Intel® Integrated Performance Primitives Cryptography

Developer Reference

Intel Integrated Performance Primitives 2018

Legal Information

Contents

Chapter 3: One-Way Hash Primitives

Chapter 6: Finite Field Arithmetic

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. Current characterized errata are available on request.

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

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.

Third Party

Intel® Integrated Performance Primitives (Intel® IPP) includes content from several 3rd party sources that was originally governed by the licenses referenced below:

- zlib library:

  zlib.h -- interface of the 'zlib' general purpose compression library version 1.2.8, April 28th, 2013

  Copyright (C) 1995-2013 Jean-loup Gailly and Mark Adler

  This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

  Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
  2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
  3. This notice may not be removed or altered from any source distribution.

  Jean-loup Gailly Mark Adler

  jloup@gzip.org madler@alumni.caltech.edu
- bzip2:

  Copyright © 1996 - 2015 julian@bzip.org

Getting Help and Support

Getting Help

You can get context-sensitive help for the Intel® Integrated Performance Primitives (Intel® IPP) Cryptography 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/ and the Intel IPP forum http://software.intel.com/en-us/forums/intel-integrated-performance-primitives/.

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

Introducing Intel® Integrated Performance Primitives Cryptography

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 and image processing and cryptography.

- The Intel IPP signal and data 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, string processing, and vector math.
- The Intel IPP image 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.

Intel IPP Cryptography 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.

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

What's New

This Developer Reference documents Intel® Integrated Performance Primitives (Intel® IPP) Cryptography 2018.

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

Intel® IPP Cryptography 2018 (Beta)

- The arithmetic of the group of elliptic curve points functionality has been extended: standard elliptic curves, key generation functions, and functions for computing digital signatures have been added. (For more details, see Arithmetic of the Group of Elliptic Curve Points).
- New finite field arithmetic functions have been added, and several existing `Method` functions have been renamed. (For more details, see Finite Field Arithmetic).
- Support functions have been added to help make Intel IPP Cryptography usable in the absence of the main Intel IPP package. (For more details, see Other Functions).
- *Domain Dependencies* blocks were removed throughout the document because Intel IPP Cryptography functions no longer depend on other Intel IPP components.

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

Notational Conventions

The code and syntax used in this document for function and variable declarations are written in the ANSI C style. However, versions of Intel IPP for different processors or operating systems may, of necessity, vary slightly.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

In this document, 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 document:

`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 document, 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

1

This document describes the structure, operation, and functions of Intel® Integrated Performance Primitives (Intel® IPP) Cryptography. The document provides a background for cryptography concepts used in the Intel IPP Cryptography software as well as detailed description of the respective Intel IPP Cryptography functions. The Intel IPP Cryptography 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.

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

2

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

Intel® Integrated Performance Primitives (Intel® IPP) Cryptography offers 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

Block Cipher Name	Data Block Size (bits)
Rijndael128 (AES)	128
TDES	64

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,

```
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`

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`

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

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

Include Files

`ippcp.h`

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

```
IppStatus ippsAESPack (const IppsAESSpec* pCtx, Ipp8u* pBuffer, int bufSize);
IppStatus ippsAESUnpack (const Ipp8u* pBuffer, IppsAESSpec* pCtx, int ctxSize);
```

Include Files

`ippcp.h`

Parameters

pCtx	Pointer to the `IppsAESSpec` context.
pBuffer	Pointer to the user-defined buffer.
bufSize	Available size of the buffer.
ctxSize	Available size of the context.

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

`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 *bufSize* or *ctxSize* is less than the real size of the `IppsAESSpec` context.

AESEncryptECB

Encrypts plaintext message by using ECB encryption mode.

Syntax

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

Include Files

`ippcp.h`

Parameters

pSrc	Pointer to the input plaintext data stream of variable length.
pDst	Pointer to the resulting ciphertext data stream.
srclen	Length of the input plaintext data in bytes.
pCtx	Pointer to the `IppsAESSpec` 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 *srclen* is not divisible by cipher block size.

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

AESDecryptECB

Decrypts 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`

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`

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

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`

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`

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`

Parameters

pSrc	Pointer to the input ciphertext data stream.
pDst	Pointer to the resulting plaintext data stream of variable length.
srclen	Length of the ciphertext 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 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 *srcLen* is not divisible by cipher block size.

AESEncryptOFB

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

Syntax

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

Include Files

`ippcp.h`

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

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

Include Files

`ippcp.h`

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`

Parameters

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

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.
`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.

AESEncryptXTS_Direct, AESDecryptXTS_Direct

Encrypts/decrypts a data buffer in the XTS mode.

Syntax

```
IppStatus ippsAESEncryptXTS_Direct(const Ipp8u* pSrc, Ipp8u* pDst, int dataBitLen, int
startBlockNo, const Ipp8u* pTweak, const Ipp8u* pKey, int keyBitLen, int
dataUnitBitLen);
```

```
IppStatus ippsAESDecryptXTS_Direct(const Ipp8u* pSrc, Ipp8u* pDst, int dataBitLen, int
startBlockNo, const Ipp8u* pTweak, const Ipp8u* pKey, int keyBitLen, int
dataUnitBitLen);
```

Include Files

`ippcp.h`

Parameters

pSrc	Pointer to the input (plain- or cipher-text) data buffer.
pDst	Pointer to the output (cipher- or plain-text) data buffer.
dataBitLen	Length of the input data being encrypted or decrypted, in bits. The output data length is equal to the input data length.
startBlockNo	The sequential number of the first plain- or cipher-text block for operation inside the data unit.
pTweak	Pointer to the little-endian 16-byte array that contains the tweak value assigned to the data unit being encrypted/decrypted.
pKey	Pointer to the XTS-AES key.
keyBitLen	Length of the XTS-AES key, in bits.
dataUnitBitLen	Length of the data unit, in bits.

Description

These functions encrypt or decrypt the input data according to the XTS-AES mode [IEEE P1619] of the AES block cipher. The XTS-AES tweakable block cipher can be used for encryption/decryption of sector-based storage. The XTS-AES algorithm acts on a single **data unit** or a section within the data unit and uses AES as the internal cipher. The length of the data unit must be 128 bits or more. The data unit is considered as partitioned into `m+1` blocks:

`T = T[0] | T[1] | … |T[m-2] | T[m-1] | T[m]`

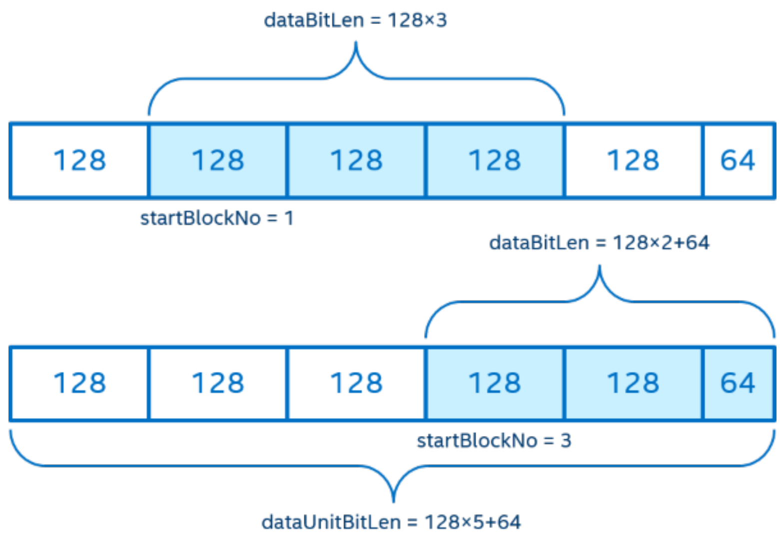
where

- `m = ceil(`*dataUnitBitLen*`/128)`
- the first `m` blocks `T[0], T[1], …,T[m-1]` are exactly 128 bits long
- the last block `T[m]` is between 0 and 127 bits long (it could be empty, for example, 0 bits long)

The cipher processes the first `(m-1)` blocks `T[0], T[1], …, T[m-2]` independently of each other. If the last block `T[m]` *is empty*, then the block `T[m-1]` is processed independently too. However, if the last block `T[m]` *is not empty,* then the cipher processes the blocks `T[m-1]` and `T[m]` together using a *ciphertext stealing* mechanism. See [IEEE P1619] for details.

With the Intel IPP implementation of XTS-AES, you can select a sequence of adjacent data blocks (section) within the data unit for processing. The section you select is specified by the *startBlockNo* and *dataBitLen* parameters.

The ciphertext stealing mechanism constrains possible section selections. If the last block `T[m]` of the data unit is not empty, the section you select must contain either both `T[m-1]` and `T[m]` or neither of them. Therefore, consider *dataBitLen*, *startBlockNo*, and *dataUnitBitLen* all together when making a function call. The following figure shows valid selections of a section within the data unit:



The XTS-AES block cipher uses tweak values to ensure that each data unit is processed differently. A tweak value is a 128-bit integer that represents the logical position of the data unit. The tweak values are assigned to the data units consecutively, starting from an arbitrary non-negative integer. Before calling the function, convert the tweak value into a 16-byte little-endian array. For example, the tweak value `0x123456789A` corresponds to the byte array

```
Ipp8u twkArray[16] = {0x9A, 0x78, 0x56, 0x34, 0x12, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}.
```

The key for XTS-AES is parsed as a concatenation of two fields of equal size, called `data key` and `tweak key`, so that `key = data key|tweak key`.

where

- `data key` is used for data encryption/decryption
- `tweak key` is used for encryption of the tweak value

The standard allows only AES128 and AES256 keys.

Refer to [IEEE P1619] for more details.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error when any of the specified pointers is `NULL`.
`ippsStsLengthErr`	Indicates an error condition when: • *dataBitLen* `< 128` • *keyBitLen* `!= 256` and *keyBitLen* `!= 512` • *dataUnitBitLen* `< 128`
`ippsStsLengthErr`	Indicates an error when:

- *startBlockNo* < 0
- *startBlockNo* >= *dataUnitBitLen*/128
- There is any other inconsistency with the assumed data unit partition

Example of Using AES Functions

AES Encryption and Decryption

```
    // 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(ctext)];
   // decryption
   ippsAESDecryptCTR(ctext, rtext, sizeof(ctext), 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

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

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`

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

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

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.
`ippStsLengthErr`	Indicates an error condition if *len* is less than zero or the value that accumulates *len* parameters from previous calls to `AES_CCMEncrypt` with the current value of *len* exceeds the tag length specified in the previous call to AES_CCMMessageLen.

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`

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.
`ippStsLengthErr`	Indicates an error condition if *len* is less than zero or the value that accumulates *len* parameters from previous calls to `AES_CCMDecrypt` with the current value of *len* exceeds the tag length specified in the previous call to AES_CCMMessageLen.

AES_CCMGetTag

Generates the message authentication tag in the CCM mode.

Syntax

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

Include Files

`ippcp.h`

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* is less than one or *tagLen* exceeds the tag length specified in the previous call to AES_CCMTagLen.

AES_CCMMessageLen

Sets up the length of the message to be processed.

Syntax

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

Include Files

`ippcp.h`

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

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

Include Files

ippcp.h

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

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

Include Files

```
ippcp.h
```

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

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

Include Files

`ippcp.h`

Parameters

pKey	Pointer to the secret key.
keyLen	Length of the secret key.
pState	Pointer to the buffer being initialized as `IppsAES_GCMState` context.
ctxSize	Available size of the buffer.

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.

> **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_GCMStart

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

Syntax

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

Include Files

ippcp.h

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_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

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

Include Files

ippcp.h

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

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 *ivLen* bytes of the initial vector *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`

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

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

Include Files

`ippcp.h`

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.
`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_GCMDecrypt

Decrypts a data buffer in the GCM mode.

Syntax

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

Include Files

`ippcp.h`

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

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

Include Files

`ippcp.h`

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.

AES-SIV Functions

This section describes functions for the Synthetic Initialization Vector (SIV) authenticated encryption using the AES cipher [RFC5297].

AES_S2V_CMAC

Produces the synthetic initialization vector.

Syntax

```
IppStatus ippsAES_S2V_CMAC(const Ipp8u* pKey, int keyLen, const Ipp8u* AD[], const int
ADlen[], int numAD, Ipp8u* pSIV);
```

Include Files

`ippcp.h`

Parameters

pKey	Pointer to the key.
keyLen	Length of the key in bytes.
AD	Array of pointers to individual input strings.
ADlen	Array of length (in bytes) of the individual input strings.
numAD	The number of the strings.
pSIV	Pointer to the output 16-byte vector.

Description

The `AES_S2V_CMAC` function takes a key and maps the vector of individual strings *AD*[0], *AD*[1], ..., *AD*[*numAD*-1] to the 16-byte output vector.

The function uses pseudorandom AES_CMAC functions to process each input string, as well as doubling and xoring operations to map the output to a single output vector.

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` or a pointer *AD*[*i*] to any individual string is `NULL` while the length *ADlen*[*i*] is non-zero.
`ippStsLengthErr`	Indicates an error condition that occurs because of one of the following: • The *keyLen* parameter is different from 16, 24, and 32 • The number of the strings *numAD* in the *AD* array is negative • The length *ADlen*[*i*] of any individual input string is negative

AES_SIVEncrypt

Performs the SIV authenticated encryption using the AES cipher.

Syntax

```
IppStatus ippsAES_SIVEncrypt(const Ipp8u* pSrc, Ipp8u* pDst, int len, Ipp8u* pSIV,
const Ipp8u* pAuthKey, const Ipp8u* pConfKey, int keyLen, const Ipp8u* AD[], const int
ADlen[], int numAD);
```

Include Files

`ippcp.h`

Parameters

pSrc	Pointer to the input data to encrypt (plaintext).
pDst	Pointer to the output encrypted data (ciphertext).
len	Length in bytes of the plaintext and ciphertext.
pSIV	Pointer to the output synthetic initialization vector.
pAuthKey	Pointer to the authentication key.
pConfKey	Pointer to the confidentiality key.
keyLen	Length of keys in bytes.
AD	Array of pointers to the associated input strings.
ADlen	Array of length (in bytes) of the associated input strings.
numAD	The number of the associated strings.

Description

The `AES_SIVEncrypt` function accepts authentication and confidentiality keys of length *keyLen* each, plaintext (**pSrc*) of an arbitrary length *len*, and a vector *AD*[] of associated data (strings). The output of the function is the 16-byte synthetic initialization vector (**pSIV*) and encrypted data (**pDst*) of the same length as the plaintext.

The computation includes the following steps:

1. Compute a synthetic initialization vector by passing the plaintext, *pAuthKey* key, and *AD*[] to AES_S2V_CMAC.
2. Encrypt the plaintext using the AES cipher in the CTR mode with the initial counter value (*CTR0*) equal to the synthetic initialization vector xored with a fixed mask.

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 or a pointer *AD*[*i*] to any individual string is NULL while the length *ADlen*[*i*] is non-zero.
ippStsLengthErr	Indicates an error condition that occurs because of one of the following: • The *keyLen* parameter is different from 16, 24, and 32 • The number of the strings *numAD* in the *AD* array is negative or greater than 127 • The length *ADlen*[*i*] of any individual input string is negative • The *len* parameter is negative

AES_SIVDecrypt

Performs the SIV authenticated decryption using the AES cipher.

Syntax

```
IppStatus ippsAES_SIVDecrypt(const Ipp8u* pSrc, Ipp8u* pDst, int len, int* pAuthPassed,
const Ipp8u* pAuthKey, const Ipp8u* pConfKey, int keyLen, const Ipp8u* AD[], const int
ADlen[], int numAD, const Ipp8u* pSIV);
```

Include Files

ippcp.h

Parameters

pSrc	Pointer to the input data to decrypt (ciphertext).
pDst	Pointer to the output decrypted data (plaintext).
len	Length in bytes of the plaintext and ciphertext.
pAuthPassed	Pointer to the result flag.
pAuthKey	Pointer to the authentication key.
pConfKey	Pointer to the confidentiality key.
keyLen	Length of keys in bytes.
AD	Array of pointers to the associated input strings.
ADlen	Array of length (in bytes) of the associated input strings.
numAD	The number of the associated strings.
pSIV	Pointer to the synthetic initialization vector.

Description

The `AES_SIVDecrypt` function accepts authentication and confidentiality keys of length *keyLen* each, a vector *AD*[] of associated data (strings), 16-byte synthetic initialization vector (**pSIV*), and ciphertext (**pSrc*) of an arbitrary length *len*. The output of the function is the decrypted plaintext (**pDst*) of the same length as the ciphertext and the result of plaintext authentication (**pAuthPassed*).

The computation includes the following steps:

1. Decrypt the input ciphertext using the AES cipher in the CTR mode with the initial counter value (*CTR0*) equal to the synthetic initialization vector (**pSIV*) xored with a fixed mask.
2. Re-compute the synthetic initialization vector using the input data *AD*[] and the computed plaintext.

If the input and re-computed values of SIV are the same, the plaintext authentication is considered passed (**pAuthPassed* = 1), otherwise, the plaintext authentication is considered failed (**pAuthPassed* = 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` or a pointer *AD*[*i*] to any individual string is `NULL` while the length *ADlen*[*i*] is non-zero.
`ippStsLengthErr`	Indicates an error condition that occurs because of one of the following: • The *keyLen* parameter is different from 16, 24, and 32 • The number of the strings *numAD* in the *AD* array is negative or greater than 127 • The length *ADlen*[*i*] of any individual input string is negative • The *len* parameter is negative

Usage Example

```
////////////////////////// Usage example for AES-SIV primitives //////////////////////
   // key:
   Ipp8u key[] =
      "\x7f\x7e\x7d\x7c\x7b\x7a\x79\x78\x77\x76\x75\x74\x73\x72\x71\x70"
      "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f";
   // ADs:
   Ipp8u ad1[] =
      "\x00\x11\x22\x33\x44\x55\x66\x77\x88\x99\xaa\xbb\xcc\xdd\xee\xff"
      "\xde\xad\xda\xda\xde\xad\xda\xda\xff\xee\xdd\xcc\xbb\xaa\x99\x88"
      "\x77\x66\x55\x44\x33\x22\x11\x00";
   Ipp8u ad2[] =
      "\x10\x20\x30\x40\x50\x60\x70\x80\x90\xa0";
   Ipp8u nonce[] =
      "\x09\xf9\x11\x02\x9d\x74\xe3\x5b\xd8\x41\x56\xc5\x63\x56\x88\xc0";

   // PT
   Ipp8u pt[] =
      "\x74\x68\x69\x73\x20\x69\x73\x20\x73\x6f\x6d\x65\x20\x70\x6c\x61"
      "\x69\x6e\x74\x65\x78\x74\x20\x74\x6f\x20\x65\x6e\x63\x72\x79\x70"
      "\x74\x20\x75\x73\x69\x6e\x67\x20\x53\x49\x56\x2d\x41\x45\x53";
   Ipp8u rt[sizeof(pt)];
   int authRes = 0xaa;
```

```
   Ipp8u auth_ct[16+sizeof(pt)-1];

   Ipp8u* adSlist[] = {ad1,
                       ad2,
                       nonce,
                       pt};
   int    adLlist[] = {(int)(sizeof(ad1)-1),
                       (int)(sizeof(ad2)-1),
                       (int)(sizeof(nonce)-1),
                       sizeof(pt)-1};
   Ipp8u v[16];

   // compute ISV
   ippsAES_S2V_CMAC(key, 16, (const Ipp8u**)adSlist, adLlist, sizeof(adSlist)/sizeof(void*), v);

   // encode
   ippsAES_SIVEncrypt(pt, auth_ct+16, sizeof(pt)-1, auth_ct,
  key, key+16, 16,
                      (const Ipp8u**)adSlist, adLlist, sizeof(adSlist)/sizeof(void*)-1);

   // decode
   ippsAES_SIVDecrypt(auth_ct+16, rt, sizeof(pt)-1, &authRes,
  key, key+16, 16,
                      (const Ipp8u**)adSlist, adLlist, sizeof(adSlist)/sizeof(void*)-1,
                      auth_ct);


   if((1==authRes) && (0==memcmp(pt, rt, sizeof(pt)-1)))
      printf("authenticated decryption passed\n");
   else
      printf("authenticated decryption failed\n");
//////////////////////////////////////////////////////////////////////
```

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. The `GetSize` function does not require allocated memory. You need to call the `GetSize` 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 `TDESEncryptCFB` operates under the CFB mode using its cipher scheme and requires having the initialization vector *pIV* and CFB block size *cfbBlkSize*.

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

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

1. Get the size required to configure the context `IppsDESSpec` by calling the function `DESGetSize`.
2. Call operating system memory allocation service function to allocate three buffers whose sizes are not less than the one specified by the function `DESGetSize`. Initialize pointers to contexts *pCtx1*, *pCtx2*, and *pCtx3* by calling the function `DESInit` three times, each with the allocated buffer and the respective DES key.
3. Specify the initialization vector and then call the function `TDESEncryptCBC` to encrypt the input data stream under CBC mode using TDES scheme.
4. 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.

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

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

See Also

Data Security Considerations

DESGetSize

Gets the size of the `IppsDESSpec` *context.*

Syntax

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

Include Files

```
ippcp.h
```

Parameters

pSize	Pointer to the `IppsDESSpec` context size value.

Description

This function gets the `IppsDESSpec` 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`.

DESInit

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

Syntax

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

Include Files

`ippcp.h`

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

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

Include Files

`ippcp.h`

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

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.

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 ifthe input data stream length *i*s not divisible by cipher block size .
ippStsContextMatchErr	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptECB

Decrypts variable length data stream in the ECB mode.

Syntax

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

Include Files

ippcp.h

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.
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.
ippStsUnderRunErr	Indicates an error condition if *srclen* is not divisible by cipher block size.

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`

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.
`ippStsUnderRunErr`	Indicates an error condition if the input data stream length is not divisible by cipher block size.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.

TDESDecryptCBC

Decrypts variable length data stream in the CBC mode.

Syntax

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

Include Files

`ippcp.h`

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.
`ippStsUnderRunErr`	Indicates an error condition if *srclen* is not divisible by cipher block size.

TDESEncryptCFB

Encrypts variable length data stream in the CFB mode.

Syntax

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

Include Files

`ippcp.h`

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

TDESDecryptCFB

Decrypts variable length data stream in the CFB mode.

Syntax

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

Include Files

`ippcp.h`

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.
`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.

TDESEncryptOFB

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

Syntax

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

Include Files

`ippcp.h`

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

TDESDecryptOFB

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

Syntax

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

Include Files

`ippcp.h`

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

Encrypts a variable length data stream in the CTR mode.

