# Security and Cryptography

Exploring Palm OS

Written by Greg Wilson
Edited by Jean Ostrem
Technical assistance from Richard Levenberg and Ricardo Lagos



Exploring Palm OS: Security and Cryptography
Document Number 3113-003
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

# Table of Contents

# Part II: Reference

## 13 Password 295



## 14 Security Services 299

# About This Document

This book covers the various security systems in Palm OS Cobalt. It also documents the various cryptographic operations that Palm OS Cobalt provides. Developers who need to work with security certificates, passwords, and the like should read this book.

## The *Exploring Palm OS* Series

This book is a part of the *Exploring Palm OS* series. Together, the books in this series document and explain how to use the APIs exposed to third-party developers by the fully ARM-native versions of Palm OS, beginning with Palm OS Cobalt. Each of the books in the *Exploring Palm OS* series explains one aspect of the Palm operating system, and contains both conceptual and reference documentation for the pertinent technology.

---

**IMPORTANT:** The *Exploring Palm OS* series is intended for developers creating native applications for Palm OS Cobalt. If you are interested in developing applications that work through PACE and that also run on earlier Palm OS releases, read the latest versions of the *Palm OS Programmer's API Reference* and *Palm OS Programmer's Companion* instead.

---

As of this writing, the complete *Exploring Palm OS* series consists of the following titles:

- *Exploring Palm OS: Programming Basics*
- *Exploring Palm OS: Memory, Databases, Files*
- *Exploring Palm OS: User Interface*
- *Exploring Palm OS: User Interface Guidelines* (coming soon)
- *Exploring Palm OS: System Management*
- *Exploring Palm OS: Text and Localization*
- *Exploring Palm OS: Input Services*
- *Exploring Palm OS: High-Level Communications*

- *Exploring Palm OS: Low-Level Communications*
- *Exploring Palm OS: Telephony and SMS*
- *Exploring Palm OS: Multimedia*
- *Exploring Palm OS: Security and Cryptography*
- *Exploring Palm OS: Creating a FEP* (coming soon)
- *Exploring Palm OS: Porting Applications to Palm OS Cobalt*
- *Exploring Palm OS: Palm OS File Formats* (coming soon)

# Additional Resources

- Documentation

  PalmSource publishes its latest versions of this and other documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/

- Training

  PalmSource and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

  http://www.palmos.com/dev/support/kb/

# Changes to This Document

This section describes the changes made in each version of this document.

## 3113-002

Minor editorial corrections.

## 3113-001

The first release of this document for Palm OS Cobalt, version 6.0.

# Part I
# Concepts

This part provides basic concepts for the security-related portions of Palm OS Cobalt. The conceptual material in this part is organized into the following chapters:

# 1

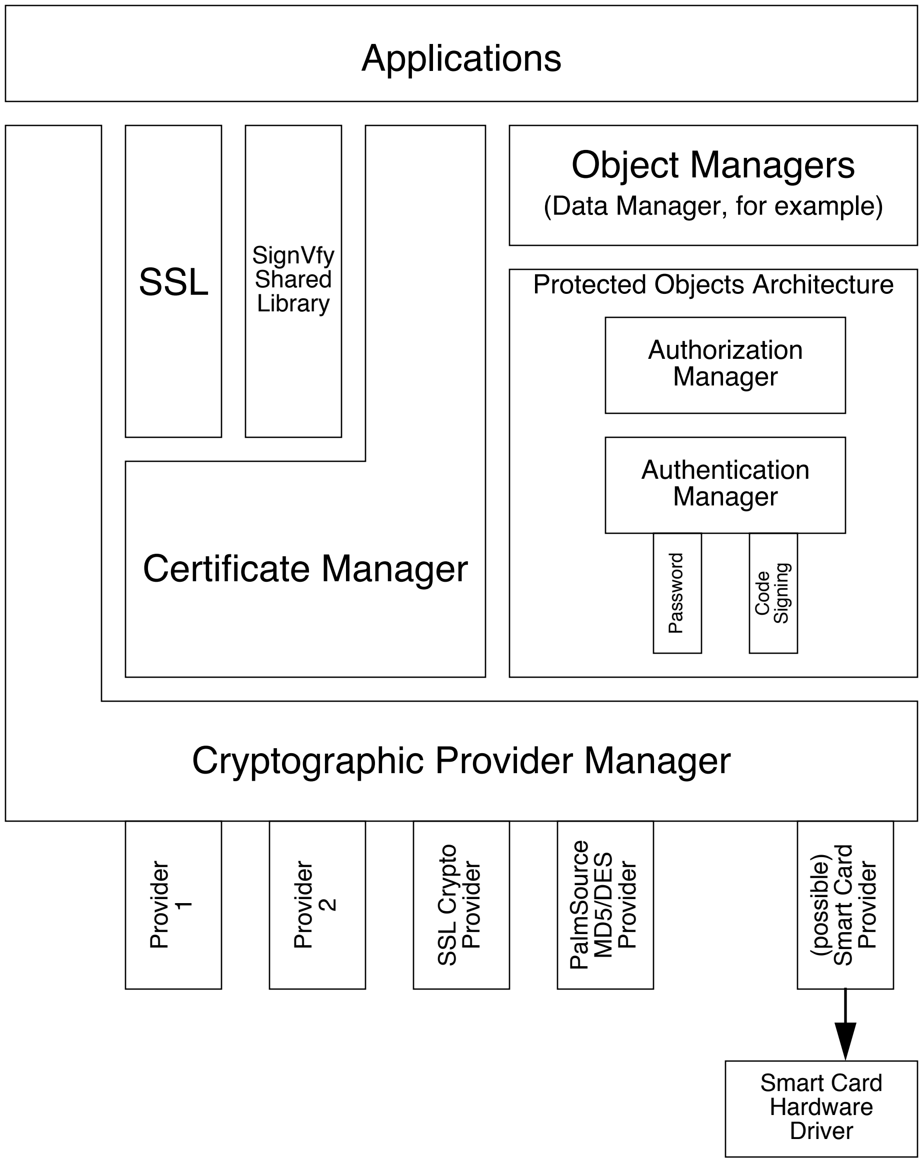
# Palm OS Cobalt Security

Palm OS Cobalt has a robust and comprehensive security architecture. Unlike other security solutions that are added in an ad-hoc manner to existing operating systems, the security for Palm OS Cobalt has been designed in from the beginning.

