInit85
HashPack85
HashUnpack85
HashUpdate87
hash function83
Hash Functions for Non-Streaming Messages
general definition122
HashMessage122
MD5MessageDigest124
SHA1MessageDigest125
SHA224MessageDigest127
SHA256MessageDigest127
SHA384MessageDigest128
SHA512MessageDigest129
SM3MessageDigest123
user-implemented121
HMAC133
HMAC Functions133

I
initialization vector iv52
Intel Performance Library Suite23
ipcppGetLibVersion267
ippsAdd_BN155
ippsAES_CCMDecrypt42
ippsAES_CCMEncrypt41
ippsAES_CBCGetSize39
ippsAES_CCMGetTag43
ippsAES_CCMInit40
ippsAESCmessageLen43
ippsAES_CCMStart41
ippsAES_CMPFinal142
ippsAES_CMAGetSize140
ippsAES_CMAGetTag143
ippsAES_CMACInit140
ippsAES_GCMUpdate141
ippsAES_GCMDecrypt51
ippsAES_GCMEncrypt50
ippsAES_GCMGetSize46
ippsAES_GCMGetTag51
ippsAES_GCMInit46
ippsAES_GCMProcessAAD49
ippsAES_GCMProcessIV49
ippsAES_GCMReset48
ippsAES_GCMStart47
ippsAESDecryptCBC32
ippsAESDecryptCFB33
ippsAESDecryptCTR37
ippsAESDecryptECB30
ippsAESDecryptOFB35
ippsAESEncryptCBC31
ippsAESEncryptCFB32
ippsAESEncryptCTR36
ippsAESEncryptECB29
ippsAESEncryptOFB34
ippsAESGetSize26
ippsAESInit27
ippsAESPack28
ippsAESSetKey28
ippsAESUnpack28
ippsARCFourCheckKey78
ippsARCFourDecrypt81
ippsARCFourEncrypt80
ippsARCFourGetSize78
ippsARCFourInit79
ippsARCFourPack79
ippsARCFourReset82
ippsARCFourUnpack79
ippsBigNumGetSize145
ippsBigNumInit146
ippsCmp_BN153
ippsCmpZero_BN154
ippsDESGetSize53
ippsDESInit54
ippsDESPack54
ippsDESUnpack54
ippsDiv_BN159
ippsDLGetResultString231
ippsDLPGenerateDH228
ippsDLPGenerateDSA222
ippsDLPGenKeyPair219
ippsDLPGet216
ippsDLPGetDP218
ippsDLPGetSize213
ippsDLPInit214
ippsDLPack215
ippsDLPublicKey220
ippsDLPset215
ippsDLPsetDP217
ippsDPSignRSA219
ippsDLPSharedSecretDH230
ippsDLPSignDSA224
ippsDLPUnpack215
ippsDLPValidateDH229
ippsDLPValidateDSA223
ippsDLPValidateKeyPair220
ippsDLPVerifyDSA225
ippsECCGetResultString264
ippsECCPAddPoint247
ippsECCPCheckPoint245
ippsECCPComparePoint246
ippsECCPGenKeyPair249
ippsECCPGet238
ippsECCPGetOrderBitSize239
ippsECCPGetPoint244
ippsECCPGetSize233
ippsECCPGetSizeStd233
ippsECCPInit234
ippsECCPInitStd235
ippsECCPMulPointScalar248
ippsECCPNegatePoint247
ippsECCPPointGetSize241
ippsECCPPointInit242
ippsECCPSetKey250
ippsECCPSet236
ippsECCPSetKeyPair252

keyed hash functions

hmac_duplicate
hmac_final
hmac_getsize
hmac_gettag
hmac_init
hmac_message
hmac_pack
hmac_unpack
hmac_update

message authentication functions

cmac functions

keyed hash functions

modes of operation for block ciphers

cbc

ccm

ctf

ecb

ofb

montgomery reduction scheme functions

montinit

montset

montexp

montform

montget

montgetsize

montmul

naming conventions

notational conventions
P
PKCS V1.5 Encryption Scheme Functions
PKCS V1.5 Signature Scheme Functions
Prime Number Generation Functions
PrimeGen182
PrimeGen_BN181
PrimeGetSize179
PrimeInit180
PrimeGet185
PrimeGet_BN186
PrimeSet184
PrimeSet_BN185
PrimeTest183
PrimeTest_BN182
Pseudorandom Number Generation Functions
PRNGen176
PRNGen_BN177
PRNGGetSize171
PRNGInit172
PRNGSetAugment174
PRNGSetH0176
PRNGSetModulus175
PRNGSetSeed173
user-implemented171

R
RC4 stream cipher77
reference code23
Rijndael Functions
AES-CCM Functions39
AES-GCM Functions45
AESDecryptCBC32
AESDecryptCFB33
AESDecryptCTR37
AESDecryptECB30
AESDecryptOFB35
AESEncryptCBC31
AESEncryptCFB32
AESEncryptCTR36
AESEncryptECB29
AESEncryptOFB34
AESGetSize26
AESInit27
AESPack28
AESSetKey28
AESUnpack28
SMS4DecryptCBC71
SMS4DecryptCFB73
SMS4DecryptCTR76
SMS4DecryptECB69
SMS4DecryptOFB74
SMS4EncryptCBC70
SMS4EncryptCFB72
SMS4EncryptCTR75
SMS4EncryptECB68
SMS4EncryptOFB73
SMS4GetSize66
SMS4Init67
SMS4SetKey68
RSA Algorithm Functions188
RSA Primitives
RSA_Decrypt199
RSA_Encrypt198
RSA System Building Functions
RSA_GenerateKeys194
RSA_GetBufferSizePrivateKey193
RSA_GetBufferSizePublicKey193
RSA_GetPrivateKeyType1192
RSA_GetPublicKey192
RSAGetSizePrivateKeyType1188
RSAGetSizePublicKey188
RSA_InitPrivateKeyType1190
RSA_InitPublicKey190
RSA_SetPrivateKeyType1191
RSA_SetPublicKey191
RSA.ValidateKeys196
RSA-based Encryption Scheme Functions
RSAEncrypt_OAEP204
RSAEncrypt_PKCSv15206
RSAEncrypt_OAEP203
RSAEncrypt_PKCSv15205
RSA-based scheme188
RSA-based Signature Scheme Functions
RSAEncrypt_PKCSv15210
RSAEncrypt_PSS208
RSAEncrypt_PKCSv15211
RSAEncrypt_PSS209
RSA-OAEP Scheme Functions202
RSA-Based-PSS Scheme Functions207

S
secret data23, 24
SM3 and SHA Algorithms
SM3Duplicate91
SM3Final92
SM3GetSize89
SM3GetTag93
SM3Init89
SM3Pack90
SM3Unpack90
SM3Update91
SMS4 Functions66
support functions and classes
symmetric algorithm modes
Cipher Block Chain (CBC)25
Cipher Feedback (CFB)25
Counter (CTR)25
Counter with Cipher Block Chaining-Message
Authentication Code (CCM)39
Electronic Code Book (ECB)25
Output Feedback (OFB)25

T
TDES Functions
DESGetSize53
DESInit54
DESExpand54
DESVpack54
TDESDecryptCBC58
TDESDecryptCFB60
TDESDecryptCTR64
TDESDecryptECB65
TDESDecryptOFB62
TDESEncryptCBC57
TDESEncryptCFB59
TDESEncryptCTR63
TDESEncryptECB55
TDESEncryptOFB61
Triple Data Encryption Standard (TDES)52
version information function267