Syntax

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

Include Files

`ippcp.h`

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 that 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

```
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`

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 descrypted plaintext data stream length is less that 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`

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`

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`

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`

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`

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.
pCtx	Pointer to the `IppsSMS4Spec` 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 *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`

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

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

Include Files

`ippcp.h`

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.
`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.

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`

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`

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

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

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 *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

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

Include Files

ippcp.h

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

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

Include Files

ippcp.h

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.
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.
`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

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

Include Files

`ippcp.h`

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

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

Include Files

`ippcp.h`

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`

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`

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`

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 encrypts the input data stream of a variable length using 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`

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

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

Include Files

`ippcp.h`

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

3

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) Cryptography 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 Cryptography 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 Cryptography declares extensions of the Hash Functions, Hash Functions for Non-Streaming Messages, 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.

Supported Hash Algorithms

Value of `IppHashAlgId`	Associated Hash Algorithm
`ippHashAlg_SHA1`	SHA-1
`ippHashAlg_SHA224`	SHA-224
`ippHashAlg_SHA256`	SHA-256
`ippHashAlg_SHA384`	SHA-384
`ippHashAlg_SHA512`	SHA-512
`ippHashAlg_SHA512_224`	SHA-512/224
`ippHashAlg_SHA512_256`	SHA-512/256
`ippHashAlg_MD5`	MD5
`ippHashAlg_SM3`	SM3

Reduced Memory Footprint Functions

When your application uses the `IppHashAlgId` enumerator, it gets linked to all available hashing algorithm implementations. This results in unnecessary memory overhead if the application does not need all the algorithms. Intel IPP Cryptography includes a number of *reduced memory footprint* functions that allow you to select the exact hashing methods for your application's needs. These functions have the _rmf suffix in their names and use pointers to IppsHashMethod structure variables instead of IppHashAlgId values. To get a pointer to a IppsHashMethod structure variable, call an appropriate function from the table below. See HashMethod for the syntax.

NOTE
Functions that have the _TT suffix in their names return pointers to dynamically dispatched IppsHashMethod structures. These structures check for support of the SHA-NI instruction set at run time and choose the implementation of an algorithm depending on the outcome of the check. Using such IppsHashMethod structures leads to a slightly larger memory footprint compared to applications that use non-dynamically dispatched IppsHashMethod structures.

HashMethod Functions

Function name	Returns pointer to implementation of
ippsHashMethod_SHA1	SHA1 (without the SHA-NI instruction set)
ippsHashMethod_SHA1_NI	SHA1 (using the SHA-NI instruction set)
ippsHashMethod_SHA1_TT	SHA1 (using the SHA-NI instructions set if it is available at run time)
ippsHashMethod_SHA256	SHA256 (without the SHA-NI instruction set)
ippsHashMethod_SHA256_NI	SHA256 (using the SHA-NI instruction set)
ippsHashMethod_SHA256_TT	SHA256 (using the SHA-NI instructions set if it is available at run time)
ippsHashMethod_SHA224	SHA224 (without the SHA-NI instruction set)
ippsHashMethod_SHA224_NI	SHA224 (using the SHA-NI instruction set)
ippsHashMethod_SHA224_TT	SHA224 (using the SHA-NI instructions set if it is available at run time)
ippsHashMethod_SHA384	SHA384
ippsHashMethod_SHA512	SHA512
ippsHashMethod_SHA512_256	SHA512-256
ippsHashMethod_SHA512_224	SHA512-224
ippsHashMethod_MD5	MD5
ippsHashMethod_SM3	SM3

Important
The crypto community does not consider SHA-1 or MD5 algorithms secure anymore.

Recommendation: use a more secure hash algorithm (for example, any algorithm from the SHA-2 family) instead of SHA-1 or MD5.

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 `IppsSHA1StateIppsHashState` context.

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

> **NOTE**
> For memory-critical applications, consider using Reduced Memory Footprint Functions.

> **Important**
> The crypto community does not consider SHA-1 or MD5 algorithms secure anymore.
>
> Recommendation: use a more secure hash algorithm (for example, any algorithm from the SHA-2 family) instead of SHA-1 or MD5.

See Also

Data Security Considerations

HashGetSize

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

Syntax

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

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

Include Files

`ippcp.h`

Parameters

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

Description

The function gets the size of the `IppsHashState` or `IppsHashState_rmf` context in bytes and stores it in *`*pSize`*.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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`.

HashInit

Initializes user-supplied memory as `IppsHashState` *or* `IppsHashState_rmf` *context for future use.*

Syntax

```
IppStatus ippsHashInit(IppsHashState* pCtx, IppHashAlgId hashAlg);
IppStatus ippsHashInit_rmf(IppsHashState_rmf* pCtx, IppsHashMethod* pMethod);
```

Include Files

`ippcp.h`

Parameters

pCtx	Pointer to the `IppsHashState` or `IppsHashState_rmf` context being intialized.
hashAlg	Identifier of the hash algorithm.
pMethod	Pointer to the hash method.

Description

The function initializes the memory pointed by *pCtx* as `IppsHashState` or `IppsHashState_rmf` context. The *hashAlg* and *pMethod* parameters define 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. To get a value for the *pMethod* parameter, call one of the HashMethod functions.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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` *or* `IppsHashState_rmf` *context into/from a user-defined buffer.*

Syntax

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

```
IppStatus ippsHashPack_rmf(const IppsHashState_rmf* pCtx, Ipp8u* pBuffer, int
bufferSize);
```

```
IppStatus ippsHashUnpack (const Ipp8u* pBuffer, IppsHashState* pCtx);
```

```
IppStatus ippsHashUnpack_rmf(const Ipp8u* pBuffer, IppsHashState_rmf* pCtx);
```

Include Files

`ippcp.h`

Parameters

pCtx	Pointer to the `IppsHashState` or `IppsHashState_rmf` context.
pBuffer	Pointer to the user-defined buffer.
bufSize, *bufferSize*	The size of the user-defined buffer in bytes.

Description

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

The value of the *bufSize* parameter must be not less than the size of `IppsHashState` or `IppsHashState_rmf` context. Call the `HashGetSize` function prior to `HashPack` to determine the size of the buffer.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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`.
`ippStsMemErr`	Indicates an error condition if the value of *bufSize* is less than the size of the `IppsHashState` context.

`ippStsContextMatchErr`	Indicates an error condition in a `ippsHashPack_rmf` call if the context parameter does not match the operation.
`ippStsNoMem`	Indicates an error condition if the value of *`bufferSize`* is less than the size of the `IppsHashState_rmf` context.

HashDuplicate

Copies one `IppsHashState` *or* `IppsHashState_rmf` *context to another.*

Syntax

```
IppStatus ippsHashDuplicate(const IppsHashState* pSrcCtx, IppsHashState* pDstCtx);
IppStatus ippsHashDuplicate_rmf(const ippsHashState_rmf* pSrcCtx, ippsHashState_rmf* pDstCtx);
```

Include Files

`ippcp.h`

Parameters

`pSrcCtx`	Pointer to the input `IppsHashState` or `IppsHashState_rmf` context to be cloned.
`pDstCtx`	Pointer to the output `IppsHashState` or `IppsHashState_rmf` context.

Description

The function copies one `IppsHashState` or `IppsHashState_rmf` context to another.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 any of the context parameters does not match the operation.

HashUpdate

Digests the current input message stream of the specified length.

Syntax

```
IppStatus ippsHashUpdate(const Ipp8u *pSrc, int len, IppsHashState *pCtx);
IppStatus ippsHashUppdate_rmf(const Ipp8u *pSrc, int srcLen, ippsHashState_rmf *pCtx);
```

Include Files

ippcp.h

Parameters

pSrc	Pointer to the buffer containing a part of or the whole message.
len, *srcLen*	Length of the actual part of the message in bytes.
pCtx	Pointer to the `IppsHashState` or `IppsHashState_rmf` 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.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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);
IppStatus ippsHashFinal_rmf(Ipp8u *pHash, ippsHashState_rmf *pCtx);
```

Include Files

ippcp.h

Parameters

pMD, *pHash*	Pointer to the resultant digest value.
pCtx	Pointer to the `IppsHashState` or `IppsHashState_rmf` context.

Description

The function completes calculation of the digest value and stores the result at the specified memory location, then re-initializes the *pCtx* context.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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);
IppStatus ippsHashGetTag_rmf(Ipp8u* pTag, int tagLen, ippsHashState_rmf* pCtx);
```

Include Files

`ippcp.h`

Parameters

pTag	Pointer to the authentication tag.
tagLen	The length of the tag (in bytes).
pCtx	Pointer to the `IppsHashState` or `IppsHashState_rmf` 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.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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.
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.

HashMethod

Returns a pointer to a pre-defined hash algorithm.

Syntax

```
const IppsHashMethod* ippsHashMethod_SHA1(void);
const IppsHashMethod* ippsHashMethod_SHA1_NI(void);
const IppsHashMethod* ippsHashMethod_SHA1_TT(void);
const IppsHashMethod* ippsHashMethod_SHA256(void);
const IppsHashMethod* ippsHashMethod_SHA256_NI(void);
const IppsHashMethod* ippsHashMethod_SHA256_TT(void);
const IppsHashMethod* ippsHashMethod_SHA224(void);
const IppsHashMethod* ippsHashMethod_SHA224_NI(void);
const IppsHashMethod* ippsHashMethod_SHA224_TT(void);
const IppsHashMethod* ippsHashMethod_SHA512(void);
const IppsHashMethod* ippsHashMethod_SHA384(void);
const IppsHashMethod* ippsHashMethod_SHA512_224(void);
const IppsHashMethod* ippsHashMethod_SHA512_256(void);
const IppsHashMethod* ippsHashMethod_MD5(void);
const IppsHashMethod* ippsHashMethod_SM3(void);
```

Include Files

ippcp.h

Description

Each of these functions returns a pointer to a method-defined implementation of a particular hash algorithm. Use these functions in calls to HashInit and HashMessage. See table HashMethod Functions for an explanation of the values returned by the HashMethod functions.

Return Values

const ippsHashMethod*	Pointer to the particular hash method.

SM3GetSize

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

Syntax

`IppStatus ippsSM3GetSize(int *pSize);`

Include Files

`ippcp.h`

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`

Parameters

pCtx	Pointer to the `IppsSM3State` context being intialized.

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`

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`

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`

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`

Parameters

pMD	Pointer to the resultant digest value.
pCtx	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

```
IppStatus ippsSM3GetTag(Ipp8u* pTag, Ipp32u tagLen, const IppsSM3State* pCtx);
```

Include Files

`ippcp.h`

Parameters

pTag	Pointer to the authentication tag.
tagLen	Length of the tag (in bytes).
pCtx	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 *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.

MD5GetSize

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

Syntax

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

Include Files

`ippcp.h`

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

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

Include Files

`ippcp.h`

Parameters

pCtx	Pointer to the `IppsMD5State` context being intialized.

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`

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`

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`

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

`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.

MD5Final

Completes computation of the MD5 digest value.

Syntax

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

Include Files

`ippcp.h`

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

`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.

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`

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`

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`

Parameters

pCtx	Pointer to the `IppsSHA1State` context being intialized.

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`

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`

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.
`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.

SHA1Update

Digests the current input message stream of the specified length.

Syntax

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

Include Files

`ippcp.h`

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

```
IppStatus ippsSHA1Final(Ipp8u *pMD, IppsSHA1State *pCtx);
```

Include Files

ippcp.h

Parameters

pMD	Pointer to the resultant digest value.
pCtx	Pointer to the IppsSHA1State 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.

SHA1GetTag

Computes the current SHA-1 digest value of the processed part of the message.

Syntax

```
IppStatus ippsSHA1GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA1State* pState);
```

Include Files

ippcp.h

Parameters

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

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

Include Files

`ippcp.h`

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

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

Include Files

`ippcp.h`

Parameters

pCtx	Pointer to the `IppsSHA224State` context being intialized.

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`

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`

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`

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`

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

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

Include Files

`ippcp.h`

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`

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`

Parameters

pCtx	Pointer to the `IppsSHA256State` context being intialized.

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.
`ippStsNullPtrErr`	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`

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`

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`

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

Completes computation of the SHA-256 digest value.

Syntax

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

Include Files

`ippcp.h`

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`

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`

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`

Parameters

pCtx	Pointer to the `IppsSHA384State` context being intialized.

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`

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

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

Include Files

`ippcp.h`

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

SHA384Update

Digests the current input message stream of the specified length.

Syntax

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

Include Files

`ippcp.h`

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

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

Include Files

`ippcp.h`

Parameters

pMD	Pointer to the resultant digest value.
pCtx	Pointer to the `IppsSHA384State` 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.

SHA384GetTag

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

Syntax

```
IppStatus ippsSHA384GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA384State* pState);
```

Include Files

`ippcp.h`

Parameters

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

`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.

SHA512GetSize

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

Syntax

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

Include Files

`ippcp.h`

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

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

Include Files

`ippcp.h`

Parameters

pCtx	Pointer to the `IppsSHA512State` context being intialized.

Description

The function initializes the memory pointed by *pCtx* as `IppsSHA512State` context.

Return Values

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

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`

Parameters

pCtx	Pointer to the `IppsSHA512State` context.
pBuffer	Pointer to the user-defined buffer.

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

`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`.

SHA512Duplicate
Copies one `IppsSHA512State` *context to another.*

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

Include Files
`ippcp.h`

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

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

Include Files

ippcp.h

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 IppsSHA512State 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.

SHA512Final

Completes computation of the SHA-512 digest value.

Syntax

```
IppStatus ippsSHA512Final(Ipp8u *pMD, IppsSHA512State *pCtx);
```

Include Files

`ippcp.h`

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

SHA512GetTag

Computes the current SHA-512 digest value of the processed part of the message.

Syntax

```
IppStatus ippsSHA512GetTag(Ipp8u* pDstTag, Ipp32u tagLen, const IppsSHA512State*
pState);
```

Include Files

`ippcp.h`

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.

> **Important**
> The crypto community does not consider SHA-1 or MD5 algorithms secure anymore.
>
> Recommendation: use a more secure hash algorithm (for example, any algorithm from the SHA-2 family) instead of SHA-1 or MD5.

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 octers.
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);
```

```
IppStatus ippsHashMessage_rmf(const Ipp8u *pMsg, int msgLen, Ipp8u *pHash, const ippsHashMethod *pMethod);
```

Include Files

`ippcp.h`

Parameters

pMsg	Pointer to the input message.
len, *msgLen*	Message length in octets.
pMD, *pHash*	Pointer to the resultant digest.
hashAlg	Identifier of the hash algorithm.
pMethod	Pointer to the hash method.

Description

The function uses the selected hash algorithm to compute the digest value of the entire (non-streaming) input message. The *hashAlg* and *pMethod* parameters define the hash algorithm used. The *hashAlg* parameter can take one of the values listed in table Supported Hash Algorithms. To get a value for the *pMethod* parameter, call one of the HashMethod functions.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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.
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.

```
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

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

Include Files

`ippcp.h`

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.

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.

MD5MessageDigest

Computes MD5 digest value of the input message.

Syntax

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

Include Files

`ippcp.h`

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.

SHA1MessageDigest

Computes SHA-1 digest value of the input message.

Syntax

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

Include Files

`ippcp.h`

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.

```
    //    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[] = "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq";
   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

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

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.

SHA384MessageDigest

Computes SHA-384 digest value of the input message.

Syntax

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

Include Files

ippcp.h

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

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.

Important
The crypto community does not consider SHA-1 or MD5 algorithms secure anymore.

Recommendation: use a more secure hash algorithm (for example, any algorithm from the SHA-2 family) instead of SHA-1 or MD5.

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 pseodorandom 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`

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string.
pMask	Pointer to the output pseodorandom 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 *algID*. 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 *pMask* pointer is `NULL`.
`ippStsLengthErr`	Indicates an error condition if any of the specified lengths is negative or zero.
`ippStsNotSupportedModeErr`	Indicates an error condition if the *hashAlg* parameter does not match any value of `IppHashAlg` listed in table Supported Hash Algorithms.

MGF1_rmf, MGF2_rmf

Generates a pseudorandom mask of the specified length using a selected hash lagorithm based on MGF1 or MGF2 specifications.

Syntax

```
IppStatus ippsMGF1_rmf(const Ipp8u* pSeed, int seedLen, Ipp8u* pMask, int maskLen,
const IppsHashMethod* pMethod);
```

```
IppStatus ippsMGF2_rmf(const Ipp8u* pSeed, int seedLen, Ipp8u* pMask, int maskLen,
const IppsHashMethod* pMethod);
```

Include Files

ippcp.h

Parameters

pSeed	Pointer to the input octet string.
seedLen	Length of the input string in bytes.
pMask	Pointer to the output pseodorandom mask.
maskLen	Desired length of the output.
pMethod	Pointer to the hash method.

Description

The function generates a pseudorandom mask of the specified length using the hash algorithm defined by *pMethod*, as defined in the MGF1 and MGF2 specifications. To get a value for the *pMethod* parameter, call one of the HashMethod functions.

NOTE
These are *reduced memory footprint* functions. To learn more, see Reduced Memory Footprint 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
ippStsLengthErr	Indicates an error condition if any of the specified lengths is negative or zero.

Data Authentication Primitive Functions

4

This chapter describes the Intel® IPP Cryptography 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 theOne-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 HMAC_GetSize 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 HMAC_Init 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 HMAC_Update to digest incoming message stream in the queue till its completion. To determine the current value of the message digest, call HMAC_GetTag between the two calls to `HMACUpdate`.
4. Call the function HMAC_Final 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:

Important
The crypto community does not consider HMACSHA1 or HMACMD5 secure anymore.

Recommendation: use a more secure hash algorithm (for example, any algorithm from the SHA-2 family) instead of HMACSHA1 or HMACMD5.

See Also

Data Security Considerations

HMAC_GetSize

Gets the size of the `IppsHMACState` *or* `IppsHMACState_rmf` *context.*

Syntax

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

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

Include Files

`ippcp.h`

Parameters

pSize	Pointer to the value of the `IppsHMACState` or `IppsHMACState_rmf` context size.

Description

The function gets the size of the `IppsHMACState` or `IppsHMACState_rmf` context in bytes and stores it in *pSize*.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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`.

HMAC_Init

Initializes user-supplied memory as `IppsHMACState` *or* `IppsHMACState_rmf` *context for future use.*

Syntax

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

```
IppStatus ippsHMACInit_rmf(const Ipp8u* pKey, int keyLen, IppsHMACState_rmf* pCtx,
const IppsHashMethod* pMethod);
```

Include Files

`ippcp.h`

Parameters

pKey	Pointer to the user-supplied key.
keyLen	Key length in bytes.
pCtx	Pointer to the `IppsHMACState` or `IppsHMACState_rmf` context being initialized.

hashAlg	Identifier of the hash algorithm.
pMethod	Pointer to the hash method.

Description

The function initializes the memory pointed to by *pCtx* as the `IppsHMACState` or `IppsHMACState_rmf` context. The function also sets up the initial chaining digest value according to the hash algorithm specified by the *hashAlg* or *pMethod*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. To get a value for the *pMethod* parameter, call one of the HashMethod functions.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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`.
`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` *or* `IppsHMACState_rmf` *context into/from a user-defined buffer.*

Syntax

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

IppStatus ippsHMACPack_rmf (const IppsHMACState_rmf* pCtx, Ipp8u* pBuffer, int
bufSize);

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

IppStatus ippsHMACUnpack_rmf (const Ipp8u* pBuffer, IppsHMACState_rmf* pCtx);
```

Include Files

`ippcp.h`

Parameters

pCtx	Pointer to the `IppsHMACState` or `IppsHMACState_rmf` 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 or IppsHMACState_rmf context. The HMAC_Pack and HMAC_Unpack functions enable replacing the position-dependent IppsHMACState or IppsHMACState_rmf context in the memory. Call the HMAC_GetSize function prior to HMAC_Pack to determine the size of the buffer.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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.
ippStsMemErr	Indicates an error condition if the value of *bufSize* is less than the size of the IppsHMACState or IppsHMACState_rmf context.

HMAC_Duplicate

Copies one IppsHMACState *or* IppsHMACState_rmf *context to another.*

Syntax

```
IppStatus ippsHMAC_Duplicate(const IppsHMACState* pSrcCtx, IppsHMACState* pDstCtx);
IppStatus ippsHMACDuplicate_rmf(const IppsHMACState_rmf* pSrcCtx, IppsHMACState_rmf* pDstCtx);
```

Include Files

ippcp.h

Parameters

pSrcCtx	Pointer to the input IppsHMACState or IppsHMACState_rmf context to be cloned.
pDstCtx	Pointer to the output IppsHMACState or IppsHMACState_rmf context.

Description

The function copies one IppsHMACState or IppsHMACState_rmf context to another.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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);
IppStatus ippsHMACUpdate_rmf(const Ipp8u *pSrc, int len, IppsHMACState_rmf *pCtx);
```

Include Files

ippcp.h

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 or IppsHMACState_rmf 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.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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);
IppStatus ippsHMACFinal_rmf(Ipp8u *pMD, int mdLen, IppsHMACState_rmf *pCtx);
```

Include Files

`ippcp.h`

Parameters

pMD	Pointer to the resultant HMAC value.
mdLen	Specified HMAC length.
pCtx	Pointer to the `IppsHMACState` or `IppsHMACState_rmf` context.

Description

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

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 the context parameter does not match the operation.
`ippStsLengthErr`	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

```
IppStatus ippsHMAC_GetTag(Ipp8u* pMD, int mdLen, const IppsHMACState* pCtx);
IppStatus ippsHMACGetTag_rmf(Ipp8u* pMD, int mdLen, const IppsHMACState_rmf* pCtx);
```

Include Files

`ippcp.h`

Parameters

pMD	Pointer to the authentication tag.
mdLen	The length of the tag (in bytes).
pCtx	Pointer to the `IppsHMACState` or `IppsHMACState_rmf` 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.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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`.
`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

```
IppStatus ippsHMAC_Message(const Ipp8u *pMsg, int msgLen, const Ipp8u *pKey, int
keyLen, Ipp8u *pMD, int mdLen, IppHashAlgId hashAlg);
```

```
IppStatus ippsHMACMessage_rmf(const Ipp8u *pMsg, int msgLen, const Ipp8u *pKey, int
keyLen, Ipp8u *pMAC, int macLen, const ippsHashMethod *pMethod);
```

Include Files

`ippcp.h`

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, *pMAC*	Pointer to the resultant HMAC value.
mdLen, *macLen*	Specified HMAC length.
hashAlg	Identifier of the hash algorithm.

pMethod	Pointer to the hash method.

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* or *pMAC* of the specified length *mdLen* or *macLen*. The *hashAlg* and *pMethod* parameters define the hash algorithm applied. The *hashAlg* parameter can take one of the values listed in table Supported Hash Algorithms. To get a value for the *pMethod* parameter, call one of the HashMethod functions.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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`.
`ippStsLengthErr`	Indicates an error condition if: • *msgLen* is less than zero • *mdLen* is less than one or greater than the length of the hash value • *macLen* is less than one or greater than the length of the hash value
`ippsStsNotSupportedModeErr`	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`

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`

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);
```

Include Files

`ippcp.h`

Parameters

pSrc	Pointer to the buffer containing a part or the entire message.
len	Length of the actual part of the message in bytes.
pState	Pointer to the `IppsAES_CMACState` context.

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`

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`

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

5

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

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

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

```
typedef enum {
IppsBigNumNEG=0,
IppsBigNumPOS=1
} IppsBigNumSGN;
```

The functions described in this section use the context `IppsBigNumState` to serve as an operational vehicle that carries not only the sign and value of the data, but also a sufficient working buffer reserved for various arithmetic operations. The length of the context `IppsBigNumState` is defined as the length of the data carried by the structure and the size of the context `IppsBigNumState` is therefore defined as the maximal length of the data that this operational vehicle can carry.

NOTE
In all unsigned big number arithmetic functions, 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

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

Include Files

`ippcp.h`

Parameters

length	The length of the integer big number in `Ipp32u`.
size	Size of the buffer in bytes required for initialization.

Description

The function specifies the buffer size required to define a structured working buffer of the context `IppsBigNumState` for the storage and operations on an integer big number in bytes.

> **NOTE**
> For security reasons, the length of the big number is restricted to 16 kilobits.

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 or greater than 512.

BigNumInit

Initializes context and partitions allocated buffer.

Syntax

`IppStatus ippsBigNumInit(int` *`length`*`, IppsBigNumState *`*`b`*`);`

Include Files

`ippcp.h`

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.

> **NOTE**
> For security reasons, the length of the big number is restricted to 16 kilobits.

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 or greater than 512.

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`

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.

```
IppsBigNumState* New_BN(int size, const Ipp32u* pData=0){
   // get the size of the Big Number context
   int ctxSize;
   ippsBigNumGetSize(size, &ctxSize);
   // allocate the Big Number context
   IppsBigNumState* pBN = (IppsBigNumState*) (new Ipp8u [ctxSize] );
   // and initialize one
   ippsBigNumInit(size, pBN);

   // if any data was supplied, then set up the Big Number value
   if(pData)
      ippsSet_BN(IppsBigNumPOS, size, pData, pBN);

   // return pointer to the Big Number context for future use
   return pBN;
}
```

SetOctString_BN

Converts octet string into a positive Big Number.

Syntax

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

Include Files

`ippcp.h`

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.

```
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 srting
   ippsSetOctString_BN(desiredBNvalue, sizeof(desiredBNvalue)-1, pBN);

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

GetSize_BN

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

Syntax

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

Include Files

ippcp.h

Parameters

b	Integer big number of the data type `IppsBigNumState`.
size	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

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

Include Files

`ippcp.h`

Parameters

sgn	Sign of `IppsBigNumState *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

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

Include Files

`ippcp.h`

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`

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

Parameters

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

```
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

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

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the Big Number A.
pB	Pointer to the context of the Big Number B.
pResult	Pointer to the result of the comparison.

Description

This function compares Big Numbers A and B and sets up the result according to the following conditions:

- if A==B, then **pResult* = `IS_ZERO`
- if A > B, then **pResult* = `GREATER_THAN_ZERO`
- if A < B, then **pResult* = `LESS_THAN_ZERO`

Return Values

`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.

CmpZero_BN

Checks the value of the input data field.

Syntax

```
IppStatus ippsCmpZero_BN(const IppsBigNumState *b, Ipp32u *result);
```

Include Files

ippcp.h

Parameters

b	Integer big number of the data type IppsBigNumState.
result	Indicates whether the input integer big number is positive, negative, or zero.

Description

The function scans the data field of the input const IppsBigNumState *b and returns

- IS_ZERO if the value held by IppsBigNumState *b is zero
- GREATER_THAN_ZERO if the input is more than zero
- LESS_THAN_ZERO if the input is less than zero.

Return Values

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

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:

$(*r) \leftarrow (*a) + (*b)$.

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 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.

```
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

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

Include Files

ippcp.h

Parameters

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

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*) ← (**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 IppsBigNumState **r* is smaller than the result data length.

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

Mul_BN

Multiplies two integer big numbers.

Syntax

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

Include Files

ippcp.h

Parameters

a	Multiplicand of IppsBigNumState.
b	Multiplier of 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←*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 `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.

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`

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←*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 `IppsBigNumState *r` is smaller than the result data length.

NOTE
The function executes only under the condition that the size `IppsBigNumState *r` is not less than the sum of the lengths of `IppsBigNumState *a` or that of `IppsBigNumState *b` minus one.

Div_BN

Divides one integer big number by another.

Syntax

```
IppStatus ippsDiv_BN(IppsBigNumState *a, IppsBigNumState *b, IppsBigNumState * q,
IppsBigNumState *r);
```

Include Files

`ippcp.h`

Parameters

a	Dividend of `IppsBigNumState`.
b	Divisor of `IppsBigNumState`.
q	Quotient of `IppsBigNumState`.
r	Remainder of `IppsBigNumState`.

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

`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*b` or when the size of `IppsBigNumState *q` is smaller than the quotient result data length.
`ippStsDivByZeroErr`	Indicates an error condition if the zero divisor is attempted.

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`

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←amod 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

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

Include Files

`ippcp.h`

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 ←*gcd* (*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 `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

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

Include Files

`ippcp.h`

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 *e = 1 mod 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 - 1 mod n`.

For better results, functions included in the cryptography package use $R = b^k$ where $b = 2^{32}$ and `k` is the Montgomery index integer computed by the ceiling function of the bit length of the integer *n* over 32.

All functions use employ the context `IppsMontState` to serve as an operational vehicle to carry the Montgomery reduction index *k*, the integer big number modulus *n*, the least significant word `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 $T = x^e \bmod n$. Consider computing $T = x^4 \bmod n$, for some integer *x* with `0 < x < n`.

First get the buffer size required to configure the context `IppsMontState` by calling `MontGetSize` and then allocate the working buffer using OS service function, with allocated buffer to call `MontInit` to initialize the context `IppsMontState`.

Set the modulus *n* by calling`MontSet` and then convert *x* into its respective Montgomery form by calling `MontForm`, that is, computing

$$\underline{x} = xR \bmod n.$$

Then compute the Montgomery reduction of

$$\underline{x}\underline{x}$$

using the function MontMul to generate

$$T = \underline{x}\underline{x}R^{-1} \bmod n .$$

The Montgomery reduction of `T*T`*mod n* with respect to *R* is

$$T^2R^{-1} \bmod n = (\underline{x}^2R^{-1})^2 R^{-1} \bmod n = x^4 R \bmod n .$$

Further applying MontMul with this value and the value of 1 yields the desired result $T = x^4 \bmod n$.

The classical modular exponentiation should be computed by performing the following sequence of operations:

1. Get the buffer size required to configure the context `IppsMontState` by calling the function `MontGetSize`. For limited memory system, choose binary method, and otherwise, choose sliding window method. Using the binary method reduces the buffer size significantly while using sliding window method enhances the performance.
2. Allocate working buffer through an operating system memory allocation function and configure the structure `IppsMontState` by calling the function `MontInit` with the allocated buffer and the choice made on the modular exponential method at time invoking `MontGetSize`.
3. Call the function `MontSet` to set the integer big number module for `IppsMontState`.
4. Call the function `MontForm` to convert the integer *x* to be its Montgomery form.
5. Call the function`MontExp` to compute the Montgomery modular exponentiation.
6. Call the function `MontMul` to compute the Montgomery modular multiplication of the above result with the integer 1 as to convert the above result back to the desired classical modular exponential result.
7. 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

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

Include Files

```
ippcp.h
```

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 structured working buffer of the context `IppsMontState` to store the modulus and perform operations using various Montgomery modulus schemes.

> **NOTE**
> For security reasons, the length of the modulus is restricted to 16 kilobits.

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 or greater than 512.

MontInit

Initializes the context and partitions the specified buffer space.

Syntax

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

Include Files

`ippcp.h`

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 *`length*sizeof`*`(Ipp32u)`-bit big number modulus and execute various Montgomery modulus operations.

> **NOTE**
> For security reasons, the length of the modulus is restricted to 16 kilobits.

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 or greater than 512.

See Also

Data Security Considerations

MontSet

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

Syntax

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

Include Files

`ippcp.h`

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 $R*R^{-1} - n*Ni = 1$.

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`.
`ippStsBadModulusErr`	Indicates an error condition if the modulus is not a positive odd integer.
`ippStsLengthErr`	Indicates an error condition if *length* is less than or equal to 0.
`ippStsOutOfRangeErr`	Indicates an error condition if *length* is larger than `IppsMontState*m`.

MontGet

Extracts the big number modulus.

Syntax

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

Include Files

`ippcp.h`

Parameters

m	context `IppsMontState`.
n	Modulus data field.
length	Modulus data length in `Ipp32u` chunks.

Description

The function extracts the big number modulus from the input `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

Converts input positive integer big number into Montgomery form.

Syntax

```
IppStatus ippsMontForm(const IppsBigNumState* a, IppsMontState* m, IppsBigNumState* r);
```

Include Files

`ippcp.h`

Parameters

a	Input integer big number within the range `[0, m - 1]`.
m	Input big number modulus of `IppsBigNumState`.
r	Resulting Montgomery form $r = a*R \bmod m$.

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:

$r \leftarrow a*R \bmod m$.

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`.
`ippStsBadArgErr`	Indicates an error condition if *a* is a negative integer.

`ippStsScaleRangeErr`	Indicates an error condition if *a* 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*.

MontMul

Computes Montgomery modular multiplication for positive integer big numbers of Montgomery form.

Syntax

```
IppStatus ippsMontMul(const IppsBigNumState *a, const IppsBigNumState *b, IppsMontState
*m, IppsBigNumState *r);
```

Include Files

`ippcp.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 `IppsMontState *m`. As a result, `IppsBigNumState *r` holds the product.

The following pseudocode represents this function:

$r \leftarrow a*b*R^{-1} \bmod m$.

Return Values

`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 `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*.

Example of Using Montgomery Reduction Scheme Functions

Montgomery Multiplication

```
void MontMul_sample(void){
   int size;

   // define and initialize Montgomery Engine over Modulus N
   Ipp32u bnuN = 19;
   ippsMontGetSize(IppsBinaryMethod, 1, &size);
   IppsMontState* pMont = (IppsMontState*)( new Ipp8u [size] );
   ippsMontInit(IppsBinaryMethod, 1, pMont);
   ippsMontSet(&bnuN, 1, pMont);

   // define and init Big Number multiplicant A
   Ipp32u bnuA = 12;
   IppsBigNumState* bnA = New_BN(1, &bnuA);
   // encode A into Montfomery 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

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:

$r \leftarrow a^{e} R^{-(e-1)} \bmod m$.

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

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`

Parameters

seedBits	Size in bits for the seed value.
pCtx	Pointer to the `IppsPRNGState` context being intialized.

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 = *0x*C3D2E1F01032547698BADCFEEFCDAB8967452301

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`

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

```
IppStatus ippsPRNGGetSeed(const IppsPRNGState* pCtx, IppsBigNumState* pSeed);
```

Include Files

`ippcp.h`

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

```
IppStatus ippsPRNGSetAugment(const IppsBigNumState* pAugment, IppsPRNGState* pCtx);
```

Include Files

`ippcp.h`

Parameters

`pAugment`	Pointer to the entropy augment value being set up.
`pCtx`	Pointer to the `IppsPRNGState` context.

Description

The function resets entropy augment value with the supplied value of the *`seedBits`* bit length. The supplied big number should be created prior to the function call using the appropriate Big Number Arithmetic functions (see Example 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.

PRNGSetModulus

Sets the initial state with the given input modulus for the pseudorandom number generation.

Syntax

```
IppStatus ippsPRNGSetModulus(const IppsBigNumState* pModulus, IppsPRNGState* pCtx);
```

Include Files

`ippcp.h`

Parameters

pModulus	Pointer to the modulus value being set up.
pCtx	Pointer to the `IppsPRNGState` context.

Description

The function resets the modulus value with the supplied value up to 160 bit length. The supplied big number should be created prior to the function call using the appropriate Big Number Arithmetic functions (see Example 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.

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`

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 bit length.

Syntax

```
IppStatus ippsPRNGen(Ipp32u* pRand, int nBits, void* pCtx);
```

Include Files

`ippcp.h`

Parameters

pRand	Pointer to the output pseudorandom unsigned integer big number.
nBits	The number of the generated pseudorandom bits.
pCtx	Pointer to the `IppsPRNGState` context.

Description

The function generates a 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.

PRNGenRDRAND

Generates a pseudorandom unsigned Big Number of the specified bit length using the RDRAND instruction.

Syntax

```
IppStatus ippsPRNGenRDRAND(Ipp32u* pRand, int nBits, void* pCtx);
```

Include Files

`ippcp.h`

Parameters

pRand	Pointer to the output pseudorandom unsigned integer big number.
nBits	The number of generated pseudorandom bits.
pCtx	Pointer to the `IppsPRNGState` context. This pointer is unused and can be `NULL`.

Description

The function generates a pseudorandom unsigned integer big number of the specified *nBits* length. The generation is based on the RDRAND instruction available on latest Intel® processors [INTEL_ARCH].

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Return Values

`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.
`ippStsNotSupportedModeErr`	Indicates an error condition if the RDRAND instruction is not available on the target processor.

TRNGenRDSEED

Generates a pseudorandom unsigned Big Number of the specified bit length using the RDSEED instruction.

Syntax

```
IppStatus ippsTRNGenRDSEED(Ipp32u* pRand, int nBits, void* pCtx);
```

Include Files

`ippcp.h`

Parameters

pRand	Pointer to the output pseudorandom unsigned integer big number.
nBits	The number of generated pseudorandom bits.
pCtx	Pointer to the `IppsPRNGState` context. This pointer is unused and can be `NULL`.

Description

The function generates a pseudorandom unsigned integer big number of the specified *nBits* length. The generation is based on the RDSEED instruction available on latest Intel® processors [INTEL_ARCH].

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-

Optimization Notice
dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Return Values

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.
ippStsNotSupportedModeErr	Indicates an error condition if the RDSEED instruction is not available on the target processor.

PRNGen_BN

Generates a pseudorandom positive Big Number of the specified bitlength.

Syntax

```
IppStatus ippsPRNGen_BN(IppsBigNumState* pRandBN, int nBits, void* pCtx);
```

Include Files

ippcp.h

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.
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.

PRNGenRDRAND_BN

Generates a pseudorandom positive Big Number of the specified bit length using the RDRAND instruction.

Syntax

```
IppStatus ippsPRNGenRDRAND_BN(IppsBigNumState* pRand, int nBits, void* pCtx);
```

Include Files

`ippcp.h`

Parameters

pRand	Pointer to the output pseudorandom Big Number.
nBits	The number of generated pseudorandom bits.
pCtx	Pointer to the `IppsPRNGState` context. This pointer is unused and can be `NULL`.

Description

The function generates a pseudorandom positive Big Number of the specified *nBits* length. The generation is based on the RDRAND instruction available on latest Intel® processors [INTEL_ARCH].

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Return Values

`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.
`ippStsNotSupportedModeErr`	Indicates an error condition if the RDRAND instruction is not available on the target processor.

TRNGenRDSEED_BN

Generates a pseudorandom positive Big Number of the specified bit length using the RDSEED instruction.

Syntax

`IppStatus ippsTRNGenRDSEED_BN(IppsBigNumState* pRand, int nBits, void* pCtx);`

Include Files

`ippcp.h`

Parameters

pRand	Pointer to the output pseudorandom Big Number.
nBits	The number of generated pseudorandom bits.
pCtx	Pointer to the `IppsPRNGState` context. This pointer is unused and can be `NULL`.

Description

The function generates a pseudorandom positive Big Number of the specified *nBits* length. The generation is based on the RDSEED instruction available on latest Intel® processors [INTEL_ARCH].

Optimization Notice

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

Notice revision #20110804

Return Values

`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.
`ippStsNotSupportedModeErr`	Indicates an error condition if the RDSEED instruction is not available on the target processor.

Example of Using Pseudorandom Number Generation Functions

Find Pseudorandom Co-primes

```
void FindCoPrimes(void){
   int size;

   // define Pseudo Random Generator (default settings)
   ippsPRNGGetSize(&size);
   IppsPRNGState* pPrng = (IppsPRNGState*)(new Ipp8u [size] );
   ippsPRNGInit(160, pPrng);
```

```
   // define 256-bits Big Numbers X and Y
   const int bnBitSize = 256;
   IppsBigNumState* bnX = New_BN(bnBitSize/32);
   IppsBigNumState* bnY = New_BN(bnBitSize/32);

   // define temporary Big Numbers GCD and 1
   IppsBigNumState* bnGCD = New_BN(bnBitSize/32);
   Ipp32u one = 1;
   IppsBigNumState* bnOne = New_BN(1, &one);


   // generate pseudo random X and Y
   // while GCD(X,Y) != 1
   Ipp32u result;
   int counter;
   for(counter=0,result=1; result; counter++) {
      ippsPRNGen_BN(bnX, bnBitSize, pPrng);
      ippsPRNGen_BN(bnY, bnBitSize, pPrng);
      ippsGcd_BN(bnX, bnY, bnGCD);
      ippsCmp_BN(bnGCD, bnOne, &result);
   }

   cout <<"Coprimes:" <<endl;
   Type_BN("X: ", bnX); cout <<endl;
   Type_BN("Y: ", bnY); cout <<endl;
   cout <<"were fond on " <<counter <<" attempt" <<endl;

   delete [] (Ipp8u*)pPrng;
   delete [] (Ipp8u*)bnX;
   delete [] (Ipp8u*)bnY;
   delete [] (Ipp8u*)bnGCD;
   delete [] (Ipp8u*)bnOne;
}
```

Prime Number Generation Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) Cryptography functions for prime number generation.

This section describes Intel IPP Cryptography 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

$$p_{k,t} \le k^{2/3} 2^{t-1/2} t 4^{2-\sqrt{tk}} \text{ for } t = 2,\ k \ge 88,\ \text{or } 3 \le t \le k/9,\ k \ge 21.$$

Subsequently, a practical strategy for generating a random k-bit probable prime is to repeatedly pick k-bit random odd integers until finding one integer that can pass a recognized probabilistic primality test scheme as a probable prime. The available set of probable prime number generation functions enables you to specify an appropriate value of the security parameter t used in the Miller-Rabin primality test to meet the cryptographic requirements for your application.

All Intel IPP for prime number generation use the context `IppsPrimeState` as an operational vehicle that carries the bitlength of the target probable prime number, the structure capturing the state of the pseudorandom number generation, the 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 `PrimeGetSize` to get the size required to configure the `IppsPrimeState` context.
2. Allocate memory through the operating system memory allocation function and configure the `IppsPrimeState` *context* by calling the function`PrimeInit`.
3. Generate a probable prime number of the specified bitlength by calling the function `PrimeGen_BN`. If the returned `IppStatus` is `ippStsInsufficientEntropy`, then change the parameters of the pseudorandom generator and call the function `PrimeGen_BN` again.
4. Clean up secret data stored in the context.
5. Free the memory allocated to the `IppsPrimeState` context by calling the operating system memory-free service function.

See Also

Data Security Considerations

PrimeGetSize

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

Syntax

`IppStatus ippsPrimeGetSize(int nMaxBits, int* pSize);`

Include Files

`ippcp.h`

Parameters

nMaxBits	Maximum length of the probable prime number in bits.
pSize	Pointer to the `IppsPrimeState` context size in bytes.

Description

The function gets the `IppsPrimeState` 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`.

`ippStsLengthErr`	Indicates an error condition if *nMaxBits* is less than 1.

PrimeInit

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

Syntax

```
IppStatus ippsPrimeInit(int nMaxBits, IppsPrimeState* pCtx);
```

Include Files

`ippcp.h`

Parameters

nMaxBits	Maximum length of the probable prime number in bits.
pCtx	Pointer to the `IppsPrimeState` context being initialized.

Description

The function initializes the memory pointed by *pCtx* 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 *nMaxBits* 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`

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`

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`

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

Parameters

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 if the given integer is a probable prime. The pseudorandom number used in the Miller-Rabin test is generated by the specified rndFunc Random Generator. The function sets up the **pResult* as IS_PRIME or IS_COMPOSITE to show if the input probable prime passes the Miller-Rabin test.

Return Values

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.

PrimeSet

Sets the Big Number for primality testing.

Syntax

```
IppStatus ippsPrimeSet(const Ipp32u* pBNU, int nBits, IppsPrimeState* pCtx);
```

Include Files

ippcp.h

Parameters

pBNU	Pointer to the unsigned integer big number.
nBits	Unsigned integer big number length in bits.
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

```
IppStatus ippsPrimeSet_BN(const IppsBigNumState* pBN, IppsPrimeState* pCtx);
```

Include Files

`ippcp.h`

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

```
IppStatus ippsPrimeGet(Ipp32u* pBNU, int *pSize, const IppsPrimeState *pCtx);
```

Include Files

ippcp.h

Parameters

pBNU	Pointer to the unsigned integer big number.
pSize	Pointer to the length of the unsinged integer big number.
pCtx	Pointer to the `IppsPrimeState` context.

Description

The function extracts the probable prime number from **pCtx* context and stores it into the specified unsigned integer 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.

PrimeGet_BN

Extracts the probable prime positive Big Number.

Syntax

```
IppStatus ippsPrimeGet_BN(IppsBigNumState* pBN, const IppsPrimeState *pCtx);
```

Include Files

ippcp.h

Parameters

pBN	Pointer to the Big NUmber context.
pCtx	Pointer to the `IppsPrimeState` context.

Description

The function extracts the probable prime positive big number from the **pCtx* context and stores it into the specified Big Number 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.
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

```
int PrimeGen_sample(void){PrimeGen   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 0==error;
}
```

RSA Algorithm Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) Cryptography 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,dP,dQ,qInv*) 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.

> **Important**
> To provide minimum security, the length of the RSA modulus must be equal to or greater than 1024 bits.

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.

> **Important**
> To provide minimum security, the length of the RSA modulus must be equal to or greater than 1024 bits.

RSA_GetSizePublicKey, RSA_GetSizePrivateKeyType1, RSA_GetSizePrivateKeyType2

Get the size of the `IppsRSAPublicKeyState` *or* `IppsRSAPrivateKeyState` *context.*

Syntax

`IppStatus ippsRSA_GetSizePublicKey(int rsaModulusBitSize, int publicExpBitSize, int* pKeySize);`

`IppStatus ippsRSA_GetSizePrivateKeyType1(int rsaModulusBitSize, int privateExpBitSize, int* pKeySize);`

`IppStatus ippsRSA_GetSizePrivateKeyType2(int factorPBitSize, int factorQBitSize, int* pKeySize);`

Include Files

`ippcp.h`

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`	For `RSA_GetSizePublicKey`, indicates an error condition if *publicExpBitSize* < 0 or *publicExpBitSize* > *rsaModulusBitSize*.

For `RSA_GetSizePrivateKeyType1`, indicates and error condition if *privateExpBitSize* <0 or *privateExpBitSize* > *rsaModulusBitSize*.

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`

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.
keyCtxSize	Available size in bytes of the memory buffer being initialized.

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 `RSA_GetSizePublicKey`, `RSA_GetSizePrivateKeyType1`, `RSA_GetSizePrivateKeyType2` 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.
`ippStsBadArgErr`	Indicates an error condition if *publicExpBitSize* > *rsaModulusBitSize* or *privateExpBitSize* > *rsaModulusBitSize*.
`ippStsMemAllocErr`	Indicates an error condition if the allocated memory is insufficient for the operation.

See Also

RSA_GetSizePublicKey, RSA_GetSizePrivateKeyType1, RSA_GetSizePrivateKeyType2
Data Security Considerations

RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2

Set up an RSA key in the existing RSA key context.

Syntax

```
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`

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*), 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

```
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`

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*).
`pInverseQ`	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

Parameters

pBufferSize	Pointer to the size of a temporary buffer.
pKey	Pointer to the RSA key context.

Description

These functions get the size of a temporary buffer for future use in public- or private-key RSA operations, respectively.

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`	For `RSA_GetBufferSizePublicKey`, indicates an error condition if the public key is not set up. For `RSA_GetBufferSizePrivateKeyType1`, indicates an error condition if the type 1 private key is not set up.

NOTE
You can set up the public key or type 1 private key in a call to `RSA_SetPublicKey` or `RSA_SetPrivateKeyType1`, respectively. For the `RSA_GetBufferSizePrivateKeyType2` function, it suffices to initialize the context for the key in a call to `RSA_InitPrivateKeyType2`.

See Also

RSA_SetPublicKey, RSA_SetPrivateKeyType1, RSA_SetPrivateKeyType2
RSA_InitPublicKey, RSA_InitPrivateKeyType1, RSA_InitPrivateKeyType2

RSA_GenerateKeys

Generates key components for the desired RSA cryptographic system.

Syntax

```
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`

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 *rndFunc*.
2. Computes the RSA composite modulus $n = (p*q)$.
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 *factorPBitSize* and *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.
ippStsSizeErr	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.
ippStsOutOfRangeErr	Indicates an error condition if the initial value for searching the public exponent, specified by *pSrcPublicExp*, is not positive.
ippStsBadArgErr	Indicates an error condition in cases not explicitly mentioned above.
ippStsInsufficientEntropy	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
RSA_ValidateKeys
Pseudorandom Number Generation Functions

RSA_ValidateKeys

Validates key components of the RSA cryptographic system.

Syntax

```
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

Parameters

pResult	Pointer to the result of validation.
pPublicKey	Pointer to the RSA public key.
pPrivateKeyType2	Pointer to the RSA private key type 2.
pPrivateKeyType1	Pointer to the RSA private key type 1. This parameter is optional and can have the value of `NULL`.
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

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 $e*d = 1 \bmod ((p-1)*(q-1))$.

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 `RSA_GetSizePublicKey` to get the size required to configure `IppsRSAPublicKeyState` 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 `RSA_GetSizePrivateKeyType2` 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

`ippcp.h`

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.
`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`.
`ippStsInvalidCryptoKeyErr`	Indicates an error condition if the RSA context has not been properly set up for the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if the big number specified by *pPtxt* is not positive or greater than the RSA modulus.
`ippStsSizeErr`	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

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

```
// P prime factor
BigNumber    P("0xEECFAE81B1B9B3C908810B10A1B5600199EB9F44AEF4FDA493B81A9E3D84F632"
                 "124EF0236E5D1E3B7E28FAE7AA040A2D5B252176459D1F397541BA2A58FB6599");
// Q prime factor
BigNumber    Q("0xC97FB1F027F453F6341233EAAAD1D9353F6C42D08866B1D05A0F2035028B9D86"
                 "9840B41666B42E92EA0DA3B43204B5CFCE3352524D0416A5A441E700AF461503");
// P's CRT exponent
BigNumber   dP("0x54494CA63EBA0337E4E24023FCD69A5AEB07DDDC0183A4D0AC9B54B051F2B13E"
                 "D9490975EAB77414FF59C1F7692E9A2E202B38FC910A474174ADC93C1F67C981");
// Q's CRT exponent
BigNumber   dQ("0x471E0290FF0AF0750351B7F878864CA961ADBD3A8A7E991C5C0556A94C3146A7"
                 "F9803F8F6F8AE342E931FD8AE47A220D1B99A495849807FE39F9245A9836DA3D");
// CRT coefficient
BigNumber invQ("0xB06C4FDABB6301198D265BDBAE9423B380F271F73453885093077FCD39E2119F"
                 "C98632154F5883B167A967BF402B4E9E2E0F9656E698EA3666EDFB25798039F7");
// rsa modulus N = P*Q
BigNumber    N("0xBBF82F090682CE9C2338AC2B9DA871F7368D07EED41043A440D6B6F07454F51F"
                 "B8DFBAAF035C02AB61EA48CEEB6FCD4876ED520D60E1EC4619719D8A5B8B807F"
                 "AFB8E0A3DFC737723EE6B4B7D93A2584EE6A649D060953748834B2454598394E"
                 "E0AAB12D7B61A51F527A9A41F6C1687FE2537298CA2A8F5946F8E5FD091DBDCB");
// private exponent
BigNumber    D("0xA5DAFC5341FAF289C4B988DB30C1CDF83F31251E0668B42784813801579641B2"
                 "9410B3C7998D6BC465745E5C392669D6870DA2C082A939E37FDCB82EC93EDAC9"
                 "7FF3AD5950ACCFBC111C76F1A9529444E56AAF68C56C092CD38DC3BEF5D20A93"
                 "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 geterators
   deletePrimeGen(pPrimeG);
   deletePRNG(pRand);

   if(IPP_IS_VALID!=validateRes) {
      cout <<"validation fail" << endl;
      error = 1;
      break;
   }

   // known plain- and ciper-texts
   BigNumber kat_PT("0x00EB7A19ACE9E3006350E329504B45E2CA82310B26DCD87D5C68F1EEA8F55267"
                      "C31B2E8BB4251F84D7E0B2C04626F5AFF93EDCFB25C9C2B3FF8AE10E839A2DDB"
                      "4CDCFE4FF47728B4A1B7C1362BAAD29AB48D2869D5024121435811591BE392F9"
                      "82FB3E87D095AEB40448DB972F3AC14F7BC275195281CE32D2F1B76D4D353E2D");
   BigNumber kat_CT("0x1253E04DC0A5397BB44A7AB87E9BF2A039A33D1E996FC82A94CCD30074C95DF7"
                      "63722017069E5268DA5D1C0B4F872CF653C11DF82314A67968DFEAE28DEF04BB"
                      "6D84B1C31D654A1970E5783BD6EB96A024C2CA2F4A90FE9F2EF5C9C140E5BB48"
                      "DA9536AD8700C84FC9130ADEA74E558D51A74DDF85D8B50DE96838D6063E0955");

   //
   // 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

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*);

IppStatus ippsRSAEncrypt_OAEP_rmf(const Ipp8u* *pSrc*, int *srcLen*, const Ipp8u* *pLabel*, int *labLen*, const Ipp8u* *pSeed*, Ipp8u* *pDst*, const IppsRSAPublicKeyState* *pKey*, const IppsHashMethod* *pMethod*, Ipp8u* *pBuffer*);

Include Files

ippcp.h

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.
pMethod	Pointer to the hash method. For details, see HashMethod functions.
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*.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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 the any input/output length parameters are inconsistent with one another.
`ippStsNotSupportedModeErr`	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 IppsRSAPrivateKeyState* pKey, IppHashAlgId hashAlg,
Ipp8u* pBuffer);
```

```
IppStatus ippsRSADecrypt_OAEP_rmf(const Ipp8u* pSrc, const Ipp8u* pLabel, int labLen,
Ipp8u* pDst, int* pDstLen, const IppsRSAPrivateKeyState* pKey, const IppsHashMethod*
pMethod, Ipp8u* pBuffer);
```

Include Files

`ippcp.h`

Parameters

pSrc	Pointer to the octet ciphertext to be decrypted.
pLabel	Pointer to the optional label to be associated with the message.
labLen	Length of the optional label.
pDst	Pointer to the output octet plaintext message.
pDstLen	Pointer to the length of the decrypted message.
pKey	Pointer to the properly initialized `IppsRSAPrivateKeyState` context.
hashAlg	ID of the hash algorithm used. For details, see table Supported Hash Algorithms.
pMethod	Pointer to the hash method. For details, see HashMethod functions.
pBuffer	Pointer to a temporary buffer of size not less than returned by the `RSA_GetBufferSizePrivateKey` function.

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.

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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`.
`ippStsLengthErr`	Indicates an error condition if the 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.

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`

Parameters

pSrc	Pointer to the input octet message to be encrypted.
srcLen	Length (in bytes) of the message. The message can be empty, that is, *srcLen*==0.
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.

> **NOTE**
> You can set up the public key in a call to `RSA_SetPublicKey`.

`ippStsSizeErr`	Indicates an error condition if any input/output length parameters are inconsistent with one another.

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`

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 and 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.

> **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`.

`ippStsSizeErr`	Indicates an error condition if any input/output length parameters are inconsistent with one another.

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

```
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);
```

```
IppStatus ippsRSASign_PSS_rmf(const Ipp8u* pMsg, int msgLen, const Ipp8u* pSalt, int
saltLen, Ipp8u* pSign, const IppsRSAPrivateKeyState* pPrivateKey, const
IppsRSAPublicKeyState* pPublicKeyOpt, const IppsHashMethod* pMethod, Ipp8u* pBuffer);
```

Include Files

`ippcp.h`

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.
pMethod	Pointer to the hash method. For details, see HashMethod functions.
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* or *pMethod* 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.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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.
`ippStsLengthErr`	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).
`ippsStsNotSupportedModeErr`	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*);

IppStatus ippsRSAVerify_PSS_rmf(const Ipp8u* *pMsg*, int *msgLen*, const Ipp8u* *pSign*, int* *pIsSignValid*, const IppsRSAPublicKeyState* *pKey*, const IppsHashMethod* *pMethod*, Ipp8u* *pBuffer*);

Include Files

ippcp.h

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.
pMethod	Pointer to the hash method. For details, see HashMethod functions.
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* or *pMethod* parameter.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 the context parameter does not match the operation.
ippStsIncompleteContextErr	Indicates an error condition if the public key is not set up.
ippsStsNotSupportedModeErr	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

```
IppStatus ippsRSASign_PKCS1v15(const Ipp8u* pMsg, int msgLen, Ipp8u* pSign, const
IppsRSAPrivateKeyState* pPrivateKey, const IppsRSAPublicKeyState* pPublicKeyOpt,
IppHashAlgId hashAlg, Ipp8u* pBuffer);
```

```
IppStatus ippsRSASign_PKCS1v15_rmf(const Ipp8u* pMsg, int msgLen, Ipp8u* pSign, const
IppsRSAPrivateKeyState* pPrivateKey, const IppsRSAPublicKeyState* pPublicKeyOpt, const
IppsHashMethod* pMethod, Ipp8u* pBuffer);
```

Include Files

ippcp.h

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.
pMethod	Pointer to the hash method. For details, see HashMethod functions.
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* or *pMethod* 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.

NOTE
This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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);
```

```
IppStatus ippsRSAVerify_PKCS1v15_rmf(const Ipp8u* pMsg, int msgLen, const Ipp8u* pSign,
int* pIsSignValid, const IppsRSAPublicKeyState* pKey, const IppsHashMethod* pMethod,
Ipp8u* pBuffer);
```

Include Files

`ippcp.h`

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.
pMethod	Pointer to the hash method. For details, see HashMethod functions.
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* or *pMethod* parameter against the input message, as defined [PKCS 1.2.1].

> **NOTE**
> This function has a *reduced memory footprint* version. To learn more, see Reduced Memory Footprint 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 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.
`ippsStsNotSupportedModeErr`	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

Discrete-Logarithm-Based Cryptography Functions

This section introduces Intel® Integrated Performance Primitives (Intel® IPP) Cryptography 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`.
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`

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*≤*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

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*≤*reBits*.

See Also

Data Security Considerations

DLPPack, DLPUnpack

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

Syntax

```
IppStatus ippsDLPPack (const IppsDLPState* pCtx, Ipp8u* pBuffer);
IppStatus ippsDLPUnpack (const Ipp8u* pBuffer, IppsDLPState* pCtx);
```

Include Files

ippcp.h

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

```
IppStatus ippsDLPSet(const IppsBigNumState* pP, const IppsBigNumState* pQ, const
IppsBigNumState* pG, IppsDLPState* pCtx);
```

Include Files

ippcp.h

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*, *pR*, 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

```
IppStatus ippsDLPGet(IppsBigNumState* pP, IppsBigNumState* pQ, IppsBigNumState* pG,
IppsDLPState* pCtx);
```

Include Files

`ippcp.h`

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 *pP*, *pR*, and *pG* is too small for the DL parameter.

DLPSetDP

Sets up a particular domain parameter of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPSetDP(const IppsBigNumState* pDP, IppDLPKeyTag tag, IppsDLPState*
pCtx);
```

Include Files

`ippcp.h`

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.
`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 the Big Number specified by *pDP* is too big to be stored in the `IppsDLPState` context.
`ippStsBadArgErr`	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`

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.
`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.
`ippStsOutOfRangeErr`	Indicates an error condition if the Big Number specified by *pDP* is too small for the DL parameter.
`ippStsBadArgErr`	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

```
IppStatus ippsDLPGenKeyPair(IppsBigNumState* pPrivate, IppsBigNumState* pPublic,
IppsDLPState* pCtx, IppBitSupplier rndFunc, void* pRndParam);
```

Include Files

`ippcp.h`

Parameters

pPrivate	Pointer to the private key *privKey*.
pPublic	Pointer to the public key *pubKey*.
pCtx	Pointer to the cryptosystem context.
rndFunc	Specifed 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 DL-based cryptosystem. The function employs specified *rndFunc* Random Generator to generate a pseudorandom private key. The value of the private key *privKey* is a random number that lies in the range of [2,*R*-2].

Return Values

`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 *`pPrivate`* and *`pPublic`* is too small for the DL key.

DLPPublicKey

Computes a public key from the given private key of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPPublicKey(const IppsBigNumState* pPrivate, IppsBigNumState* pPublic,
IppsDLPState* pCtx);
```

Include Files

`ippcp.h`

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 computes a public key *pubKey* of the DL-based cryptosystem.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if the context parameter does not match the operation.
`ippStsIncompleteContextErr`	Indicates an error condition if the cryptosystem context has not been properly set up.
`ippStsInvalidPrivateKey`	Indicates an error condition if the *privKey* has an illegal value.
`ippStsRangeErr`	Indicates an error condition if Big Number specified by *`pPublic`* is too small for the DL public key.

DLPValidateKeyPair

Validates private and public keys of the DL-based cryptosystem over GF(p).

Syntax

```
IppStatus ippsDLPValidateKeyPair(const IppsBigNumState* pPrivate, const
IppsBigNumState* pPublic, IppDLResult* pResult, IppsDLPState* pCtx);
```

Include Files

`ippcp.h`

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`

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 *priveKey* 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`

Parameters

pSeedIn	Pointer to the input *Seed*.
nTrials	Security parameter specified for the Miller-Rabin probable primality.
pCtx	Pointer to the cryptosystem context.
pSeedOut	Pointer to the output *Seed* value (if requested).
pCounter	Pointer to the *counter* value (if requested).
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator 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

```
IppStatus ippsDLPValidateDSA(int nTrials, IppDLResult* pResult, IppsDLPState* pCtx,
IppBitSupplier rndFunc, void* pRndParam);
```

Include Files

`ippcp.h`

Parameters

nTrials	Security parameter specified for the Miller-Rabin probable primality.
pResult	Pointer to the validation result.
pCtx	Pointer to the cryptosystem context.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator 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.
`ippDLInvalidBaseRange`	$P \leq 2^{peBits-1}$ or $P \geq 2^{peBits}$.
`ippDLInvalidOrderRange`	$R \leq 2^{reBits-1}$ or $R \geq 2^{reBits}$.
`ippDLCompositeBase`	*P* is not a prime.
`ippDLCompositeOrder`	*R* is not a prime.
`ippDLInvalidCofactor`	*R* is not divisible by (*P* -1).
`ippDLInvalidGenerator`	(1 < *G* < (*P* -1)) is false or *G*^*R* != 1 (mod *P*).

To ensure that both *p* and *r* are primes, the function applies *`nTrial`*-round Miller-Rabin primality test. Test data for primality test is provided by the specified *`rndFunc`* Random Generator.

Return Values

`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

```
IppStatus ippsDLPSignDSA(const IppsBigNumState* pMsg, const IppsBigNumState* pPrivate,
IppsBigNumState* pSignR, IppsBigNumState* pSignS, IppsDLPState* pCtx);
```

Include Files

`ippcp.h`

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

```
IppStatus ippsDLPVerifyDSA(const IppsBigNumState* pMsg, const IppsBigNumState* pSignR,
const IppsBigNumState* pSignS, IppDLResult* pResult, IppsDLPState* pCtx);
```

Include Files

`ippcp.h`

Parameters

pMsg	Pointer to the message representation *msgRep*.
pSignR	Pointer to the signature *r*-component to be verified.
pSignS	Pointer to the signature *s*-component to be verified.
pResult	Pointer to the result of the verification.
pCtx	Pointer to the cryptosystem context.

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

```
//
      // known domain parameters
      //
      static const int M = 512; // DSA system bitsize
      static const int L = 160; // DSA order  bitsize

      static
      BigNumber P("0x8DF2A494492276AA3D25759BB06869CBEAC0D83AFB8D0CF7" \
                  "CBB8324F0D7882E5D0762FC5B7210EAFC2E9ADAC32AB7AAC" \
                  "49693DFBF83724C2EC0736EE31C80291");

      static
      BigNumber Q("0xC773218C737EC8EE993B4F2DED30F48EDACE915F");

      static
      BigNumber G("0x626D027839EA0A13413163A55B4CB500299D5522956CEFCB" \
                  "3BFF10F399CE2C2E71CB9DE5FA24BABF58E5B79521925C9C" \
                  "C42E9F6F464B088CC572AF53E6D78802");

      //
      // known DSA regular key pair
      //
      static
      BigNumber X("0x2070B3223DBA372FDE1C0FFC7B2E3B498B260614");


      static
      BigNumber Y("0x19131871D75B1612A819F29D78D1B0D7346F7AA77BB62A85" \
                  "9BFD6C5675DA9D212D3A36EF1672EF660B8C7C255CC0EC74" \
                  "858FBA33F44C06699630A76B030EE333");


      int DSAsign_verify_sample(void)
      {
         // DLP context
         IppsDLPState *DLPState = newDLP(M, L);

         // set up DLP crypto system
         ippsDLPSet(P, Q, G, DLPState);

        // message
         Ipp8u message[] = "abc";

         // compute message digest to be signed
         Ipp8u md[SHA1_DIGEST_LENGTH/8];
```

```
    ippsSHA1MessageDigest(message, sizeof(message)-1, md);
    BigNumber digest(0, BITS_2_WORDS(SHA1_DIGEST_LENGTH));
    ippsSetOctString_BN(md, SHA1_DIGEST_LENGTH/8, digest);

    // generate ephemeral key pair (ephX,ephY)
    BigNumber ephX(0, BITS_2_WORDS(L));
    BigNumber ephY(0, BITS_2_WORDS(M));


    IppsPRNGState* pRand = newPRNG();
    ippsDLPGenKeyPair(ephX, ephY, DLPState, ippsPRNGen, pRand);
    deletePRNG(pRand);
    //
    // generate signature
    //
    BigNumber signR(0, BITS_2_WORDS(L));      // R and S signature's component
    BigNumber signS(0, BITS_2_WORDS(L));
    ippsDLPSetKeyPair(ephX, ephY, DLPState);  // set up ephemeral keys
    ippsDLPSignDSA(digest, X,                 // sign digest
                   signR, signS,
                   DLPState);


    //
    // verify signature
    //
    ippsDLPSetKeyPair(0, Y, DLPState);        // set up regular public key
    IppDLResult result;
    ippsDLPVerifyDSA(digest, signR,signS,     // verify
                     &result, DLPState);

    // 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`

Parameters

pSeedIn	Pointer to the input *Seed*.
nTrials	Security parameter specified for the Miller-Rabin probable primality.

pCtx	Pointer to the cryptosystem context.
pSeedOut	Pointer to the output *Seed* value (if requested).
pCounter	Pointer to the *counter* value (if requested).
rndFunc	Specified Random Generator.
pRndParam	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.
`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 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).
`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.

DLPValidateDH

Validates domain parameters of the DL-based cryptosystem over GF(p) to use the DH Agreement scheme.

Syntax

```
IppStatus ippsDLPValidateDH(int nTrials, IppDLResult* pResult, IppsDLPState* pCtx,
IppBitSupplier rndFunc, void* pRndParam);
```

Include Files

`ippcp.h`

Parameters

nTrials	Security parameter specified for the Miller-Rabin probable primality.
pResult	Pointer to the validation result.
pCtx	Pointer to the cryptosystem context.

rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

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:

ippDLValid	Validation has passed successfully.
ippDLBaseIsEven	*P* is even.
ippDLOrderIsEven	*R* is even.
ippDLInvalidBaseRange	$P \leq 2^{peBits-1}$ or $P \geq 2^{peBits}$.
ippDLInvalidOrderRange	$R \leq 2^{reBits-1}$ or $R \geq 2^{reBits}$.
ippDLCompositeBase	*P* is not a prime.
ippDLCompositeOrder	*R* is not a prime.
ippDLInvalidCofactor	*R* is not divisible by (*P* -1).
ippDLInvalidGenerator	(1 < *G* < (*P* -1)) is false or *G*^*R* != 1 (mod *P*).

To ensure that both *p* and *r* are primes, the function applies *nTrial*-round Miller-Rabin primality test. Test data for primality test is provided by the specified *rndFunc* Random Generator.

Return Values

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

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

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:

default	"Unknown DL result"
ippDLValid	"Validation passed successfully"
ippDLBaseIsEven	"Base is even"
ippDLOrderIsEven	"Order is even"
ippDLInvalidBaseRange	"Invalid Base (*P*) range"
ippDLInvalidOrderRange	"Invalid Order (*R*) range"

`ippDLCompositeBase`	"Composite Base (*P*)"
`ippDLCompositeOrder`	"Composite Order (*R*)"
`ippDLInvalidCofactor`	"*R* does not divide (*P* -1)"
`ippDLInvalidGenerator`	"1 != *G*^*R* (mod *P*)"
`ippDLInvalidPrivateKey`	"Invalid Private Key"
`ippDLInvalidPublicKey`	"Invalid Public Key"
`ippDLInvalidKeyPair`	"Invalid Key Pair"
`ippDLInvalidSignature`	"Invalid Signature"

See Also

DLPValidateDH
DLPValidateDSA
DLPValidateKeyPair

Elliptic Curve Cryptography Functions

Cryptography Intel® Integrated Performance Primitives (Intel® IPP) Cryptography offers functions allowing for different operations with an elliptic curve defined over a prime finite field GF(*p*).

The functions are based on standards [IEEE P1363A], [SEC1], [ANSI], and [SM2].

Intel IPP Cryptography 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.

Functions Based on GF(*p*)

This section describes functions designed to specify the elliptic curve cryptosystem and perform various operations on the elliptic curve defined over a prime finite field. The examples of the operations are shown below:

- 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*).

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.

Standard Elliptic Curves

Value of `IppECCType`	Name of the Curve	Reference
`ippECarbitrary`	Not applicable	No reference because of arbitrary parameters.
`ippECstd112r1`	`secp112r1`	[SEC2]
`ippECstd112r2`	`secp112r2`	[SEC2]
`ippECstd128r1`	`secp128r1`	[SEC2]
`ippECstd128r2`	`secp128r2`	[SEC2]
`ippECstd160r1`	`secp160r1`	[SEC2]
`ippECstd160r2`	`secp160r2`	[SEC2]
`ippECstd192r1`	`secp192r1`	[SEC2]
`ippECstd224r1`	`secp224r1`	[SEC2]
`ippECstd256r1`	`secp256r1`	[SEC2]
`ippECstd384r1`	`secp384r1`	[SEC2]
`ippECstd521r1`	`secp521r1`	[SEC2]
`ippECstdSM2`	`SM2`	[SM2]

For more information on parameters recommended for the functions, see [SEC2] and [SM2].

Important
To provide minimum security of the elliptic curve cryptosystem over a prime finite field, the length of the underlying prime must be equal to or greater than 160 bits.

ECCPGetSize

Gets the size of the `IppsECCPState` *context.*

Syntax

```
IppStatus ippsECCPGetSize(int feBitSize, int *pSize);
```

Include Files

`ippcp.h`

Parameters

`feBitSize`	Size (in bits) of the underlying prime number.
`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.

NOTE
For security reasons, the length of the underlying prime number is restricted to 1 kilobit.

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.
`ippStsLengthErr`	Indicates an error condition if the value of the *`feBitsize`* parameter is less than 2 or greater than 1024.

ECCPGetSizeStd

Gets the size of the `IppsECCPState` *context for a standard elliptic curve.*

Syntax

```
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`

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`

Parameters

feBitSize	Size (in bits) of the underlying prime number.
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.

NOTE
For security reasons, the length of the underlying prime number is restricted to 1 kilobit.

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.
`ippStsLengthErr`	Indicates an error condition if the value of the *feBitsize* parameter is less than 2 or greater than 1024.

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 ippsECCPInitStd224r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd256r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd384r1(IppsECCPState* pECC);
IppStatus ippsECCPInitStd521r1(IppsECCPState* pECC);
```

```
IppStatus ippsECCPInitStdSM2(IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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

ECCPBindGxyTblStd

Enable the use of base point-based pre-computed tables of standard elliptic curves.

Syntax

```
IppStatus ippsECCPBinfGxyTblStd192r1(IppsECCPState* pEC);
IppStatus ippsECCPBinfGxyTblStd224r1(IppsECCPState* pEC);
IppStatus ippsECCPBinfGxyTblStd256r1(IppsECCPState* pEC);
IppStatus ippsECCPBinfGxyTblStd384r1(IppsECCPState* pEC);
IppStatus ippsECCPBinfGxyTblStd521r1(IppsECCPState* pEC);
IppStatus ippsECCPBinfGxyTblStdSM2(IppsECCPState* pEC);
```

Include Files

`ippcp.h`

Parameters

pEC	Pointer to the context of the elliptic curve

Description

The functions `ECCPValidate`, `ECCPGenKeyPair` and `ECCPVerify` perform time-consuming math operations on the elliptic curve base point. In Intel IPP Cryptography-supported standards, the base point is fixed, and you may use pre-computed values.

The function `ECCPBindGxyTbl` stores a pointer the to the pre-computed base point data in the elliptic curve context. For performance-critical applications, consider calling `ECCPBindGxyTbl` at the completion of elliptic curve initialization. The use of `ECCPBindGxyTbl` improves the performance of `ECCPValidate`, `ECCPGenKeyPair` and `ECCPVerify` up to 2 times.

NOTE
The size of the pre-computed table is quite large (~100-150KB), so using `ECCPBindGxyTbl` increases the size of your application.

Important
The set of `ECCPBindGxyTbl` functions covers only curves defined by the following standards: NIST P-192r1, NIST P-224r1, NIST P-256r1, NIST P-384r1, NIST P521r1, and SM2. Other standard elliptic curves supported in Intel IPP Cryptography do not have a similar mechanism because they do not match modern security strength requirements.

Return Values

`ippsStsNoErr`	Indicates no error. Any other message indicates an error or warning.
`ippsStsNullPtrErr`	Indicates an error condition if *`pEC`* is `NULL`.
`ippsStsContextMatchErr`	Indicates an error condition if the elliptic curve context is not valid.

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`

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:

  $y^2 = x^3 + A \cdot x + B \pmod{p}$.
- *pGX*, *pGY* are pointers to the affine coordinates of the elliptic curve base point *G*.
- *pOrder* is a pointer to the order *n* of the elliptic curve base point *G* such that $n \cdot G = O$, where *O* is the point at infinity and *n* is a prime number.
- *cofactor* sets up the ratio *h* of a general number of points #E on the elliptic curve (including the point at infinity) to the order *n* of the base point:

  $h = \#E/n$.

The domain parameters are set in the cryptosystem context which must be already created by the `ECCPGetSize` and `ECCPInit` functions.

Return Values

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 of 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 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

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 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

`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.
`ippStsECCInvalidFlagErr`	Indicates an error condition if the value of the parameter *flag* is not valid.

ECCPGet

Retrieves elliptic curve domain parameters over GF(p).

Syntax

```
IppStatus ippsECCPGet(IppsBigNumState* pPrime, IppsBigNumState* pA, IppsBigNumState*
pB, IppsBigNumState* pGX,IppsBigNumState* pGY, IppsBigNumState* pOrder, int* cofactor,
IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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`

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 *pBitsSize*. 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 the cryptosystem contextis 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

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 *pResult*.

Elliptic curve domain parameters must be hitherto defined by one of the functions: ECCPSet or ECCPSetStd. The purpose of the parameters *rndFunc*, *pRndParam*, and *nTrials* is analogous to that of the parameters *rndFunc*, *pRndParam*, and *nTrials* in the PrimeTest function.

The result of the elliptic curve domain parameters check can take one of the following values:

ippECValid	The parameters are valid.
ippECCompositeBase	The prime finite field characterisitc *p* is a composite number.
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.

`ippECPointIsNotValid`	The base point *G* is not on the elliptic curve.
`ippECCompositeOrder`	The order *n* of the base point *G* is a composite number.
`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.
`ippECIsWeakSSSA`	The order *n* of the base point *G* is equal to the finite field characteristic *p*.
`ippECIsWeakMOV`	The curve is excluded because it is subject to the MOV reduction attack.

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 *`c`* or *`pECC`* is not valid.
`ippStsBadArgErr`	Indicates an error condition if the memory size of the parameter *`seed`* is less than five words (32 bytes in each) or the value of the parameter *`nTrails`* is less than 1.

ECCPPointGetSize

Gets the size of the `IppsECCPPoint` *context in bytes for a point on the elliptic curve point defined over GF(p).*

Syntax

```
IppStatus ippsECCPPointGetSize(int feBitSize, int* pSize);
```

Include Files

`ippcp.h`

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`

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`

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`

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`

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: `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.

ECCPCheckPoint

Checks correctness of the point on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippsECCPCheckPoint(const IppsECCPPointState* pP, IppECResult* pResult,
IppsECCPState* pECC);
```

Include Files

`ippcp.h`

Parameters

pP	Pointer to the elliptic curve point.
pResult	Pointer to the result of the check.
pECC	Pointer to the context of the elliptic cryptosystem.

Description

The function checks the correctness of the point on the elliptic curve defined over a prime finite field GF(*p*) and allocates the result of the check in accordance with the pointer *pResult*.

The elliptic curve domain parameters must be hitherto defined by one of the functions: ECCPSet or ECCPSetStd.

The result of the check for the correctness of the point can take one of the following values:

ippECValid	Point is on the elliptic curve.
ippECPointIsNotValid	Point is not on the elliptic curve and is not the point at infinity.
ippECPointIsAtInfinite	Point is the point at infinity.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if one of the contexts pointed by *pP* or *pECC* is not valid.

ECCPComparePoint

Compares two points on the elliptic curve defined over GF(p).

Syntax

```
IppStatus ippsECCPComparePoint(const IppsECCPPointState* pP, const IppsECCPPointState* pQ, IppECResult* pResult, IppsECCPState* pECC);
```

Include Files

ippcp.h

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: ECCPSet 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 IppsECCPPointState* pP, IppsECCPPointState* pR,
IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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 IppsECCPPointState* pP, const IppsECCPPointState* pQ,
IppsECCPPointState* pR, IppsECCPState* pECC);
```

Include Files

`ippcp.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 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 GF(p).

Syntax

```
IppStatus ippsECCPMulPointScalar(const IppsECCPPointState* pP, const IppsBigNumState*
pK, IppsECCPPointState* pR, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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 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

```
IppStatus ippsECCPGenKeyPair(IppsBigNumState* pPrivate, IppsECCPPointState* pPublic,
IppsECCPState* pECC, IppBitSupplier rndFunc, void* pRndParam);
```

Include Files

`ippcp.h`

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

```
IppStatus ippsECCPPublicKey(const IppsBigNumState* pPrivate, IppsECCPPointState*
pPublic, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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.
ippStsContextMatchErr	Indicates an error condition if one of the contexts pointed by *pPrivate*, *pPublic*, or *pECC* is not valid.
ippStsIvalidPrivateKey	Indicates an error condition if the value of the private key falls outside the range of [1, n-1].

ECCPValidateKeyPair

Validates private and public keys of the elliptic cryptosystem over GF(p).

Syntax

```
IppStatus ippsECCPValidateKeyPair(const IppsBigNumState* pPrivate, const
IppsECCPPointState* pPublic, IppECResult* pResult, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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.
`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.

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`

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:

`ippTrue`	Keys are regular.
`ippFalse`	Keys are ephemeral.

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.

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`

Parameters

pPrivate	Pointer to your own private key *privKey*.
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`

Parameters

pPrivate	Pointer to your own private key *privKey*.
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*: $pubKeyA = privKeyA \cdot G$, where *G* is the base point of the elliptic curve. Alice passes the public key to Bob.
2. Bob calculates his own public key *pubKeyB* by using his private key *privKeyB*: $pubKeyB = privKeyB \cdot G$, where *G* is a base point of the elliptic curve. Bob passes the public key to Alice.
3. Alice gets Bob's public key and calculates the secret point *shareA*. When calculating, she uses her own private key and Bob's public key and applies the following formula: $shareA = h \cdot privKeyA \cdot pubKeyB = h \cdot privKeyA \cdot privKeyB \cdot G$, where *h* is the elliptic curve cofactor.
4. Bob gets Alice's public key and calculates the secret point *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: $shareB = h \cdot privKeyB \cdot pubKeyA = h \cdot privKeyB \cdot privKeyA \cdot G$, where *h* is the elliptic curve cofactor.

Shared secret *bnShare* is an x-coordinate of the secret point on the elliptic curve.

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.)

ECCPSignDSA

Computes a digital signature over a message digest.

Syntax

```
IppStatus ippsECCPSignDSA(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pPrivate, IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

Parameters

pMsgDigest	Pointer to the message digest *msg* to be digitally signed, that is, to be ecrypted 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 the ECDSA scheme, an elliptic curve analogue of the DSA scheme. ECDSA assumes that the following keys are hitherto set by a message signer:

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*.

`ippStsEphemeralKeyErr`	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).

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

```
IppStatus ippsECCPVerifyDSA(const IppsBigNumState* pMsgDigest, const IppsBigNumState* pSignX, const IppsBigNumState* pSignY, IppECResult* pResult, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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

```
IppStatus ippsECCPSignNR(const IppsBigNumState* pMsgDigest, const IppsBigNumState* pPrivate, IppsBigNumState* pSignX, IppsBigNumState* pSignY, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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*.
`ippStsEphemeralKeyErr`	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

```
IppStatus ippsECCPVerifyNR(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pSignX, const IppsBigNumState* pSignY, IppECResult* pResult, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

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

```
IppStatus ippsECCPSignSM2(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pRegPrivate, const IppsBigNumState* pEphPrivate, IppsBigNumState* pSignR,
IppsBigNumState* pSignS, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

Parameters

`pMsgDigest`	Pointer to the message digest *msg*.
`pRegPrivate`	Pointer to the regular private key *regPrivKey*.
`pEphPrivate`	Pointer to the ephmeral 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 `ECCPGenKeyPair` and `ECCPSetKeyPair` functions with the only requirement that the key *regPrivKey* is different from the key *ephPrivKey*.

Before calling `ECCPSignSM2`, set up the domain parameters of the elliptic curve in the `*pECC` context by calling one of the functions: `ECCPSet` or `ECCPSetStdSM2`.

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 specified contexts 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 `pSignR` or `pSignS` has a smaller 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 *ephPrivKey* and *ephPubKey* are not valid. (Either $r = 0$ or $s = 0$ is received as a result of the digital signature calculation).

ECCPVerifySM2

Verifies authenticity of a digital signature over a message digest using the SM2 scheme.

Syntax

```
IppStatus ippsECCPVerifySM2(const IppsBigNumState* pMsgDigest, const
IppsECCPPointState* pRegPublic, const IppsBigNumState* pSignR, const IppsBigNumState*
pSignS, IppECResult* pResult, IppsECCPState* pECC);
```

Include Files

`ippcp.h`

Parameters

`pMsgDigest`	Pointer to the message digest *msg*.
`pRegPublic`	Pointer to the message sender's regular private key *regPubKey*.
`pSignR`	Pointer to the integer *r* of the digital signature.
`pSignS`	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, represented as integer big numbers *r* and *s*, over a message digest *msg*. The digital signature over the message digest *msg* must be computed using the SM2 scheme [SM2] by to the ECCPSignSM2 function.

The scheme requires the following cryptosystem key set up by the message sender:

regPubKey	Message sender's regular private key.

You can generate and set up the key in a call to the `ECCPGenKeyPair` function.

The result of the digital signature verification can take one of these values:

`ippECValid`	The digital signature is valid.
`ippECInvalidSignature`	The digital signature is not valid.

Before calling `ECCPVerifySM2`, set up the domain parameters of the elliptic curve in the `*pECC` context by calling one of the functions: `ECCPSet` or `ECCPSetStdSM2`.

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 specified contexts 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*.

Signing/Verification Using the Elliptic Curve Cryptography Functions over a Prime Finite Field

Use of ECCPSignDSA, ECCPVerifyDSA

Arithmetic of the Group of Elliptic Curve Points

This section describes the Intel IPP functions that implement arithmetic operations with points of elliptic curves [EC]. The elliptic curve is defined by the following equation:

$y^2 = x^3 + A \cdot x + B$

where

- *A* and *B* are the parameters of the curve
- *x* and *y* are the coordinates of a point on the curve

This document considers elliptic curves constructed over the finite field GF(*p*) (prime or its extension), therefore the arithmetic of elliptic curves is based on the arithmetic of the underlying finite field. In the equation above, *A, B, x,* and *y* belong to the underlying field GF(*p*).

You can use standard elliptic curves by calling `GFpECInitStd` or `GFpECBindGxyTblStd`. The following table contains the supported standard elliptic curves:

Standard Elliptic Curves

Name of the Curve	Reference
`secp128r1`	[SEC2]
`secp128r2`	[SEC2]
`secp160r1`	[SEC2]

Name of the Curve	Reference
secp160r2	[SEC2]
secp192r1	[SEC2]
secp224r1	[SEC2]
secp256r1	[SEC2]
secp384r1	[SEC2]
secp521r1	[SEC2]
SM2	[SM2]
BN256	[ISO/IEC 11889-4]

For more information on parameters of the standard elliptic curves, see [SEC2], [SM2], and [ISO/IEC 11889-4].

NOTE
In this table, the name `BN256` corresponds to the Barreto-Naehrig Prime 256-bit elliptic curve.

Important
To provide minimum security of the elliptic curve cryptosystem over a prime finite field, the length of the underlying prime must be equal to or greater than 160 bits.

GFpECGetSize

Gets the size of an elliptic curve over the finite field.

Syntax

```
IppStatus ippsGFpECGetSize(const IppsGFpState* pGF, int* pCtxSizeInBytes);
```

Include Files

`ippcp.h`

Parameters

pGF	Pointer to the `IppsGFpState` context of the underlying finite field.
pCtxSizeInBytes	Buffer size in bytes needed for the `IppsGFpECState` context.

Description

This function returns the size of the buffer associated with the `IppsGFpECState` context, suitable for storing data for the elliptic curve over the finite field specified by the context *pGF*.

Return Values

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

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

GFpECInit

Initializes the context of an elliptic curve over a finite field.

Syntax

```
IppStatus ippsGFpECInit(const IppsGFpState* pGF, const IppsGFpElement* pA, const
IppsGFpElement* pB, IppsGFpECState* pEC);
```

Include Files

ippcp.h

Parameters

pGF	Pointer to the IppsGFpState context of the underlying finite field.
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.
pEC	Pointer to the context of the elliptic curve being initialized.

Description

This function initializes the memory buffer *pEC* associated with the IppsGFpECState context and sets up the parameters of the elliptic curve if they are supplied. The initialized context is used in functions that create contexts of points on the curve (elements of the group of points) and perform operations with the points.

NOTE
Only the *pEC* and *pGF* parameters are required. You can omit the other parameters by setting their values to NULL or zero and set them up later on by calling GFpECSet or GFpECSetSubGroup.

NOTE
When calling arithmetic functions for the elliptic curve defined by *pEC*, a properly initialized *pGF* context of the underlying field is required.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if either *pEC* or *pGF* is NULL.
ippStsContextMatchErr	Indicates an error condition in the following cases: • IppsGFpState context parameter does not match the operation. • *pA* or *pB* is not zero and the corresponding context parameter does not match the operation.

GFpECSet

Sets up the parameters of an elliptic curve over a finite field.

Syntax

```
IppStatus ippsGFpECSet(const IppsGFpElement* pA, const IppsGFpElement* pB,
IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

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.
pEC	Pointer to the context of the elliptic curve.

Description

This function assigns input values to the parameters of the elliptic curve in the `IppsGFpECState` context, if they are supplied.

> **NOTE**
> Only the *pEC* parameter is required. You can omit the other parameters by setting their values to `NULL` or zero.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if *pEC* is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition in the following cases: • `IppsGFpECState` context parameter does not match the operation. • *pA* or *pB* is not zero, and the corresponding context parameter does not match the operation.

GFpECSetSubgroup

Sets up the parameters defining an elliptic curve points subgroup.

Syntax

```
IppStatus ippsGFpECSetSubGroup(const IppsGFpElement* pX, const IppsGFpElement* pY,
const IppsBigNumState* pOrder, const IppsBigNumState* pCofactor, IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pX, *pY*	Pointers to the *X* and *Y* coordinates of the base point of the elliptic curve.
pOrder	Pointer to the big number context storing the order of the base point.
pCofactor	Pointer to the big number context storing the cofactor.
pEC	Pointer to the context of the elliptic curve.

Description

This function sets up an elliptic curve as the subgroup generated by the base point over the finite field.

NOTE
Only the *pEC* parameter is required. You can omit the other parameters by setting their values to `NULL` or zero.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if *pEC* is `NULL`.
ippStsContextMatchErr	Indicates an error condition in the following cases: • `IppsGFpECState` context parameter does not match the operation. • Any of the pointers to elliptic curve parameters is not zero and the context parameter does not match the operation.
ippStsBadArgErr	Indicates an error condition if any of the specified `IppsBigNumState` contexts defines zero or a negative number.
ippStsOutOfRangeErr	Indicates an error if the base point coordinates (*pX*, *pY*) do not belong to the finite field over which the elliptic curve is initialized.
ippStsRangeErr	Indicates an error condition in the following cases: • The size of the base point order exceeds the maximal size of the order for the given curve. • The bit size of the cofactor exceeds the bit size of the element of the finite field over which the elliptic curve is initialized.

GFpECInitStd

Initializes the context of a standard elliptic curve over a finite field

Syntax

```
IppStatus ippsGFpECInitStd128r1(const IppsGFpState* pGF, IppsGFpECState* pEC);
IppStatus ippsGFpECInitStd128r2(const IppsGFpState* pGF, IppsGFpECState* pEC);
IppStatus ippsGFpECInitStd192r1(const IppsGFpState* pGF, IppsGFpECState* pEC);
IppStatus ippsGFpECInitStd224r1(const IppsGFpState* pGF, IppsGFpECState* pEC);
IppStatus ippsGFpECInitStd256r1(const IppsGFpState* pGF, IppsGFpECState* pEC);
```

```
IppStatus ippsGFpECInitStd384r1(const IppsGFpState* pGF, IppsGFpECState* pEC);
IppStatus ippsGFpECInitStd521r1(const IppsGFpState* pGF, IppsGFpECState* pEC);
IppStatus ippsGFpECInitStdSM2(const IppsGFpState* pGF, IppsGFpECState* pEC);
IppStatus ippsGFpECInitStdBN256(const IppsGFpState* pGF, IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pGF	Pointer to the `IppsGFpState` context of the underlying finite field.
pEC	Pointer to the context of the elliptic curve being initialized.

Description

This function initializes the memory buffer *pEC* associated with the `IppsGFpECState` context and sets up the parameters of 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 *pEC* is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if the `IppsGFpState` context parameter does not match the operation.
`ippStsBadArgErr`	Indicates an error condition if the `IppsGFpState` context parameter does not specify the finite field over which the given standard elliptic curve is defined.

GFpECBindGxyTblStd

Enables the use of base point-based pre-computed tables of standard elliptic curves.

Syntax

```
IppStatus ippsGFpECBindGxyTblStd192r1(IppsGFpECState* pEC);
IppStatus ippsGFpECBindGxyTblStd224r1(IppsGFpECState* pEC);
IppStatus ippsGFpECBindGxyTblStd256r1(IppsGFpECState* pEC);
IppStatus ippsGFpECBindGxyTblStd384r1(IppsGFpECState* pEC);
IppStatus ippsGFpECBindGxyTblStd521r1(IppsGFpECState* pEC);
IppStatus ippsGFpECBindGxyTblStdSM2(IppsGFpECState* pEC);
IppStatus ippsGFpECBindGxyTblStdBN256(IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pEC	Pointer to the context of the elliptic curve.

Description

The functions `GFpECVerify*`, `GFpECPublicKey` and `GFpECSign*` perform time-consuming math operations on the elliptic curve base point. In Intel IPP Cryptography-supported standards, the base point is fixed, and you may use pre-computed values.

The function `GFpECBindGxyTblStd` stores a pointer the to the pre-computed base point data in the elliptic curve context. For performance-critical applications, consider calling `GFpECBindGxyTblStd` at the completion of elliptic curve initialization. The use of `GFpECBindGxyTblStd` improves the performance of `GFpECVerify*`, `GFpECPublicKey` and `GFpECSign*`.

NOTE
The size of the pre-computed table is quite large (~100-150KB), so using `GFpECBindGxyTblStd` increases the size of your application.

Important
The set of `GFpECBindGxyTblStd` functions covers only curves defined by the following standards: NIST P-192r1, NIST P-224r1, NIST P-256r1, NIST P-384r1, NIST P521r1, SM2 and BN256. Other standard elliptic curves supported in Intel IPP Cryptography do not have a similar mechanism because they do not match modern security strength requirements.

Return Values

`ippsStsNoErr`	Indicates no error. Any other message indicates an error or warning.
`ippsStsNullPtrErr`	Indicates an error condition if *pEC* is `NULL`.
`ippsStsContextMatchErr`	Indicates an error condition if the *IppsGFpECState* context parameter does not match the operation.
`ippStsBadArgErr`	Indicates an error condition if the elliptic curve specified by the *IppsGFpECState* context is not the target standard elliptic curve.

GFpECGet

Extracts the parameters of an elliptic curve over a finite field from the context.

Syntax

```
IppStatus ippsGFpECGet(IppsGFpState** const ppGF, IppsGFpElement* pA, IppsGFpElement*
pB, const IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

ppGF	Pointer to the context of the elliptic curve underlying finite field.
pA	Pointer to a copy of the coefficient *A* of the equation defining the elliptic curve.
pB	Pointer to a copy of the coefficient *B* of the equation defining the elliptic curve.

pEC	Pointer to the context of the elliptic curve.

Description

This function extracts parameters of the elliptic curve from the input `IppsGFpECState` context. You can get any combination of the following parameters: a reference to the underlying field and copies of the *A* and *B* coefficients. To turn off extraction of a particular parameter of the elliptic curve, 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 *pEC* is `NULL`.

`ippStsContextMatchErr` — Indicates an error condition in the following cases:

- `IppsGFpECState` context parameter does not match the operation.
- Either *pA* or *pB* is not zero and the corresponding context parameter does not match the operation.

`ippStsOutOfRangeErr` — Indicates an error if either *pA* or *pB* does not belong to the finite field over which the elliptic curve is initialized.

GFpECGetSubgroup

Extracts the parameters (base point and its order) that define an elliptic curve point subgroup.

Syntax

```
IppStatus ippsGFpECGetSubGroup(IppsGFpState** const ppGF, IppsGFpElement* pX,
IppsGFpElement* pY, IppsBigNumState* pOrder,IppsBigNumState* pCofactor, const
IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

ppGF	Pointer to the context of the underlying finite field.
pX, *pY*	Pointers to the *X* and *Y* coordinates of the base point of the elliptic curve.
pOrder	Pointer to the big number context storing the order of the base point.
pCofactor	Pointer to the big number context storing the cofactor.
pEC	Pointer to the context of the elliptic curve.

Description

This function extracts parameters of an elliptic curve subgroup. You can get any combination of the following parameters: the *X* and *Y* coordinates, the order of the base point, and the value of the cofactor. To turn off extraction of a particular parameter of the elliptic curve, set the appropriate function parameter to `NULL`.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if the specified pointer *`pEC`* is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition in the following cases: • `IppsGFpECState` context parameter does not match the operation. • Any of the pointers to elliptic curve parameters is not zero and the corresponding context parameter does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error if the base point coordinates (*`pX`*, *`pY`*) do not belong to the finite field over which the elliptic curve is initialized.
`ippStsLengthErr`	Indicates an error condition in the following cases: • The size of the base point order exceeds the maximal size of the order for the given curve. • The bit size of the cofactor exceeds the bit size of the element of the finite field over which the elliptic curve is initialized.

GFpECScratchBufferSize

Gets the size of the scratch buffer.

Syntax

```
IppStatus ippsGFpECScratchBufferSize(int nScalars, const IppsGFpECState* pEC, int* pBufferSize);
```

Include Files

`ippcp.h`

Parameters

`nScalars`	Number of scalar values. This may take the following values: • Number of scalar values used in the multiplication operation. • 1 if it is not applicable.
`pEC`	Pointer to the context of the elliptic curve.
`pBufferSize`	Pointer to the calculated buffer size in bytes.

Description

This function computes the size of the scratch buffer for functions that require an external scratch 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`.
`ippStsContextMatchErr`	Indicates an error condition if the `IppsGFpECState` context parameter does not match the operation.

GFpECVerify

Verifies the parameters of an elliptic curve.

Syntax

```
IppStatus ippsGFpECVerify(IppECResult* pResult, IppsGFpECState* pEC, Ipp8u*
pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

pResult	Pointer to the verification result.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

Description

This function verifies the parameters of the elliptic curve from the input `IppsGFpECState` context and returns the result in *pResult*. The result of the verification may have the following values:

`ippECValid`	Parameters are valid.
`ippECIsZeroDiscriminant`	$4 \cdot A^3 + 3 \cdot B^2 = 0$.
`ippECPointIsAtInfinity`	Base point $G = (x, y)$ is a point at infinity.
`ippECPointIsNotValid`	Base point $G = (x, y)$ does not belong to the curve.
`ippECInvalidOrder`	Order of the base point $G = (x, y)$ is invalid.

If the pointer to the scratch buffer is `NULL`, the function uses a short internal buffer for computations.

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 `IppsGFpECState` context parameter does not match the operation.

GFpECPointGetSize

Gets the size of the `IppsGFpECPoint` *context of a point on an elliptic curve.*

Syntax

```
IppStatus ippsGFpECPointGetSize(const IppsGFpECState* pEC, int* pSizeInBytes);
```

Include Files

`ippcp.h`

Parameters

pEC	Pointer to the context of the elliptic curve.
pSizeInBytes	Buffer size, in bytes, needed for the `IppsGFpECPoint` context.

Description

This function returns the size of the buffer associated with the `IppsGFpECPoint` context, which you may use to store data for a point on the elliptic curve over the finite field.

Return Values

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

GFpECPointInit

Initializes the context of a point on an elliptic curve.

Syntax

```
IppStatus ippsGFpECPointInit(const IppsGFpElement* pX, const IppsGFpElement* pY,
IppsGFpECPoint* pPoint, IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pX, *pY*	Pointers to the *X* and *Y* coordinates of a point on the elliptic curve.
pPoint	Pointer to the `IppsGFpECPoint` context being initialized.
pEC	Pointer to the context of the elliptic curve.

Description

This function initializes the `IppsGFpECPoint` context and sets the coordinates of an elliptic curve point to the values stored in *pX* and *pY*. If any of the pointers to the *X* and *Y* coordinates is zero, the function sets the coordinates of the elliptic curve point in the `IppsGFpECPoint` context to the coordinates of a point at infinity.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if either *pPoint* or *pEC* is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition in the following cases: • `IppsGFpECState` context parameter does not match the operation. • Neither of the pointers to the *X* and *Y* coordinates is zero, and any of the corresponding context parameters does not match the operation.

`ippStsOutOfRangeErr`	Indicates an error if the point coordinates (*pX*, *pY*) do not belong to the finite field over which the elliptic curve is initialized.

GFpECSetPointAtInfinity

Sets a point on an elliptic curve as a point at infinity.

Syntax

```
IppStatus ippsGFpECSetPointAtInfinity(IppsGFpECPoint* pPoint, IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pPoint	Pointer to the `IppsGFpECPoint` context.
pEC	Pointer to the context of the elliptic curve.

Description

This function sets the coordinates of an elliptic curve point in the `IppsGFpECPoint` context to the coordinates of a point at infinity.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if *pPoint* or *pEC* is `NULL`.

GFpECSetPoint

Sets up the coordinates of a point on an elliptic curve.

Syntax

```
IppStatus ippsGFpECSetPoint(const IppsGFpElement* pX, const IppsGFpElement* pY,
IppsGFpECPoint* pPoint, IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pX, *pY*	Pointers to the *X* and *Y* coordinates of the point on the elliptic curve.
pPoint	Pointer to the `IppsGFpECPoint` context.
pEC	Pointer to the context of the elliptic curve.

Description

This function sets up the coordinates of a point on the elliptic curve over the finite field.

Return Values

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

ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if any of the specified contexts does not match the operation.
ippStsOutOfRangeErr	Indicates an error if the point coordinates (*pX*, *pY*) do not belong to the finite field over which the elliptic curve is initialized.

GFpECSetPointRandom

Sets the coordinates of a point on an elliptic curve to random values.

Syntax

```
IppStatus ippsGFpECSetPointRandom(IppsGFpECPoint* pPoint, IppsGFpECState* pEC,
IppBitSupplier rndFunc, void* pRndParam, Ipp8u* pScratchBuffer);
```

Include Files

ippcp.h

Parameters

pPoint	Pointer to the IppsGFpECPoint context.
pEC	Pointer to the context of the elliptic curve.
rndFunc	Pesudorandom number generator.
pRndParam	Pointer to the pseudorandom number generator context.
pScratchBuffer	Pointer to the scratch buffer.

Description

This function assigns random values to the coordinates of an elliptic curve point in the IppsGFpECPoint context.

If the pointer to the scratch buffer is NULL, the function uses a short internal buffer for computations.

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.
ippStsOutOfRangeErr	Indicates an error if the specified point does not belong to the finite field over which the elliptic curve is initialized.

GFpECMakePoint

Constructs the coordinates of a point on an elliptic curve based on the X-coordinate.

Syntax

```
IppStatus ippsGFpECMakePoint(const IppsGFpElement* pX, IppsGFpECPoint* pPoint,
IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pX	Pointer to the *X*-coordinate of the point on the elliptic curve.
pPoint	Pointer to the `IppsGFpECPoint` context.
pEC	Pointer to the context of the elliptic curve.

Description

This function computes the coordinates of a point on an elliptic curve based on the *X*-coordinate.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if any of the specified contexts does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition in the following cases: • The coordinates of the point *pPoint* do not belong to the finite field over which the elliptic curve is initialized. • The point coordinate *pX* does not belong to the finite field over which the elliptic curve is initialized.
`ippStsBadArgErr`	Indicates an error condition if the finite field over which the elliptic curve is initialized is not prime.
`ippStsQuadraticNonResidueErr`	Indicates an error condition if the square of the *Y*-coordinate of the point is a quadratic non-residue modulo *p*.

GFpECSetPointHash

Constructs a point on an elliptic curve based on the hash of the input message.

Syntax

```
IppStatus ippsGFpECSetPointHash(Ipp32u hdr, const Ipp8u* pMsg, int msgLen,
IppsGFpECPoint* pPoint, IppsGFpECState* pEC, IppHashAlgId hashID, Ipp8u*
pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

hdr	Header of the input message.
pMsg	Pointer to the input message.
msgLen	Length of the input message.
pPoint	Pointer to the `IppsGFpECPoint` context.

pEC	Pointer to the context of the elliptic curve.
hashID	ID of the hash algorithm used. For details, see Supported Hash Algorithms.
pScratchBuffer	Pointer to the scratch buffer. Can be `NULL`.

Description

This function makes the coordinates of a point on the elliptic curve over the finite field from a hash of the *X*-coordinate. If the pointer to the scratch buffer is `NULL`, the function uses a short internal buffer for computations.

The *X*-coordinate is computed by the following pseudocode formula: `X = hash(hdr || message)`.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition in the following cases: • *pPoint* or *pEC* is `NULL`. • Length of the message is more than zero, and the pointer *pMsg* is `NULL`.
ippStsContextMatchErr	Indicates an error condition if either *pPoint* or *pEC* context parameter does not match the operation.
ippStsBadArgErr	Indicates an error condition if the finite field over which the elliptic curve is initialized is not prime.
ippStsOutOfRangeErr	Indicates an error condition if the coordinates of the point *pPoint* do not belong to the finite field over which the elliptic curve is initialized.
ippStsLengthErr	Indicates an error condition if *msgLen* is negative.
ippStsQuadraticNonResidueErr	Indicates an error condition if the square of the *Y*-coordinate of the point is a quadratic non-residue modulo *p*.

GFpECGetPoint

Retrieves coordinates of a point on an elliptic curve.

Syntax

```
IppStatus ippsGFpECGetPoint(const IppsGFpECPoint* pPoint, IppsGFpElement* pX,
IppsGFpElement* pY, IppsGFpECState* pEC);
```

Include Files

ippcp.h

Parameters

pPoint	Pointer to the `IppsGFpECPoint` context.
pX, *pY*	Pointers to the *X* and *Y* coordinates of a point on the elliptic curve.
pEC	Pointer to the context of the elliptic curve.

Description

This function exports the coordinates of an elliptic curve point from the `IppsGFPECPoint` context to the user-defined elements of the underlying field. To turn off the extraction of a particular coordinate, 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 *pPoint* or *pEC* is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if any of the specified contexts does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition in the following cases: • The coordinates of the point *pPoint* do not belong to the underlying finite field of the elliptic curve. • *pX* or *pY* does not belong to the underlying finite field of the elliptic curve.
`ippStsPointAtInfinity`	Indicates an error condition if the specified point is a point at infinity.

GFpECGetPointRegular

Retrieves coordinates of a point on an elliptic curve in the regular domain.

Syntax

```
IppStatus ippsGFpECGetPointRegular(const IppsGFpECPoint* pPoint, IppsBigNumState* pX,
IppsBigNumState* pY, IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pPoint	Pointer to the `IppsGFpECPoint` context.
pX, *pY*	Pointers to the X and Y coordinates of a point on the elliptic curve.
pEC	Pointer to the context of the elliptic curve.

Description

This function exports the coordinates of an elliptic curve point from the `IppsGFpECPoint` context to the big number values *pX* and *pY*. To turn off the extraction of a particular coordinate, 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 *pPoint* or *pEC* is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if any of the specified contexts does not match the operation.

ippStsOutOfRangeErr	Indicates an error condition if the coordinates of the point *pPoint* do not belong to the underlying finite field of the elliptic curve.
ippStsPointAtInfinity	Indicates an error condition if the specified point is the point at infinity.

GFpECTstPoint

Checks if a point belongs to an elliptic curve.

Syntax

```
IppStatus ippsGFpECTstPoint(const IppsGFpECPoint* pP, IppECResult* pResult,
IppsGFpECState* pEC);
```

Include Files

ippcp.h

Parameters

pP	Pointer to the IppsGFpECPoint context.
pResult	Pointer to the result of the check.
pEC	Pointer to the context of the elliptic curve.

Description

This function checks whether the given point belongs to the elliptic curve over the finite field. The result of the testing is returned in *pResult* and may have the following values:

ippECValid	The point belongs to the curve.
ippECPointIsAtInfinite	The point is a point at infinity.
ippECPointIsNotValid	The point does not belong to the curve.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if any of the specified contexts does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if the coordinates of the point *pP* do not belong to the finite field over which the elliptic curve is initialized.

GFpECTstPointInSubgroup

Checks if a point belongs to a specified subgroup.

Syntax

```
IppStatus ippsGFpECTstPointInGroup(const IppsGFpECPoint* pP, IppECResult* pResult,
IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

pP	Pointer to the `IppsGFpECPoint` context.
pResult	Pointer to the result received upon the check that the point belongs to the elliptic curve over the finite field.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer; can be `NULL`.

Description

This function checks whether a point belongs to the pre-defined subgroup of the elliptic curve defined over the finite field. The result of the testing is returned in *pResult* and may have the following values:

`ippECValid`	The point is in the subgroup of the curve.
`ippECPointOutOfGroup`	The point is out of the subgroup.

If the pointer to the scratch buffer is NULL, the function uses a short internal buffer for computations.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the pointers *pP*, *pResult*, and *pEC* is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if any of the specified contexts does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if the point does not belong to the finite field over which the elliptic curve is initialized.

GFpECCpyPoint

Copies one point to another.

Syntax

```
IppStatus ippsGFpECCpyPoint(const IppsGFpECPoint* pA, IppsGFpECPoint* pR,
IppsGFpECState* pEC);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the elliptic curve point being copied.
pR	Pointer to the context of the elliptic curve point being changed.
pEC	Pointer to the context of the elliptic curve.

Description

This function copies one point of the elliptic curve over the finite field 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 is NULL.
ippStsContextMatchErr	Indicates an error condition if any of the specified contexts does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if any of the specified points does not belong to the finite field over which the elliptic curve is initialized.

GFpECCmpPoint

Compares two points.

Syntax

```
IppStatus ippsGFpECCmpPoint(const IppsGFpECPoint* pP, const IppsGFpECPoint* pQ,
IppECResult* pResult, IppsGFpECState* pEC);
```

Include Files

ippcp.h

Parameters

pA	Pointer to the context of the first elliptic curve point.
pQ	Pointer to the context of the second elliptic curve point.
pResult	Pointer to the result of the comparison.
pEC	Pointer to the context of the elliptic curve.

Description

This function compares the coordinates of two points on the elliptic curve over the finite field and returns the result in *pResult*. The result of the comparison may have the following values:

ippECPointIsEqual	The points are equal.
ippECPointIsNotEqual	The points are not equal.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if any of the specified contexts does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if any of the points does not belong to the finite field over which the elliptic curve is initialized.

GFpECNegPoint

Computes the inverse of a point.

Syntax

```
IppStatus ippsGFpECNegPoint(const IppsGFpECPoint* pP, IppsGFpECPoint* pR,
IppsGFpECState* pEC);
```

Include Files

ippcp.h

Parameters

pP	Pointer to the context of the given point on the elliptic curve.
pR	Pointer to the context of the resulting point on the elliptic curve.
pEC	Pointer to the context of the elliptic curve.

Description

For a given point of the elliptic curve over the finite field, this function computes the coordinates of the inverse point. The following pseudocode represents this operation: *R = 0 - 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 is NULL.
ippStsContextMatchErr	Indicates an error condition if any of the specified contexts does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if any of the specified points does not belong to the finite field over which the elliptic curve is initialized.

GFpECAddPoint

Computes the sum of two points on an elliptic curve.

Syntax

```
IppStatus ippsGFpECAddPoint(const IppsGFpECPoint* pP, const IppsGFpECPoint* pQ,
IppsGFpECPoint* pR, IppsGFpECState* pEC);
```

Include Files

ippcp.h

Parameters

pA	Pointer to the context of the first point on the elliptic curve to be added.
pQ	Pointer to the context of the second point on the elliptic curve to be added.
pR	Pointer to the context of the resulting point on the elliptic curve.
pEC	Pointer to the context of the elliptic curve.

Description

This function computes the coordinates of the elliptic curve point that is equal to the sum of two given points. The following pseudocode represents this operation: *R = P + Q*.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified is NULL.
ippStsContextMatchErr	Indicates an error condition if any of the specified contexts does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if any of the specified points does not belong to the finite field over which the elliptic curve is initialized.

GFpECMulPoint

Multiplies a point on an elliptic curve by a scalar.

Syntax

```
IppStatus ippsGFpECMulPoint(const IppsGFpECPoint* pP, const IppsBigNumState* pN,
IppsGFpECPoint* pR, IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

ippcp.h

Parameters

pP	Pointer to the context of the given point on the elliptic curve.
pN	Pointer to the Big Number context storing the scalar value.
pR	Pointer to the context of the resulting point on the elliptic curve.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer. Can be NULL.

Description

This function computes the coordinates of the elliptic curve point that equals the product of the given point and a scalar. The following pseudocode represents this operation: $R = scalar \cdot P$.

If the pointer to the scratch buffer is NULL, the function uses a short internal buffer for computations.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if any of the specified contexts does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition in the following cases: • Any of the points does not belong to the finite field over which the elliptic curve is initialized. • The scalar value does not belong to the finite field over which the elliptic curve is initialized.

GFpECPrivateKey

Generates a private key of the elliptic curve cryptosystem over GF(p).

Syntax

```
IppStatus ippsGFpECPrivateKey(IppsBigNumState* pPrivate, IppsGFpECState* pEC,
IppBitSupplier rndFunc, void* pRndParam);
```

Include Files

`ippcp.h`

Parameters

pPrivate	Pointer to the private key *privKey*.
pEC	Pointer to the context of the elliptic curve.
rndFunc	Specified Random Generator.
pRndParam	Pointer to the Random Generator context.

Description

The function generates a private key *privKey* 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 memory size of the parameter *privKey* pointed to by *pPrivate* must be not less than order of the base point, which can also be defined by the function GFpECGetSubgroup.

The elliptic curve domain parameters must be hitherto defined by the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if any of the specified contexts does not match the operation.
`ippStsSizeErr`	Indicates an error condition if the parameter pointed to by *pPrivate* has a memory size that is less than the order *n* of the elliptic curve base point *G*.

GFpECPublicKey

Computes a public key from the given private key of the elliptic curve cryptosystem over GF(p).

Syntax

```
IppStatus ippsGFpECPublicKey(const IppsBigNumState* pPrivate, IppsGFpECPoint* pPublic,
IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

pPrivate	Pointer to the private key *privKey*.
pPublic	Pointer to the public key *pubKey*.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

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 private key *privKey* can be generated by the function GFpECPrivateKey.

The context of the point *pubKey* as an elliptic curve point must be created by using the functions GFpECPointGetSize and GFpECPointInit.

The elliptic curve domain parameters must be defined by the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 contexts pointed to by *pPrivate*, *pPublic*, or *pEC* does not match the operation.
ippStsIvalidPrivateKey	Indicates an error condition if the value of the private key falls outside the range of [1, *n*-1].

GFpECTstKeyPair

Tests private and public keys of the elliptic curve cryptosystem over GF(p).

Syntax

```
IppStatus ippsGFpECTstKeyPair(const IppsBigNumState* pPrivate, const IppsGFpECPoint*
pPublic, IppECResult* pResult, IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

ippcp.h

Parameters

pPrivate	Pointer to the private key privKey.
pPublic	Pointer to the public key pubKey.
pResult	Pointer to the validation result.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

Description

The function tests the private key *privKey* and public key *pubKey* of the elliptic curve 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 the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if any of the contexts pointed by *pPrivate*, *pPublic*, or *pEC* does not match the operation.
ippStsRangeErr	Indicates an error condition if the public key point does not belong to the finite field over which the elliptic curve is initialized.

GFpECSharedSecretDH

Computes a shared secret field element by using the Diffie-Hellman scheme.

Syntax

```
IppStatus ippsGFpECSharedSecretDH(const IppsBigNumState* pPrivateA, const
IppsGFpECPoint* pPublicB, IppsBigNumState* pShare, IppsGFpECState* pEC, Ipp8u*
pScratchBuffer);
```

Include Files

ippcp.h

Parameters

pPrivateA	Pointer to your own private key *privKey*.
pPublicB	Pointer to the public key *pubKey*.
pShare	Pointer to the secret number *bnShare*.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

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 the x-coordinate of the secret point on the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 to by *pPublicB*, *pPrivateA*, *pShare*, or *pEC* does not match the operation.
ippStsRangeErr	Indicates an error condition if the memory size of *bnShare* pointed to by *pShare* is less than the size of the GFp modulus that is base for the specified elliptic curve.
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.)

GFpECSharedSecretDHC

Computes a shared secret field element by using the Diffie-Hellman scheme and the elliptic curve cofactor.

Syntax

```
IppStatus ippsGFpECSharedSecretDHC(const IppsBigNumState* pPrivateA, const
IppsGFpECPoint* pPublicB, IppsBigNumState* pShare, IppsGFpECState* pEC, Ipp8u*
pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

pPrivate	Pointer to your own private key *privKey*.
pPublic	Pointer to the public key *pubKey*.
pShare	Pointer to the secret number *bnShare*.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

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 *shareB*. When calculating, he uses his own private key and Alice's public key and applies the following formula: *shareB* = *h* · *privKeyB* · *pubKeyA* = *h* · *privKeyB* · *privKeyA* · *G*, where *h* is the elliptic curve cofactor.

Shared secret *bnShare* is the x-coordinate of the secret point on the elliptic curve.

The elliptic curve domain parameters must be hitherto defined by the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 contexts pointed to by *pPrivate*, *pPublic*, *pShare*, or *pEC* does not match the operation.
ippStsRangeErr	Indicates an error condition if the memory size of *bnShare* pointed to by *pShare* is less than the size of the GFp modulus that is the base for the specified elliptic curve.
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.)

GFpECSignDSA

Computes a digital signature over a message digest.

Syntax

```
IppStatus ippsGFpECSignDSA(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pRegPrivate, const IppsBigNumState* pEphPrivate, IppsBigNumState* pSignR,
IppsBigNumState* pSignS, IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

pMsgDigest	Pointer to the message digest *msg* to be digitally signed, that is, to be ecrypted with a private key.
pRegPrivate	Pointer to the signer's regular private key.
pEphPrivate	Pointer to the signer's ephemeral private key.
pSignR	Pointer to the integer *r* of the digital signature.
pSignS	Pointer to the integer *s* of the digital signature.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

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 the ECDSA scheme, an elliptic curve analogue of the DSA scheme.

The regular private key *regPrivKey* and the ephemeral private key *ephPrivKey* can be generated by the functions GFpECPrivateKey and GFpECPublicKey with only the requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 any of the contexts pointed to by *pMsgDigest*, *pRegPrivate*, *pEphPrivate*, *pSignR*, *pSignS*, or *pEC* does not match the operation.
ippStsMessageErr	Indicates an error condition if the value of *msg* pointed to 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 any of the parameters pointed to by *pSignR* or *pSignS* has a memory size that is less than the order *n* of the elliptic curve base point *G*.
ippStsIvalidPrivateKey	Indicates an error condition if any of the parameters pointed to by *pRegPrivate* or *pEphPrivate* has a memory size that is less than the order *n* of the elliptic curve base point *G*.
ippStsNotSupportedModeErr	Indicates an error condition if the finite field GFp under the elliptic curve is not prime.

GFpECVerifyDSA

Verifies authenticity of the digital signature over a message digest (ECDSA).

Syntax

```
IppStatus ippsGFpECVerifyDSA(const IppsBigNumState* pMsgDigest, const IppsGFpECPoint*
pRegPublic, const IppsBigNumState* pSignR, const IppsBigNumState* pSignS, IppECResult*
pResult, IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

ippcp.h

Parameters

pMsgDigest	Pointer to the message digest *msg*.
pRegPublic	Pointer to the signer's regular public key.
pSignR	Pointer to the integer *r* of the digital signature.
pSignS	Pointer to the integer *s* of the digital signature.
pResult	Pointer to the digital signature verification result.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

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. You can get the message sender's regular public key *regPubKey* by calling the function GFpECPublicKey.

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 `GFpECVerifyDSA` function must be preceded by a call to the GFpECSignDSA 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 the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 any of the contexts pointed to by *pMsgDigest*, *pRegPublic*, *pSignR*, *pSignS*, or *pEC* does not match the operation.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed to by *pMsgDigest* is negative.
`ippStsRangeErr`	Indicates an error condition if any of the parameters pointed to by *pSignR* or *pSignS* is negative.
`ippStsNotSupportedModeErr`	Indicates an error condition if the finite field GFp under the elliptic curve is not prime.
`ippStsOutOfRangeErr`	Indicates an error condition if the public key point does not belong to the finite field over which the elliptic curve is initialized.

GFpECSignNR

Computes the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippsGFpECSignNR(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pRegPrivate, const IppsBigNumState* pEphPrivate, IppsBigNumState* pSignR,
IppsBigNumState* pSignS, IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

pMsgDigest	Pointer to the message digest *msg* to be digitally signed, that is, to be ecrypted with a private key.
pRegPrivate	Pointer to the signer's regular private key.
pEphPrivate	Pointer to the signer's ephemeral private key.
pSignR	Pointer to the integer *r* of the digital signature.
pSignS	Pointer to the integer *s* of the digital signature.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

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 regular private key *regPrivKey* and the ephemeral private key *ephPrivKey* can be generated by the functions GFpECPrivateKey and GFpECPublicKey with only the requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 any of the contexts pointed by *pMsgDigest*, *pRegPrivate*, *pEphPrivate*, *pSignR*, *pSignS*, or *pEC* does not match the operation.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed to 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 any of the parameters pointed to by *pSignR* or *pSignS* has a memory size that is less than the order *n* of the elliptic curve base point *G*.
`ippStsIvalidPrivateKey`	Indicates an error condition if any of the parameters pointed to by *pRegPrivate* or *pEphPrivate* has a memory size that is less than the order *n* of the elliptic curve base point *G*.
`ippStsNotSupportedModeErr`	Indicates an error condition if the finite field GFp under the elliptic curve is not prime.

GFpECVerifyNR

Verifies authenticity of the digital signature over a message digest (the Nyberg-Rueppel scheme).

Syntax

```
IppStatus ippsGFpECVerifyNR(onst IppsBigNumState* pMsgDigest, const IppsGFpECPoint*
pRegPublic, const IppsBigNumState* pSignR, const IppsBigNumState* pSignS, IppECResult*
pResult, IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

pMsgDigest	Pointer to the message digest *msg*.
pRegPublic	Pointer to the signer's regular public key.
pSignR	Pointer to the integer *r* of the digital signature.
pSignS	Pointer to the integer *s* of the digital signature.
pResult	Pointer to the digital signature verification result.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

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 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).

You can get the message sender's regular public key *regPubKey* by calling the function GFpECPublicKey.

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 `GFpECVerifyNR` function must be preceded by a call to the GFpECSignNR 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 the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 any of the contexts pointed to by *pMsgDigest*, *pRegPublic*, *pSignR*, *pSignS*, or *pEC* does not match the operation.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed to 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 any of the parameters pointed to by *pSignR* or *pSignS* is negative.
`ippStsOutOfRangeErr`	Indicates an error condition if the public key point does not belong to the finite field over which the elliptic curve is initialized.
`ippStsNotSupportedModeErr`	Indicates an error condition if the finite field GFp under the elliptic curve is not prime.

GFpECSignSM2

Computes a digital signature over a message digest using the SM2 scheme.

Syntax

```
IppStatus ippsGFpECSignSM2(const IppsBigNumState* pMsgDigest, const IppsBigNumState*
pRegPrivate, const IppsBigNumState* pEphPrivate, IppsBigNumState* pSignR,
IppsBigNumState* pSignS, IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

`ippcp.h`

Parameters

pMsgDigest	Pointer to the message digest *msg* to be digitally signed, that is, to be ecrypted with a private key.
pRegPrivate	Pointer to the signer's regular private key.
pEphPrivate	Pointer to the signer's ephemeral private key.
pSignR	Pointer to the integer *r* of the digital signature.
pSignS	Pointer to the integer *s* of the digital signature.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

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 regular private key *regPrivKey* and the ephemeral private key *ephPrivKey* can be generated by the functions GFpECPrivateKey and GFpECPublicKey with only the requirement that the key *regPrivKey* be different from the key *ephPrivKey*.

The elliptic curve domain parameters must be hitherto defined by the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 contexts pointed to by *pMsgDigest*, *pRegPrivate*, *pEphPrivate*, *pSignR*, *pSignS*, or *pEC* does not match the operation.
`ippStsMessageErr`	Indicates an error condition if the value of *msg* pointed to by *pMsgDigest* is negative.
`ippStsRangeErr`	Indicates an error condition if any of the parameters pointed to by *pSignR* or *pSignS* has a memory size that is less than the order *n* of the elliptic curve base point *G*.
`ippStsIvalidPrivateKey`	Indicates an error condition in the following cases: • Any of the parameters pointed to by *pRegPrivate* or *pEphPrivate* has a memory size that is less than the order *n* of the elliptic curve base point *G*. • The value of any of the private keys is greater than or equal to the order *n* of the elliptic curve base point *G*.
`ippStsNotSupportedModeErr`	Indicates an error condition if the finite field GFp under the elliptic curve is not prime.

GFpECVerifySM2

Verifies authenticity of a digital signature over a message digest using the SM2 scheme.

Syntax

```
IppStatus ippsGFpECVerifySM2(const IppsBigNumState* pMsgDigest, const IppsGFpECPoint*
pRegPublic, const IppsBigNumState* pSignR, const IppsBigNumState* pSignS, IppECResult*
pResult, IppsGFpECState* pEC, Ipp8u* pScratchBuffer);
```

Include Files

ippcp.h

Parameters

pMsgDigest	Pointer to the message digest *msg*.
pRegPublic	Pointer to the signer's regular public key.
pSignR	Pointer to the integer *r* of the digital signature.
pSignS	Pointer to the integer *s* of the digital signature.
pResult	Pointer to the digital signature verification result.
pEC	Pointer to the context of the elliptic curve.
pScratchBuffer	Pointer to the scratch buffer.

Description

The function verifies authenticity of the digital signature, represented as integer big numbers *r* and *s*, over a message digest *msg*. The digital signature over the message digest *msg* must be computed using the SM2 scheme [SM2] by to the GFpECSignSM2 function.

You can get the message sender's regular public key *regPubKey* by calling the function GFpECPublicKey.

The result of the digital signature verification can take one of these values:

ippECValid	Digital signature is valid.
ippECInvalidSignature	Digital signature is not valid.

The elliptic curve domain parameters must be hitherto defined by the functions: GFpECInitStd, GFpECInit, GFpECSet, or GFpECSetSubgroup.

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 contexts pointed to by *pMsgDigest*, *pRegPublic*, *pSignR*, *pSignS*, or *pEC* does not match the operation.
ippStsMessageErr	Indicates an error condition if the value of *msg* pointed to by *pMsgDigest* is negative.
ippStsRangeErr	Indicates an error condition if any of the parameters pointed to by *pSignR* or *pSignS* is negative.
ippStsOutOfRangeErr	Indicates an error condition if the public key point does not belong to the finite field over which the elliptic curve is initialized.
ippStsNotSupportedModeErr	Indicates an error condition if the finite field GFp under the elliptic curve is not prime.

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`

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`	"Validation passed successfully"
`ippECCompositeBase`	"Finite Field produced by Composite"
`ippECComplicatedBase`	"Too many non-zero terms in the polynomial"
`ippECIsZeroDiscriminant`	"Zero discriminant"
`ippECCompositeOrder`	"Composite Base Point order"
`ippECInvalidOrder`	"Composite Base Point order"
`ippECIsWeakMOV`	"EC cover by MOV Reduction Test"
`ippECIsWeakSSSA`	"EC cover by SS-SA Reduction Test"
`ippECIsSupersingular`	"EC is supersingular curve"
`ippECInvalidPrivateKey`	"Invalid Private Key"
`ippECInvalidPublicKey`	"Invalid Public Key"
`ippECInvalidKeyPair`	"Invalid Key Pair"
`ippECPointOutOfGroup`	"Point is out of group"
`ippECPointAtInfinite`	"Point at infinity"
`ippECPointIsNotValid`	"Invalid EC Point"
`ippECPointIsEqual`	"Points are equal"
`ippECPointIsNotEqual`	"Points are different"
`ippECInvalidSignature`	"Invalid Signature"

See Also

ECCPValidate
ECCPValidateKeyPair

Finite Field Arithmetic

6

This section describes the Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography) functions that implement arithmetic operations with elements of the following finite fields [ANT]:

GF(*p*)	A finite field of *p* elements.
GF(*q*)	If *q* is an odd prime number, then the finite field is represented by integers modulo *q*. This field is also known as the *prime finite field*.
GF(p^d)	If *p* = *q*, *q* is an odd prime number and *d* > 1, the finite field is represented by polynomials modulo *g*(*x*), GF(*p*)[*x*]/*g*(*x*), where *g*(*x*) is an irreducible polynomial over GF(*p*). This field is also known as *a degree d extension of the GF(p) field*.
GF($((q^{n1})^{n2})^{n3}$)	A very complex extension of the prime finite field GF(*q*). The initial prime field GF(*q*) used at the lowest level of the construct is frequently called the *basic finite field* with respect to the extension.

The finite field arithmetic functions use context structures of the `IppsGFpState` and `IppsGFpElement` types to store data of the finite field and the field elements, respectively.

The `IppsGFpElement` type structure is used for *internal* representation of field elements. In application (*or external*) representation of field element is straightforward. Each element *E* of the prime field GF(*q*) is an unsigned number in the range [0, *q* - 1], which is represented by a data array `Ipp32u qe[len32]` , so that

$$E = \sum_{i=0}^{len32-1} qe[i]2^{32i}$$

where $len32 = \lceil bitsize(q)/32 \rceil$ is the length of the prime *q*, expressed in *dwords* (32-bit chunks).

Each element *E* of GF(p^d) is represented by a polynomial of degree less than *d*. This polynomial is represented by an array of coefficients `pe[d]` that belong to GF(*p*).

$$E = \sum_{j=0}^{d-1} x^j \left(\sum_{i=0}^{len32-1} qe[i]2^{32i} \right)$$

Thus,

```
Ipp32u a[4] = {0xBFF9AEE1,0xBF59CC9B,0xD1B3BBFE,0xD6031998};
```

is an external (application-side) representation of an element that belongs to some prime field GF(*q*), bitsize(*q*)=128.

Similarly,

```
Ipp32u b[2][4] = { {0xBFF9AEE1,0xBF59CC9B,0xD1B3BBFE,0xD6031998},
                   {0xBB6D8A5D,0xDC2C6558,0x80D02919,0x5EEEFCA3} };
```

is an external (application-side) representation of an element that belongs to GF(q^2) - a degree 2 extension of some prime field GF(*q*), bitsize(*q*)=128.

You can use Intel IPP Cryptography finite field functions to convert between the internal and the external representations of a finite field element.

Prime finite fields are the basic mathematical objects of Elliptic Curve (EC) cryptography. Intel IPP Cryptography supports different kinds of EC over finite fields and, in particular, the *standard* elliptic curves - elliptic curves with pre-defined parameters, including the underlying finite field. The performance of EC functionality directly depends on the efficiently of the implementation of operations with finite field elements such as addition, multiplication, and squaring.

Intel IPP Cryptography contains several different optimized implementations of finite field arithmetic functions. These implementations, referred to in this document as "methods", are grouped together in structures. Intel IPP Cryptography does not reveal the content of these structures. The implementations, including those optimized for a particular prime *q*, are accessed by special Intel IPP Cryptography functions. For example, `ippsGFpMethod_p192r1()` returns a pointer to the structure containing optimized arithmetic over prime *p192r1* (see `GFpMethod` for details).

Similarly, for GF(p^d), additional knowledge concerning the predefined field polynomial $g(x)$ allows Intel IPP Cryptography to provide a more efficient implementation of finite field arithmetic than in the case of an arbitrary field polynomial $g(x)$. Intel IPP Cryptography contains *methods* dedicated to certain predefined $g(x)$. For example, the functions `ippsGFpxMethod_binom2()` returns a pointer to the structure containing optimized arithmetic over GF(p^2).

The comparison function `GFpCmpElement` returns the result of comparison:

```
#define IPP_IS_EQ (0) // elements are equal
#define IPP_IS_GT (1) // the first element is greater than the second one
#define IPP_IS_LT (2) // the first element is less than the second one
#define IPP_IS_NE (3) // elements are not equal
#define IPP_IS_NA (4) // elements are not comparable
```


GFpInitFixed

Initializes the context of a prime finite field GF(q) with a predefined modulus q.

Syntax

`IppStatus ippsGFpInitFixed(int primeBitSize, const IppsGFpMethod* method, IppsGFpState* pGF);`

Include Files

`ippcp.h`

Parameters

primeBitSize	Size, in bytes, of the odd prime number *q* (modulus of GF(*q*)).
method	Pointer to the implementation of a basic arithmetic (methods) over the prime finite field GF(*q*) with a predefined *q*.
pGF	Pointer to the context of the GF(*q*) field being initialized.

Description

The function initializes the memory buffer *pGF* associated with the `IppsGFPState` context and sets up the specific value of the GF(q) modulus corresponding to the chosen *method*. The initialized context is used in the functions that create contexts of elements of the GF(p) field, which, in turn, are used to perform operations with the field elements.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is `NULL`.
`ippStsBadArgErr`	Indicates an error condition in the following cases: • *method* is not a pointer to an implementation of a prime finite field arithmetic with a predefined modulus • *method* does not correspond to the size of modulus q defined in a `ippsGFpGetSize()` call.

GFpInitArbitrary

Initializes the context of an arbitrary prime finite field GF(q).

Syntax

`IppStatus ippsGFpInitArbitrary(const IppsBigNumState* pPrime, int primeBitSize, IppsGFpState* pGF);`

Include Files

`ippcp.h`

Parameters

pPrime	Pointer to the Big Number context storing the GF(q) modulus.
primeBitSize	Size, in bytes, of the odd prime number q (modulus of GF(q)).
pGF	Pointer to the context of the GF(q) field being initialized.

Description

The function initializes the memory buffer *pGF* associated with the `IppsGFPState` context and sets the GF(q) modulus to the value specified by *pPrime*. This function uses `ippsGFpMethod_pArb()` to get an implementation of the finite field arithmetic. The initialized context is used in the functions that create contexts of elements of the GF(p) field, which, in turn, are used to perform operations with the field elements.

NOTE
This function does not check if *pPrime* actually refers to a prime 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`.
`ippStsSizeErr`	Indicates an error condition if *`primeBitSize`* is less than 2 or greater than 1024.
`ippStsContextMatchErr`	Indicates an error condition if the *`pPrime`* context does not match the operation.
`ippStsBadArgErr`	Indicates an error condition in the following cases: • The GF(q) modulus q is less than 3. • bitsize(q) != *`primeBitSize`*. • q is even.

GFpInit

Initializes the context of a prime finite field GF(q).

Syntax

```
IppStatus ippsGFpInit(const IppsBigNumState* pPrime, int primeBitSize, const
IppsGFpMethod* method, IppsGFpState* pGF);
```

Include Files

`ippcp.h`

Parameters

`pPrime`	Pointer to the Big Number context storing the GF(q) modulus.
`primeBitSize`	Size, in bytes, of the odd prime number p (modulus of GF(q)).
`method`	Pointer to the implementation of a basic arithmetic (methods) over the prime finite field GF(q).

NOTE
If your application uses one of predefined values of the modulus q, the use of the `GFpMethod` function corresponding to that value is preferable. In other cases, use `ippsGfpMethod_pArb()`.

`pGF`	Pointer to the context of the GF(q) field being initialized.

Description

NOTE
This function combines the roles of `ippsGFpInitFixed()` and `ippsGFpInitArbitrary()` and is kept for backward compatibility. Using `ippsGFpInitFixed()` and `ippsGFpInitArbitrary()` explicitly is considered preferable.

The function initializes the *`pGF`* context parameter with the values of the input parameters*`pPrime`*, *`primeBitSize`*, and *`method`*. The three parameters have to be compatible with each other.

If *pPrime* == `NULL`, then the behavior of `ippsGFpInit()` is similar to that of `ippsGFpInitFixed()`. *method* must be an output from one of the `GFpMethod` functions with predefined modulus *q*, and the parameters *primeBitSize* and *method* must be compatible with each other.

If *pPrime* is not `NULL`, and *method* is an output from one of the `GFpMethod` functions with predefined modulus *q*, then the pair *pPrime* and *primeBitSize* should define the same prime *q* as defined in *method*.

If *method* == `NULL`, then the behavior of `ippsGFpInit()` is similar to that of `ippsGFpInitArbitrary()`.

If both *pPrime* and *method* are not `NULL`, then `ippsGFpInit()` provides the required initialization if the parameters are compatible with each other.

The initialized context is used in the functions that create contexts of elements of the GF(*p*) field, which, in turn, are used to perform operations with the field elements.

NOTE
This function does not check if *pPrime* actually refers to a prime value.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition in the following cases: • *pGF* is `NULL`. • Both *pPrime* and *method* are `NULL`.
`ippStsSizeErr`	Indicates an error condition if *primeBitSize* is less than 2 or greater than 1024.
`ippStsContextMatchErr`	Indicates an error condition if the *pPrime* context parameter is not `NULL` and does not match the operation.

`ippStsBadArgErr`	Indicates an error condition in the following cases:
	• The modulus *q* defined in *pPrime* is less than 3.
	• bitsize(*q*) != *primeBitSize*.
	• *q* is even.
	• *method* is not `NULL` and not an output of `GFpMethod`.
	• *method* is an output from one of the `GFpMethod` functions with predefined modulus *q*, but:
	  • The bit size of *q* of *method* is different from the bit size of the value stored in the context pointed to by *pPrime*.
	  • *q* of *method* is different from the value stored in the context pointed to by *pPrime*.

GFpMethod

Returns a reference to an implementation of arithmetic operations over GF(q).

Syntax

```
const IppsGFpMethod* ippsGFpMethod_p192r1(void);
```

```
const IppsGFpMethod* ippsGFpMethod_p224r1(void);
```

```
const IppsGFpMethod* ippsGFpMethod_p256r1(void);
const IppsGFpMethod* ippsGFpMethod_p384r1(void);
const IppsGFpMethod* ippsGFpMethod_p521r1(void);
const IppsGFpMethod* ippsGFpMethod_p256sm2(void);
const IppsGFpMethod* ippsGFpMethod_p256bn(void);
const IppsGFpMethod* ippsGFpMethod_pArb(void);
```

Include Files

`ippcp.h`

Description

Each of these functions returns a pointer to a structure containing an implementation of arithmetic operations over GF(q).

`ippsGFpMethod_pArb()` assumes an arbitrary modulus q; each of the rest of the functions returns a pointer to the implementation of arithmetic operations over GF(q) tailored for a particular q. See the table below for the correspondence between method functions and values of the modulus q.

Function	**Value of modulus *q***
`ippsGFpMethod_p192r1()`	$q = 2^{192} - 2^{64} - 1$
`ippsGFpMethod_p224r1()`	$q = 2^{224} - 2^{96} - 1$
`ippsGFpMethod_p256r1()`	$q = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
`ippsGFpMethod_p384r1()`	$q = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
`ippsGFpMethod_p521r1()`	$q = 2^{521} - 1$
`ippsGFpMethod_p256sm2()`	$q = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$
`ippsGFpMethod_p256bn()`	$q =$ `0xFFFFFFFFFFFCF0CD46E5F25EEE71A49F0CDC65FB12980A82D3292DDBAED33013`
`ippsGFpMethod_pArb()`	Arbitrary modulus q

GFpGetSize

Gets the size of the context of a GF(q) field.

Syntax

`IppStatus ippsGFpGetSize(int bitSize, int* pStateSizeInBytes);`

Include Files

`ippcp.h`

Parameters

bitSize	Size, in bytes, of the odd prime number q (modulus of GF(q)).
pStateSizeInBytes	Pointer to the buffer size, in bytes, needed for the `IppsGFpState` context.

Description

This function returns the size of the buffer associated with the `IppsGFpState` context, which you can use to store data of the finite field GF(q) determined by the odd prime number q of size not greater than *bitSize* bit.

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 *`bitSize`* is less than 2 or greater than 1024.

GFpxInitBinomial

Initializes the context of a GF(p^d) field.

Syntax

```
IppStatus ippsGFpxInitBinomial(const IppsGFpState* pParentGF, int extDeg, const
IppsGFpElement* const pParentElm, const IppsGFpMethod* method, IppsGFpState* pGFpx);
```

Include Files

`ippcp.h`

Parameters

`pParentGF`	Pointer to the context of the finite field GF(p) being extended.
`extDeg`	Degree of the extension.
`pParentElm`	Pointer to the `IppsGFpElement` context containing the trailing coefficient of the field binomial.
`method`	Pointer to the implementation of a basic arithmetic (methods) over GF(p^d).
`pGFpx`	Pointer to the context of the GF(p^d) field being initialized.

Description

This function initializes the memory buffer *`pGFpx`* associated with the `IppsGFpState` context and sets up the specific irreducible binomial. The initialized context is used in the functions that create contexts of elements of the GF(p^d) field and perform operations with field elements.

NOTE
The function does not check the binomial's irreducibility.

Important
When calling the functions over the GF(p^d) field, a properly initialized *`pParentGF`* context of the finite field GF(p) is required.

Return Values

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

ippStsContextMatchErr	Indicates an error condition if any of the context parameters *pParentGF* and *pParentElm* does not match the operation.
ippStsBadArgErr	Indicates an error condition in the following cases: • *extDeg* > 8 or *extDeg* < 2. • *method* is not in agreement with other parameters.
ippStsOutOfRangeErr	Indicates an error condition if the length of the value defined in *pParentElm* is not equal to that of an element of *pParentGF*.

GFpxInit

Initializes the context of a GF(p^d) field.

Syntax

```
IppStatus ippsGFpxInit(const IppsGFpState* pParentGF, int extDeg, const IppsGfpElement*
const ppParentElm[], int polyTerms, const IppsGFpMethod* method, IppsGFpState* pGFpx);
```

Include Files

ippcp.h

Parameters

pParentGF	Pointer to the context of the finite field GF(p) being extended.
extDeg	Degree of the extension.
ppParentElm[]	Pointer to the array of IppsGFpElement contexts representing coefficients of the field polynomial.
polyTerms	Number of the field polynomial coefficients.
method	Pointer to the implementation of a basic arithmetic (methods) over the extended GF(p) finite field.
pGFpx	Pointer to the context of the GF(p^d) field being initialized.

Description

The function initializes the memory buffer *pGFpx* associated with the IppsGFpState context and sets up the specific irreducible polynomial. The initialized context is used in the functions that create contexts of elements of the GF(p^d) field and perform operations with the field elements. The function assumes the use of a general field polynomial $g(x) = x^d + x^{d-1}a_{d-1} + x^{d-2}a_{d-2} + \cdots + x^1 a_1 + a_0$ over GF(p).

- The function does not check the polynomial's irreducibility.
- In general, the GF(p^d) extension requires a field polynomial $g(x)$ of degree d. However, because $g(x)$ is considered a monic polynomial (the coefficient of *xd* is always assumed equal to 1), the leading coefficient is not required: *polyTerms* <= (*extDeg* - 1).

- When calling the functions over the GF(p^d) field, a properly initialized *pParentGF* context of the finite field GF(p) is required.
- Do not release the *pParentGF* context of the parent field as long as application deals with either the parent or the extended finite field pointed to by *pGFpx*.

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 referenced by elements of *`ppParentElm[]`* or *`pParentGF`* does not match the operation.
`ippStsBadArgErr`	Indicates an error condition in the following cases: • *`extDeg`* > 8 or *`extDeg`* < 2. • *`polyTerms`* > (*`extDeg`* - 1) or *`polyTerms`* < 1. • *`method`* is not an output of a `GFpxMethod` function. • *`method`* is not compatible with the value of *`extDeg`*.
`ippStsOutOfRangeErr`	Indicates an error condition if the length of any of the values defined by *`ppPrentElm[]`* is not equal to the length of an element of the parent finite field *`pParentGF`*.

GFpxMethod

Returns a reference to the implementation of arithmetic operations over GF(p^d).

Syntax

```
const IppsGFpMethod* ippsGFpxMethod_com(void);
const IppsGFpMethod* ippsGFpxMethod_binom2(void);
const IppsGFpMethod* ippsGFpxMethod_binom3(void);
const IppsGFpMethod* ippsGFpxMethod_binom(void);
const IppsGFpMethod* ippsGFpxMethod_binom2_epid2(void);
const IppsGFpMethod* ippsGFpxMethod_binom3_epid2(void);
```

Include Files

`ippcp.h`

Description

Each of these functions returns a pointer to a structure containing an implementation of arithmetic operations over GF(p^d).

`ippsGFpxMethod_com` assumes an arbitrary value of the field polynomial $g(x)$; each of the rest of the functions returns a pointer to the implementation of arithmetic operations over GF(p^d) tailored for a particular value of $g(x)$. See the table below for the correspondence between method functions and values of the field polynomial $g(x)$.

Function	Value of the field polynomial $g(x)$
`ippsGFpxMethod_com`	$g(x) = x^d + x^{d-1}a_{d-1} + x^{d-2}a_{d-2} + \cdots + x^1a_1 + a_0$, $a_i \in$ GF(p)
`ippsGFpxMethod_binom2`	$g(x) = x^2 - a_0$, $a_0 \in$ GF(p)
`ippsGFpxMethod_binom3`	$g(x) = x^3 - a_0$, $a_0 \in$ GF(p)
`ippsGFpxMethod_binom`	$g(x) = x^d - a_0$, $a_0 \in$ GF(p)

Function	Value of the field polynomial $g(x)$
ippsGFpxMethod_binom2_epid2	$g(x) = x^2 - a_0$, $a_0 \in \mathrm{GF}(q)$, $a_0 = 1$
	$g(w) = w^2 - V_0$, $v_0 \in \mathrm{GF}((q^2)^3)$, $V_0 = 0 \cdot v^2 + v + 0$
ippsGFpxMethod_binom3_epid2	$g(v) = v^3 - U_0$, $U_0 \in \mathrm{GF}(q^2)$, $U_0 = u + 2$

NOTE

ippsGFpxMethod_binom2_epid2() and ippsGFpxMethod_binom3_epid2() are designed especially for the construction of finite field extensions for applications that use the Intel® Enhanced Privacy ID 2.0 scheme.

GFpxGetSize

Gets the size of the context of a GF(p^d) field.

Syntax

```
IppStatus ippsGFpxGetSize(const IppsGFpState* pParentGF, int degree, int*
pStateSizeInBytes);
```

Include Files

ippcp.h

Parameters

pParentGF	Pointer to the context of the finite field GF(p) being extended.
degree	Degree of the extension.
pStateSizeInBytes	Pointer to the buffer size, in bytes, needed for the IppsGFpState context.

Description

The function returns the size of the buffer associated with the IppsGFpState context, suitable for storing data for the finite field GF(p^d) determined by the extension degree d supplied in the *degree* 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 IppsGFpState context parameter does not match the operation.
ippStsBadArgErr	Indicates an error condition if the degree of the extension is greater than or equal to 9 or is less than 2.

GFpScratchBufferSize

Gets the size of the scratch buffer.

Syntax

```
IppStatus ippsGFpScratchBufferSize(int nExponents, int ExpBitSize, const IppsGFpState*
pGF, int* pBufferSize);
```

Include Files

`ippcp.h`

Parameters

nExponents	Number of exponents.
ExpBitSize	Maximum bit size of the exponents.
pGFp	Pointer to the context of the finite field.
pBufferSize	Pointer to the calculated buffer size in bytes.

Description

This function computes the size of the scratch buffer for the `ippsGFpExp` and `ippsGFpMultiExp` functions. The *pGFp* parameter specifies the context of the finite field.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if the *pGFp* context parameter does not match the operation.
`ippStsBadArgErr`	Indicates an error condition in the following cases: • The number of exponents is zero or negative. • The number of exponents is greater than 6.

GFpElementGetSize

Gets the size of the context for an element of the finite field.

Syntax

```
IppStatus ippsGFpElementGetSize(const IppsGFpState* pGFp, int* pElementSize);
```

Include Files

`ippcp.h`

Parameters

pGFp	Pointer to the context of the finite field.
pElementSize	Pointer to the buffer size, in bytes, needed for the `IppsGFpElement` context.

Description

This function returns the size of the buffer associated with the `IppsGFpElement` context, suitable for storing an element of the finite field specified by the context *pGFp*.

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 *pGFp* context parameter does not match the operation.

GFpElementInit

Initializes the context of an element of the finite field.

Syntax

```
IppStatus ippsGFpElementInit(const Ipp32u* pA, int nsA, IppsGFpElement* pR,
IppsGFpState* pGF);
```

Include Files

ippcp.h

Parameters

pA	Pointer to the data array storing the finite field element.
lenA	Length of the element.
pR	Pointer to the context of the finite field element being initialized.
pGFp	Pointer to the context of the finite field.

Description

This function initializes the memory buffer *pR* associated with the IppsGFpElement context and sets up the specific element of the finite field specified by the *pGFp* context. The initialized IppsGFpElement context is used in all the operations with this element of the finite field.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the specified pointers is NULL.
ippStsContextMatchErr	Indicates an error condition if the *pGFp* context parameter does not match the operation.
ippStsSizeErr	Indicates an error condition if *lenA*≤ 0.

GFpSetElement

Assigns a value to an element of the finite field.

Syntax

```
IppStatus ippsGFpSetElement(const Ipp32u* pA, int nsA, IppsGFpElement* pR,
IppsGFpState* pGFp);
```

Include Files

ippcp.h

Parameters

pA	Pointer to the data array storing the finite field element.
nsA	Length of the element.
pR	Pointer to the context of the finite field element being assigned.
pGFp	Pointer to the context of the finite field.

Description

This function copies (and converts if needed) the value from the user-defined *pA* buffer to the `IppsGFpElement` context of the finite field element. If *pR* is `NULL`, `GFpSetElement` assigns zero to the element.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition in the following cases: • Either *pR* or *pGFp* is `NULL`. • The length of the element *nsA* is greater than zero and the pointer *pA* is `NULL`.
ippStsContextMatchErr	Indicates an error condition if any of the *pGFp* and*pR* context parameters does not match the operation.
ippStsSizeErr	Indicates an error condition in the following cases: • *nsA* is not equal to the length of an element of the finite field. • The maximum length of the element stored in the context *pR* exceeds the maximum length of an element of the finite field specified by the context *pGFp*.
ippStsOutOfRangeErr	Indicates an error condition if the value contained in *pA* exceeds the modulus *q* of the basic prime finite field.

GFpSetElementOctString

Assigns a value from the input octet string to an element of the finite field.

Syntax

```
IppStatus ippsGFpSetElementOctString(const Ipp8u* pStr, int strSize, IppsGFpElement*
pR, IppsGFpState* pGFp);
```

Include Files

ippcp.h

Parameters

pStr	Pointer to the octet string.

strSize	Size of the octet string buffer in bytes.
pR	Pointer to the context of the finite field element.
pGFp	Pointer to the context of the finite field.

Description

This function assigns a value from the input octet string to an element of the finite field.

Return Values

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

`ippStsNullPtrErr` — Indicates an error condition in any of the following cases:

- Either *pR* or *pGFp* is `NULL`.
- The length of the string is greater than zero and the pointer *pStr* is `NULL`.

`ippStsContextMatchErr` — Indicates an error condition if any of the *pGFp* and *pR* context parameters does not match the operation.

`ippStsSizeErr` — Indicates an error condition in any of the following cases:

- *strSize* exceeds the length of an element of the finite field.
- *strSize*≤ 0.
- The maximum length of the element stored in the context *pR* exceeds the maximum length of an element of the finite field specified by the context *pGFp*.

`ippStsOutOfRangeErr` — Indicates an error condition in any of the following cases:

- The length of the element stored in the context *pR* is not equal to the length of an element of the finite field specified by the context *pGFp*.
- The value defined by *pStr* exceeds the modulus *q* of the basic prime finite field.

GFpSetElementRandom

Assigns a random value to an element of the finite field.

Syntax

```
IppStatus1 ippsGFpSetElementRandom(IppsGFpElement* pR, IppsGFpState* pGFp,
IppBitSupplier rndFunc, void* pRndParam);
```

Include Files

`ippcp.h`

Parameters

pR	Pointer to the context of the finite field element.
pGFp	Pointer to the context of the finite field.

rndFunc	Pseudorandom number generator.
pRndParam	Pointer to the context of the pseudorandom number generator.

Description

This function assigns a random value to an element of the finite field.

Return Values

ippStsNoErr	Indicates no error. Any other value indicates an error or warning.
ippStsNullPtrErr	Indicates an error condition if any of the pointers *pR*, *pGFp* and *rndFunc* is NULL.
ippStsContextMatchErr	Indicates an error condition if any of *pGFp* or *pR* context parameters does not match the operation.
ippStsErr	Indicates an error condition in the following cases: • A call to the *rndFunc*() function returns a status value other than ippStsNoErr. • The maximum length of the element stored in the context *pR* exceeds the maximum length of an element of the finite field specified by the context *pGFp*.
ippStsOutOfRangeErr	Indicates an error condition if the length of the element stored in the context *pR* is not equal to the length of an element of the finite field specified by the context *pGFp*.

GFpSetElementHash

Assigns a value from the input hash to an element of the finite field.

Syntax

```
IppStatus ippsGFpSetElementHash(const Ipp8u* pMsg, int msgLen, IppsGFpElement* pElm,
IppsGFpState* pGF, IppHashAlgId hashID);
```

Include Files

ippcp.h

Parameters

pMsg	Pointer to the input message.
msgLen	Length of the input message.
pElm	Pointer to the context of the finite field element.
pGF	Pointer to the context of the finite field.
hashID	ID of the hash algorithm used. For details, see table Supported Hash Algorithms.

Description

This function computes an element of the finite field from the hash of the input message.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNotSupportedModeErr`	Indicates an error condition if *hashID* does not correspond to any supported hash ID.
`ippStsNullPtrErr`	Indicates an error condition in one of the following cases: • Any of the pointers *pElm* and *pGF* is `NULL`. • The *msgLen* is greater than zero and the pointer *pMsg* is `NULL`.
`ippStsLengthErr`	Indicates an error condition if *msgLen* is negative.
`ippStsContextMatchErr`	Indicates an error condition if any of the*pGF* and *pElm* context parameters does not match the operation.
`ippStsBadArgErr`	Indicates an error condition if the finite field specified by the context *pGF* is not a prime finite field.
`ippStsOutOfRangeErr`	Indicates an error condition if the length of the element stored in the context *pElm* is not equal to the length of an element of the finite field specified by the context *pGF*.

GFpCpyElement

Copies one element of the finite field to another element.

Syntax

```
IppStatus ippsGFpCpyElement(const IppsGFpElement* pA, IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the finite field element being copied.
pR	Pointer to the context of the finite field element being changed.
pGFp	Pointer to the context of the finite field.

Description

This function copies one element of the finite field to another. The finite field is specified by the context *pGFp*.

Return Values

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

`ippStsOutOfRangeErr`	Indicates an error condition if the input elements do not belong to the finite field specified by the context *pGFp*.

GFpGetElement

Extracts an element of the finite field from the context.

Syntax

`IppStatus ippsGFpGetElement(const IppsGFpElement* pA, Ipp32u* pDataA, int nsA, IppsGFpState* pGFp);`

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the finite field element.
pDataA	Pointer to the data array to copy the finite field element from.
nsA	Length of the data array.
pGFp	Pointer to the context of the finite field.

Description

This function copies the element of the finite field from the `IppsGFpElement` context to the user-defined *pDataA* buffer. The finite field is specified by the context *pGFp*.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition any of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if any of `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	The input elements do not belong to the finite field specified by the context *pGFp*
`ippStsSizeErr`	The length of the data array is negative or less than the finite field element length.

GFpGetElementOctString

Extracts an element of the finite field from the context to the output octet string.

Syntax

`IppStatus ippsGFpGetElementOctString(const IppsGFpElement* pA, Ipp8u* pStr, int strSize, IppsGFpState* pGFp);`

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the finite field element.
pStr	Pointer to the octet string.
strSize	Size of the octet string buffer in bytes.
pGFp	Pointer to the context of the finite field.

Description

This function extracts the element of the finite field from the context to the octet string. If the string length is not enough to hold the whole finite field element, the function writes only a part of the element.

Return Values

`ippStsNoErr`	Indicates no error. Any other value indicates an error or warning.
`ippStsNullPtrErr`	Indicates an error condition if any of the specified pointers is `NULL`.
`ippStsContextMatchErr`	Indicates an error condition if any of the *pGFp* and *pA* context parameters does not match the operation.
`ippStsSizeErr`	Indicates an error if the length of the string is zero or negative.
`ippStsOutOfRangeErr`	Indicates an error condition if the element *pA* does not belong to the finite field specified by the context *pGFp*.

GFpCmpElement

Compares two elements of the finite field.

Syntax

```
IppStatus ippsGFpCmpElement(const IppsGFpElement* pA, const IppsGFpElement* pB, int*
pResult, const IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the first finite field element.
pB	Pointer to the context of the second finite field element.
pResult	Pointer to the result of the comparison. For details, see comparison results.
pGFp	Pointer to the context of the finite field.

Description

This function compares two elements of the finite field and returns the result in *pResult*. The finite field is specified by the context *pGFp*.

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 `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if either *pA* or *pB* does not belong to the finite field specified by the context *pGFp*.

GFpIsZeroElement

Compares an element of the finite field with the zero element.

Syntax

```
IppStatus ippsGFpIsZeroElement(const IppsGFpElement* pA, int* pResult, const
IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the first finite field element.
pResult	Pointer to the result of the comparison. For details, see comparison results.
pGFp	Pointer to the context of the finite field.

Description

This function compares an element of the finite field with the zero element. The finite field is specified by the context *pGFp*.

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 `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if *pA* does not belong to the finite field specified by the context *pGFp*.

GFpIsUnityElement

Compares an element of the finite field with the unity element.

Syntax

```
IppStatus ippsGFpIsUnityElement(const IppsGFpElement* pA, int* pResult, const IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the first finite field element.
pResult	Pointer to the result of the comparison.For details, see comparison results.
pGFp	Pointer to the context of the finite field.

Description

This function compares an element of the finite field with the unity element. The finite field is specified by the context *pGFp*.

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 `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if *pA* does not belong to the finite field specified by the context *pGFp*.

GFpConj

Computes the conjugate of the element of the finite field GF(p^2).

Syntax

```
IppStatus ippsGFpConj(const IppsGFpElement* pA, IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the finite field element.

pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the conjugate of an element of the finite field GF(p^2). If the element of the GF(p^2) field is the polynomial $x + a$, the conjugate element is equal to $x - a$, where a is an element of the ground field 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`.
`ippStsContextMatchErr`	Indicates an error condition if any of `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if the element *pA* does not belong to the finite field specified by the context *pGFp*.
`ippStsBadArgErr`	Indicates an error condition if the element *pA* does not belong to the GF(p^2) field.

GFpNeg

Computes the additive inverse of an element of the finite field.

Syntax

```
IppStatus ippsGFpNeg(const IppsGFpElement* pA, IppsGFpElement* pR, IppsGFpState* pGF);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the finite field element.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the additive inverse of an element of the finite field. The following pseudocode represents this operation: $R + A = 0$. The finite field is specified by the context *pGFp*.

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 `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if *pA* does not belong to the finite field specified by the context *pGFp*.

GFpInv

Computes the multiplicative inverse of an element of the finite field.

Syntax

```
IppStatus ippsGFpInv(const IppsGFpElement* pA, IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the finite field element.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the multiplicative inverse of an element of the finite field. The following pseudocode represents this operation: $R \cdot A = 1$. The finite field is specified by the context *pGFp*.

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 `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if the element *pA* does not belong to the finite field specified by the context *pGFp*.
`ippStsDivByZeroErr`	Indicates an error condition if *pA* is the zero element.
`ippStsBadArgErr`	Indicates an error condition if a computational error occurs.

GFpSqrt

Computes the square root of an element of the finite field.

Syntax

```
IppStatus ippsGFpSqrt(const IppsGFpElement* pA, IppsGFpElement* pR, IppsGFpState*
pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the finite field element.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the square root of a given element of the GF(p) field. The following pseudocode represents this operation: $R \cdot R = A$. The finite field is specified by the *pGFp* 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 any of the IppsGFpState and IppsGFpElement context parameters does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if *pA* does not belong to the finite field specified by the context *pGFp*.
ippStsBadArgErr	Indicates an error condition the finite field specified by the context *pGFp* is not prime.
ippStsQuadraticNonResidueErr	Indicates an error condition if *pA* is a square non-residue element.

GFpAdd

Computes the sum of two elements of the finite field.

Syntax

```
IppStatus ippsGFpAdd(const IppsGFpElement* pA, const IppsGFpElement* pB,
IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the first element of the finite field to be added.
pB	Pointer to the context of the second element of the finite field to be added.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the sum of the elements of the finite field. The following pseudocode represents this operation: *R* = *A* + *B*. The finite field is specified by the *pGFp* 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 any of `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if either the *pA* or *pB* element does not belong to the finite field specified by the context *pGFp*.

GFpSub

Subtracts two elements of the finite field.

Syntax

```
IppStatus ippsGFpSub(const IppsGFpElement* pA, const IppsGFpElement* pB,
IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the minuend element of the finite field.
pB	Pointer to the context of the subtrahend element of the finite field.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the difference of the elements of the finite field. The following pseudocode represents this operation: *R* = *A* - *B*. The finite field is specified by the context *pGFp*.

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 `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition if *pA* or *pB* does not belong to the finite field specified by the context *pGFp*.

GFpMul

Multiplies two elements of the finite field.

Syntax

```
IppStatus ippsGFpMul(const IppsGFpElement* pA, const IppsGFpElement* pB,
IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

ippcp.h

Parameters

pA	Pointer to the context of the first multiplicand element of the finite field.
pB	Pointer to the context of the second multiplicand element of the finite field.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the product of two elements of the finite field. The following pseudocode represents this operation: $R = A \cdot B$. The finite field is specified by the context *pGFp*.

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 either IppsGFpState or IppsGFpElement context parameters do not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if *pA* or *pB* does not belong to the finite field specified by the context *pGFp*.

GFpSqr

Computes the square of an element of the finite field.

Syntax

```
IppStatus ippsGFpSqr(const IppsGFpElement* pA, IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

ippcp.h

Parameters

pA	Pointer to the context of the finite field element.
pR	Pointer to the context of the resulting element of the finite field.

pGFp	Pointer to the context of the finite field.

Description

This function computes the square of a given element of the finite field. The following pseudocode represents this operation: $R = A^2$. The finite field is specified by the context *pGFp*.

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 IppsGFpState and IppsGFpElement context parameters does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if *pA* does not belong to the finite field specified by the context *pGFp*.

GFpExp

Raises an element of the finite field to the specified power.

Syntax

```
IppStatus ippsGFpExp(const IppsGFpElement* pA, const IppsBigNumState* pE,
IppsGFpElement* pR, IppsGFpState* pGFp, Ipp8u* pScratchBuffer);
```

Include Files

ippcp.h

Parameters

pA	Pointer to the context of the element of the finite field representing the base of the exponentiation.
pE	Pointer to the Big Number context storing the exponent.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.
pScratchBuffer	Pointer to the scratch buffer.

Description

This function raises the element of the finite field to the given non-negative power. The following pseudocode represents this operation: $R = A^E$. The finite field is specified by the context *pGFp*. You can get the size of the scratch buffer by calling the function GFpScratchBufferSize.

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 IppsGFpState, IppsBigNumState, and IppsGFpElement context parameters does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if *pA* or *pR* does not belong to the finite field specified by the context *pGFp*.

GFpMultiExp

Multiplies exponents of two elements of the finite field.

Syntax

```
IppStatus ippsGFpMultiExp(const IppsGFpElement* const ppElmA[], const IppsBigNumState*
const ppE[], int nItems, IppsGFpElement* pElmR, IppsGFpState* pGF, Ipp8u*
pScratchBuffer);
```

Include Files

ippcp.h

Parameters

ppElmA	Pointer to the array of contexts of the finite field elements representing the base of the exponentiation.
ppE	Pointer to the array of the Big Number contexts storing the exponents.
nItems	Number of exponents.
pElmR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.
pScratchBuffer	Pointer to the scratch buffer.

Description

This function multiplies exponents of elements of the finite field. The finite field is specified by the context *pGFp*. You can get the size of the scratch buffer by calling the ippsGFpScratchBufferSize 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 any of the context parameters IppsGFpState, IppsBigNumState, and IppsGFpElement does not match the operation.
ippStsOutOfRangeErr	Indicates an error condition if any of the elements of *ppElmA* do not belong to the finite field specified by the context *pGFp*.
ippStsBadArgErr	Indicates an error condition if *nItems* is less than 1 or greater than 6.

GFpAdd_PE

Computes the sum of an element of the finite field and an element of its parent field.

Syntax

```
IppStatus ippsGFpAdd_PE(const IppsGFpElement* pA, const IppsGFpElement* pParentB,
IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the first element of the finite field to be added.
pB	Pointer to the context of the second element to be added, which is an element of the parent finite field.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

The function computes the sum of the elements of the finite field specified by the context *pGFp* and its ground finite field. The following pseudocode represents this operation: *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`.
`ippStsContextMatchErr`	Indicates an error condition if any of `IppsGFpState` or `IppsGFpElement` context parameter does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition in the following cases: • the element *pA* does not belong to the finite field specified by the context *pGFp*. • the element *pB* does not belong to the ground field of the finite field specified by the context *pGFp*.
`ippStsBadArgErr`	Indicates an error condition if the context *pGFp* does not specify a prime field.

GFpSub_PE

Subtracts an element of the finite field from an element of its parent field.

Syntax

```
IppStatus ippsGFpSub_PE(const IppsGFpElement* pA, const IppsGFpElement* pParentB,
IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the minuend, an element of the finite field.
pB	Pointer to the context of the subtrahend, an element of the parent finite field.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the difference of the elements of the finite field specified by the context *pGFp* and its ground finite field. The following pseudocode represents this operation: *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`.
`ippStsContextMatchErr`	Indicates an error condition if any of `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition in the following cases: • The element *pA* does not belong to the finite field specified by the context *pGFp*. • The element *pB* does not belong to the ground field of the finite field specified by the context *pGFp*.
`ippStsBadArgErr`	Indicates an error condition if the context *pGFp* does not specify a prime field.

GFpMul_PE

Multiplies an element of the finite field and an element of its parent field.

Syntax

```
IppStatus ippsGFpMul_PE(const IppsGFpElement* pA, const IppsGFpElement* pParentB,
IppsGFpElement* pR, IppsGFpState* pGFp);
```

Include Files

`ippcp.h`

Parameters

pA	Pointer to the context of the first multiplicand, an element of the finite field.

pB	Pointer to the context of the second multiplicand, an element of the parent finite field.
pR	Pointer to the context of the resulting element of the finite field.
pGFp	Pointer to the context of the finite field.

Description

This function computes the product of the element *pA* of the finite field specified by the context *pGFp* and the element *pB* of its ground finite field. The following pseudocode represents this operation: $R = A \cdot B$.

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 any of the `IppsGFpState` and `IppsGFpElement` context parameters does not match the operation.
`ippStsOutOfRangeErr`	Indicates an error condition in the following cases: • The element *pA* does not belong to the finite field specified by the context *pGFp*. • The element *pB* does not belong to the ground field of the finite field specified by the context *pGFp*.
`ippStsBadArgErr`	Indicates an error condition if the context *pGFp* does not specify a prime field.

Support Functions and Classes

A

This appendix contains miscellaneous information on support functions and classes that may be helpful to users of the Intel® Integrated Performance Primitives (Intel® IPP) Cryptography.

The Version Information Function section describes an Intel IPP Cryptography 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 document chapters.

Version Information Function

GetLibVersion

Returns information about the active version of the Intel IPP software for cryptography.

Syntax

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

Include Files

`ippcp.h`

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.miror*) 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_l.lib", *Version* = "7.0 build 205.68", *BuildDate* = "Jul 20 2011".

Example

The code example below shows how to use the function `ippcpGetLibVersion`.

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

```
printf("%s %s %d.%d.%d.%d\n", lib->Name, lib->Version, lib->major, lib->minor, lib->majorBuild,
lib->build);
```

```
}
```

Output:

```
ippcp_l.lib 7.0 build 205.68
```

Other Functions

GetCpuFeatures

Retrieves the processor features.

Syntax

`IppStatus ippcpGetCpuFeatures(Ipp64u* pFeaturesMask);`

Include Files

`ippcp.h`

Parameters

pFeaturesMask	Pointer to the features mask. Possible value is `ippCPUID_GETINFO_A`.

Description

This function retrieves some of the CPU features returned by the function CPUID.1 and stores them consecutively in the mask *pFeaturesMask*. The following table lists the features stored in the mask.

Mask Value	Bit Name	Feature	Mask Bit Number
0x00000001	`ippCPUID_MMX`	MMX™ technology	0
0x00000002	`ippCPUID_SSE`	Intel® Streaming SIMD Extensions	1
0x00000004	`ippCPUID_SSE2`	Intel® Streaming SIMD Extensions 2	2
0x00000008	`ippCPUID_SSE3`	Intel® Streaming SIMD Extensions 3	3
0x00000010	`ippCPUID_SSSE3`	Supplemental Streaming SIMD Extensions	4
0x00000020	`ippCPUID_MOVBE`	MOVBE instruction is supported	5
0x00000040	`ippCPUID_SSE41`	Intel® Streaming SIMD Extensions 4.1	6

Mask Value	Bit Name	Feature	Mask Bit Number
0x00000080	ippCPUID_SSE42	Intel® Streaming SIMD Extensions 4.2	7
0x00000100	ippCPUID_AVX	The processor supports Intel® Advanced Vector Extensions (Intel® AVX) instruction set	8
0x00000200	ippAVX_ENABLEDBYOS	The operating system supports Intel® AVX	9
0x00000400	ippCPUID_AES	Advanced Encryption Standard (AES) instructions are supported	10
0x00000800	ippCPUID_CLMUL	PCLMULQDQ instruction is supported	11
0x00002000	ippCPUID_RDRAND	Read Random Number instructions are supported	13
0x00004000	ippCPUID_F16C	16-bit floating point conversion instructions are supported	14
0x00008000	ippCPUID_AVX2	Intel® Advanced Vector Extensions 2 (Intel® AVX2) instruction set is supported	15
0x00010000	ippCPUID_ADCOX	ADCX and ADOX instructions are supported	16
0x00020000	ippCPUID_RDSEED	Read Random SEED instruction is supported.	17
0x00040000	ippCPUID_PREFETCHW	PREFETCHW instruction is supported	18
0x00080000	ippCPUID_SHA	Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) are supported	19
0x00100000	ippCPUID_AVX512F	Intel® Advanced Vector Extensions 512 (Intel® AVX-512)	20

Mask Value	Bit Name	Feature	Mask Bit Number
		foundation instructions are supported	
0x00200000	ippCPUID_AVX512CD	Intel® AVX-512 conflict detection instructions are supported	21
0x00400000	ippCPUID_AVX512ER	Intel® AVX-512 exponential and reciprocal instructions are supported	22
0x80000000	ippCPUID_KNC	Intel® Xeon Phi™ is supported	23

NOTE
Intel® Itanium® processors are not supported.

Optimization Notice

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

Notice revision #20110804

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition when the *pFeaturesMask* pointer is `NULL`.
`ippStsNotSupportedCpu`	Indicates that the processor is not supported.

SetCpuFeatures

Sets the processor-specific library code for the specified processor features.

Syntax

```
IppStatus ippcpSetCpuFeatures(Ipp64u cpuFeatures);
```

Include Files

`ippcp.h`

Parameters

cpuFeatures	Features to be supported by the library. Refer to `ippcpdefs.h` for `ippCPUID_xx` definition.

Description

This function sets the processor-specific code of the Intel IPP Cryptography library according to the processor features specified in *cpuFeatures*. You can use the following predefined sets of features (the `FM` suffix below means *feature mask*):

32-bit code:

```
#define PX_FM ( ippCPUID_MMX | ippCPUID_SSE )
#define W7_FM ( PX_FM | ippCPUID_SSE2 )
#define V8_FM ( W7_FM | ippCPUID_SSE3 | ippCPUID_SSSE3 )
#define S8_FM ( V8_FM | ippCPUID_MOVBE )
#define P8_FM ( V8_FM | ippCPUID_SSE41 | ippCPUID_SSE42 | ippCPUID_AES | ippCPUID_CLMUL |
ippCPUID_SHA )
#define G9_FM ( P8_FM | ippCPUID_AVX | ippAVX_ENABLEDBYOS | ippCPUID_RDRAND | ippCPUID_F16C )
#define H9_FM ( G9_FM | ippCPUID_MOVBE | ippCPUID_AVX2 | ippCPUID_ADCOX | ippCPUID_RDSEED |
ippCPUID_PREFETCHW )
```

64-bit code:

```
#define PX_FM ( ippCPUID_MMX | ippCPUID_SSE | ippCPUID_SSE2 )
#define M7_FM ( PX_FM | ippCPUID_SSE3 )
#define U8_FM ( M7_FM | ippCPUID_SSSE3 )
#define N8_FM ( U8_FM | ippCPUID_MOVBE )
#define Y8_FM ( U8_FM | ippCPUID_SSE41 | ippCPUID_SSE42 | ippCPUID_AES | ippCPUID_CLMUL |
ippCPUID_SHA )
#define E9_FM ( Y8_FM | ippCPUID_AVX | ippAVX_ENABLEDBYOS | ippCPUID_RDRAND | ippCPUID_F16C )
#define L9_FM ( E9_FM | ippCPUID_MOVBE | ippCPUID_AVX2 | ippCPUID_ADCOX | ippCPUID_RDSEED |
ippCPUID_PREFETCHW )
#define K0_FM ( L9_FM | ippCPUID_AVX512F )
```

NOTE
Do not use any other Intel IPP Cryptography function while `ippcpSetCpuFeatures` is executing. Otherwise, your application behavior is undefined.

NOTE
To avoid initialization of internal structures for one Intel® architecture and then call of the processing function that is optimized for another architecture, do not use the `ippcpSetCpuFeatures` function in chains of Intel IPP Cryptography connected calls like <*processing function*`GetSize` + <*processing function*`Init` + <*processing function*>. Otherwise, Intel IPP Cryptography functionality behavior is undefined.

Intel IPP Cryptography library supports two internal sets of CPU features:

- *Real CPU features*: the features that are supported by the CPU at which the library is executed. These features are read-only and can be obtained with the `ippcpGetCpuFeatures` function.
- *Enabled features*: the features that are enabled externally to Intel IPP Cryptography by the application. These features can be set with `ippcpSetCpuFeatures`.

The `ippcpSetCpuFeatures` function provides additional flexibility in measuring performance improvements reached by using specific CPU features. For example, the first call of any Intel IPP Cryptography function in an application running on the 4th Generation Intel® Core™ i7 processor with 64-bit OS installed dispatches the L9 code version optimized for Intel® Advanced Vector Extensions 2 (Intel® AVX2) with several other features like fast 16-bit floating point support, Intel® AES New Instructions (Intel® AES-NI), PCLMULQDQ new instructions support.

To check performance improvement for all Intel IPP Cryptography functionality reached by using Intel® AVX2, you can run a benchmark for the currently dispatched version of code and then compare performance with the Intel® Advanced Vector Extensions (Intel® AVX) version of code with Intel® AVX2 disabled. To disable Intel AVX2, call `ippcpSetCpuFeatures(E9_FM)`. To enable Intel AVX2 back, call `ippcpSetCpuFeatures( L9_FM )`. Thus, you can use the `ippcpSetCpuFeatures` function to dispatch any version of Intel IPP Cryptography code and enable/disable specific CPU features. If you are not well familiar with the features of your CPU, use the auto-initialization mechanism for the default library behavior.

Optimization Notice

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

Notice revision #20110804

Return Values

`ippStsNoErr`	Indicates that the required processor-specific code is successfully set.
`ippStsCpuMismatch`	Indicates that the specified processor features are not valid. Previously set code is used. If the requested feature is below the minimal supported by the **px** library - that is Intel® Streaming SIMD Extensions (Intel® SSE) for IA-32 and Intel® SSE2 for Intel® 64 architecture, **px** code is dispatched.
`ippStsFeatureNotSupported`	Indicates that the current CPU does not support at least one of the requested features. If the `ippCPUID_NOCHECK` bit of the *`cpuFeatures`* parameter is set to 1, these not supported features are enabled, otherwise - disabled.
`ippStsUnknownFeature`	Indicates that at least one of the requested features is unknown. It means that the feature is not defined in the `ippdefs.h` file. Further behavior of the library depends on known features passed to *`cpuFeatures`*. Unknown features are ignored.
`ippStsFeaturesCombination`	Indicates that the combination of features is not correct. For example, `ippCPUID_AVX2` bit is set to 1 in *`cpuFeatures`*, but at least one of the `ippCPUID_MMX`, `ippCPUID_SSE`, …, `ippCPUID_AVX` bits is not set. All these missing bits, if supported by CPU, are set to 1. This means that if the library supports the Intel® AVX2 code, it also internally uses all known MMX™, Intel® SSE, and Intel® AVX extensions, which are below Intel® AVX2.

GetNumThreads

Returns the number of existing threads in the multithreading environment.

Syntax

```
IppStatus ippcpGetNumThreads(int* pNumThr);
```

Include Files

`ippcp.h`

Parameters

pNumThr	Pointer to the number of threads.

Description

This function returns the number of OpenMP* threads specified by the user previously. If it is not specified, the function returns the initial number of threads that depends on the number of logical processors.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsNullPtrErr`	Indicates an error condition when the *pNumThr* pointer is `NULL`.
`ippStsNoOperation`	Indicates that there is no such operation in the static version of the library.

SetNumThreads

Sets the number of threads in the multithreading environment.

Syntax

```
IppStatus ippcpSetNumThreads(int numThr);
```

Include Files

`ippcp.h`

Parameters

numThr	Number of threads, should be more than zero.

Description

This function sets the number of OpenMP* threads. A number of established threads may be less than specified *numThr*.

Return Values

`ippStsNoErr`	Indicates no error.
`ippStsSizeErr`	Indicates an error when *numThr* is less than, or equal to zero.

`ippStsNoOperation`	Indicates that the function is called from the application linked to the single-threaded version of the library. No operation is performed.

GetStatusString

Translates a status code into a message.

Syntax

`const char* ippcpGetStatusString(IppStatus stsCode);`

Include Files

`ippcp.h`

Parameters

stsCode	Code that indicates the status type.

Description

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

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:

```
#if !defined _BIGNUMBER_H_
#define _BIGNUMBER_H_

#include "ippcp.h"

#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

class BigNumber
{
public:
   BigNumber(Ipp32u value=0);
   BigNumber(Ipp32s value);
   BigNumber(const IppsBigNumState* pBN);
   BigNumber(const Ipp32u* pData, int length=1, IppsBigNumSGN sgn=IppsBigNumPOS);
   BigNumber(const BigNumber& bn);
```

```
BigNumber(const char *s);
virtual ~BigNumber();

// set value
void Set(const Ipp32u* pData, int length=1, IppsBigNumSGN sgn=IppsBigNumPOS);
// conversion to IppsBigNumState
friend IppsBigNumState* BN(const BigNumber& bn) {return bn.m_pBN;}
operator IppsBigNumState* () const { return m_pBN; }

// some useful constatns
static const BigNumber& Zero();
static const BigNumber& One();
static const BigNumber& Two();

// arithmetic operators probably need
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.

```
#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;
   ippsRef_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
//
BigNumber& 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;
}

BigNumber& 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;
}

BigNumber& 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;
}

BigNumber& 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);
   *this = result;
   return *this;
}

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);
   return q /= b;
}

BigNumber operator % (const BigNumber& a, const BigNumber& b )
{
   BigNumber r(b);
   ippsMod_BN(BN(a), BN(b), BN(r));
   return r;
}

// modulo arithmetic
//
BigNumber BigNumber::Modulo(const BigNumber& a) const
{
   return a % *this;
}

BigNumber BigNumber::InverseAdd(const BigNumber& a) const
{
   BigNumber t = Modulo(a);
   if(t==BigNumber::Zero())
      return t;
   else
   return *this - t;
}

BigNumber BigNumber::InverseMul(const BigNumber& a) const
{
   BigNumber r(*this);
   ippsModInv_BN(BN(a), BN(*this), BN(r));
   return r;
}

BigNumber BigNumber::ModAdd(const BigNumber& a, const BigNumber& b) const
{
   BigNumber r = this->Modulo(a+b);
   return r;
}

BigNumber BigNumber::ModSub(const BigNumber& a, const BigNumber& b) const
{
   BigNumber r = this->Modulo(a + this->InverseAdd(b));
   return r;
```

```
}

BigNumber BigNumber::ModMul(const BigNumber& a, const BigNumber& b) const
{
   BigNumber r = this->Modulo(a*b);
   return r;
}

// comparison
//
int BigNumber::compare(const BigNumber &bn) const
{
   Ipp32u result;
   BigNumber tmp = *this - bn;
   ippsCmpZero_BN(BN(tmp), &result);
   return (result==IS_ZERO)? 0 : (result==GREATER_THAN_ZERO)? 1 : -1;
}

bool operator <  (const BigNumber &a, const BigNumber &b) { return a.compare(b) < 0; }
bool operator >  (const BigNumber &a, const BigNumber &b) { return a.compare(b) > 0; }
bool operator == (const BigNumber &a, const BigNumber &b) { return 0 == a.compare(b);}
bool operator != (const BigNumber &a, const BigNumber &b) { return 0 != a.compare(b);}

// easy tests
//
bool BigNumber::IsOdd() const
{
   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;
         }
         break;
      }
   }
   return msb;
}

int Bit(const vector<Ipp32u>& v, int n)
{
   return 0 != ( v[n>>5] & (1<<(n&0x1F)) );
}

// conversions and output
//
void BigNumber::num2vec( vector<Ipp32u>& v ) const
{
   int bnBitLen;
   Ipp32u* bnData;
   ippsRef_BN(NULL, &bnBitLen, &bnData, *this);

   int len = BITSIZE_WORD(bnBitLen);;
   for(int n=0; n<len; n++)
      v.push_back( bnData[n] );
}

void BigNumber::num2hex( string& s ) const
{
   IppsBigNumSGN bnSgn;
   int bnBitLen;
   Ipp32u* bnData;
   ippsRef_BN(&bnSgn, &bnBitLen, &bnData, *this);

   int len = BITSIZE_WORD(bnBitLen);

   s.append(1, (bnSgn==ippBigNumNEG)? '-' : ' ');
   s.append(1, '0');
   s.append(1, 'x');
   for(int n=len; n>0; n--) {
      Ipp32u x = bnData[n-1];
      for(int nd=8; nd>0; nd--) {
         char c = HexDigitList[(x>>(nd-1)*4)&0xF];
```

```
        s.append(1, c);
      }
   }
}

ostream& operator << ( ostream &os, const BigNumber& a)
{
   string s;
   a.num2hex(s);
   os << s.c_str();
   return os;
}
```

Functions for Creation of Cryptographic Contexts

The section presents source code for creation of some cryptographic contexts.

Declarations

Contents of the header file (`xsample_cpobjs.h`) declaring functions for creation of some cryptographic contexts are presented below:

```
#if !defined _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.

```
#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;
   ippsBigNumGetSize(len, &size);
   IppsBigNumState* pBN = (IppsBigNumState*)( new Ipp8u [size] );
   ippsBigNumInit(len, pBN);
   if(pData)
      ippsSet_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; }
```

Bibliography

This bibliography provides a list of publications that might be helpful to you in using cryptography functions of Intel IPP.

[3GPP 35.202] *3GPP TS 35.202 V3.1.1 (2001-07). 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; Specification of the 3GPP Confidentiality and Integrity Algorithms; 3G Security; Document 2: KASUMI Specification (Release 1999)*. Available from http://isearch.etsi.org/3GPPSearch/isysquery/403fe057-469e-46a4-b298-f80b78bf4343/3/doc/35202-311.pdf.

[3GPP 2006] *Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification*. September 2006. Available from http://www.gsmworld.com/using/algorithms/docs/snow_3g_spec.pdf.

[AC] Schneier, Bruce. *Applied Cryptography. Protocols, Algorithms, and Source Code in C*. Second Edition. John Wiley & Sons, Inc., 1996.

[AES] Daemen, Joan, and Vincent Rijmen. *The Rijndael Block Cipher. AES Proposal*. Available from http://www.nist.gov/aes.

[ANSI] *ANSI X9.62-1998 Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)*. American Bankers Association, 1999.

[ANT] Cohen, Henri. *A Course in Computational Algebraic Number Theory*. Springer, 1998.

[EC] Koblitz, Neal. *Introduction to Elliptic Curves and Modular Forms*. Springer, 1993.

[EHCC] Cohen, Henri, and Gerald Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryprography*. Chapman & Hall/CRC, 2006.

[FIPS PUB 46-3] *Federal Information Processing Standards Publications, FIPS PUB 46-3*. Data Encryption Standard (DES), October 1999. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 113] *Federal Information Processing Standards Publications, FIPS PUB 113*. Computer Data Authentication, May 1985. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 180-2] *Federal Information Processing Standards Publications, FIPS PUB 180-2*. Secure Hash Standard, August 2002. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 180-4] *Federal Information Processing Standards Publications, FIPS PUB 186-2*. Secure Hash Standard (SHS), March 2012. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 186-2] *Federal Information Processing Standards Publications, FIPS PUB 186-2*. Digital Signature Standard (DSS), January 2000. Available from http://csrc.nist.gov/publications/fips.

[FIPS PUB 198-1] *Federal Information Processing Standards Publications, FIPS PUB 198*. The Key-Hash Message Authentication Code (HMAC), July 2008. Available from http://csrc.nist.gov/publications/fips.

[IEEE P1363A] *Standard Specifications for Public-Key Cryptography: Additional Techniques*. May, 2000. Working Draft.

[IEEE P1619] *IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*. April 2008.

[INTEL ARCH] *Intel® 64 and IA-32 Architectures Software Developer's Manual* . Volume 1: Basic Architecture. Available from http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf.

[ISO/IEC 11889-4] *ISO/IEC 11889-4:2015* Information technology - TPM Library - Part 4: Supporting Routines.

[NIST SP 800-38A] *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*. NIST Special Publication 800-38A, December 2001. Available from http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[NIST SP 800-38B] *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. NIST Special Publication 800-38B, May 2005. Available from http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pd

[NIST SP 800-38C] *Draft Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. NIST Special Publication 800-38C, September 2003. Available from http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf.

[NIST SP 800-38D] *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D, November 2007. Available from http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[PKCS 1.2.1] *RSA Laboratories. PKCS #1 v2.1: RSA Cryptography Standard*. June 2002. Available from http://www.rsasecurity.com/rsalabs/pkcs.

[PKCS 7] *RSA Laboratories. PKCS #7: Cryptographic Message Syntax Standard*. An RSA Laboratories Technical Note Version 1.5 Revised, November 1, 1993.

[RC5] Rivest, Ronald L. *The RC5 Encryption Algorithm*. Proceedings of the 1994 Leuven Workshop on Algorithms (Springer), 1994. Revised version, dated March 1997, is available from http://theory.lcs.mit.edu/~cis/pubs/rivest/rc5rev.ps.

[RFC 1321] Rivest, Ronald L. *The MD5 Message-Digest Algorithm*. RFC 1321, MIT and RSA Data Security, Inc, April 1992. Available from http://www.faqs.org/rfc1321.html.

[RFC 2401] Krawczyk, Hugo, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2401, February 1997. Available from http://www.faqs.org/rfcs/rfc2401.html.

[RFC 3566] Frankel, Sheila, and Howard C. Herbert. *The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec*. RFC 3566, September 1996. Available from http://www.rfc-archive.org/getrfc.php?rfc=3566.

[RFC 5297] D. Harkins. *Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)*. RFC 5297, October 2008. Available from https://tools.ietf.org/pdf/rfc5297.pdf.

[SEC1] *SEC1: Elliptic Curve Cryptography.* Standards for Efficient Cryptography Group, September 2000. Available from http://www.secg.org/secg_docs.htm.

[SEC2] *SEC2: Recommended Elliptic Curve Domain Parameters.* Standards for Efficient Cryptography Group, September 2000. Available from http://www.secg.org/secg_docs.htm/.

[SM2] *SM2 Digital Signature Algorithm.* Available from http://tools.ietf.org/html/draft-shen-sm2-ecdsa-01.

[SM3] *SM3 Hash Function.* Available from https://tools.ietf.org/html/draft-shen-sm3-hash-00.

[SMS4] *SMS4 Encryption Algorithm for Wireless Networks.* Available from http://www.oscca.gov.cn/UpFile/200621016423197990.pdf (Chinese) and http://eprint.iacr.org/2008/329.pdf (English).

[X9.42] *X9.42-2003: Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography.* American National Standards Institute, 2003.

Index

A

B

C

D

E

F

G

H

I

K

M

N

O

P

R

S

T

V