The basis of Palm OS Cobalt security is the secure kernel. The kernel relies on a capabilities model for security. The capabilities model, typically a model of least privilege, has been hybridized for Palm OS Cobalt in order to maintain the open nature of Palm OS. At boot time, keys are carefully distributed to various system managers that need to communicate with each other. Only managers that have keys enabling communication are able to communicate with other system components. This prevents unauthorized access to important system modules.

On top of the secure kernel are the components that make up the basis of a secure infrastructure. Figure 1.1 illustrates how these components interrelate.

**Figure 1.1 Palm OS Cobalt security components**




The **Authorization Manager** (AM) provides system managers with the ability to protect managed resources. In the case of Data Manager, database resources are protected via the Authorization Manager. Other components, such as drivers, may protect other resources such as network access through the Authorization Manager. In conjunction with the Authorization Manager the

**Authentication Manager** (AM) provides authentication for authorization rules and supports other authentication requirements. The Authentication Manager employs a plug-in architecture so that additional authentication mechanisms can be added. The Authentication Manager supports password/pass phrase authentication for users and both PKI and cryptographic fingerprint for code modules. Additional authentication mechanisms, such as biometric, can be added via the Authentication Manager plug-in framework.

Palm OS Cobalt also includes the **Cryptographic Provider Manager** (CPM). The Cryptographic Provider Manager exposes a simple yet robust API for performing cryptographic operations including key generation, hashing, encryption, decryption, signing, and verification. The Cryptographic Provider Manager includes a FIPS–approved pseudo random number generator and a provider architecture for cryptographic algorithms. The default provider, developed by RSA Security, includes RC4 (128 bit), SHA-1 hashing, and RSA public key operations (1024 bit). Additional providers can be added, either statically by the licensee or dynamically, to the Cryptographic Provider Manager to support other algorithms.

Palm OS Cobalt includes a **Certificate Manager**, developed by RSA Security. The Certificate Manager handles X.509 standard certificates. The Certificate Manager exposes a standard API for applications and system modules that need certificate services.

Making use of both the Cryptographic Provider Manager and the Certificate Manager, Palm OS Cobalt includes a **Signature Verification Library** that allows applications and system modules to easily verify signatures on code modules and resources.

Palm OS Cobalt also includes a robust and highly optimized version of SSL for end-to-end secure communications. The Palm OS Cobalt SSL implementation, by RSA Security, supports SSL v2, v3, and TLS 1.0.

The Palm OS Cobalt **Security Services module** that supports a variety of mechanisms for specifying and controlling the security policies of a particular device or class of devices. Security Services supports a policy API that applications and code modules can query to get policies for various operations or functionality. Examples include policies for what types of patches are allowed on the system and polices for what drivers are allowed on the system. The Security Services also supports a set of APIs so that the user can indicate a perceived level of security of the device (None, Medium, or High). Various modules and applications can read the user's security preference and react accordingly.

PalmSource signs all shared library code modules. Code modules easily support multiple signatures, enabling licensees and carriers to sign code modules.

The following sections provide details on each of the Palm OS Cobalt security components. Note that the details of SSL are covered in Chapter 2, "SSL Concepts," on page 55.

# Cryptographic Provider Manager (CPM)

The Cryptographic Provider Manager (CPM) provides an interface for cryptographic related functions. At its heart it contains an X 9.31 FIPS–approved pseudo random number generator. The CPM allows you to do the following:

- RC4 encrypt/decrypt
- SHA1 digest
- RSA verify
- DES/MD5 encrypt/decrypt

The CPM also supports the SSL cryptographic package. See Chapter 2, "SSL Concepts," on page 55 for more on SSL in Palm OS Cobalt.

Note that the CPM is export controlled. CPM providers must be signed.

The Cryptographic Provider Manager, or CPM, provides an easy to use, yet robust cryptographic API. Applications can use the CPM to perform cryptographic operations for data and protocols. Under the CPM there are one or more providers which supply the actual cryptographic functionality via the CPM API.

The CPM supports various classes of functions, some of which are described herein. For further information see the CPM documentation.

The classes of functions discussed herein can be grouped as follows:



## Provider Information and Manipulation

The CPM itself provides the ability to query the number of providers present, modify the order the providers are called, and a pseudo random number generator based on ANSI X9.31

The CPM contains a default provider which includes SHA-1 hashing, RC4 encryption and decryption, and RSA verification (public key operations). Other providers may or may not be present.

The operation of the CPM is different from other cryptographic providers that perform the same types of operations. Applications that use the CPM can get default "reasonable" behavior without specifying a great deal of information related to the cryptographic operation requested. Examples will illustrate this.

Listing 1.1 shows how to enumerate and identify the providers the CPM currently knows about.

#### Listing 1.1 Enumerating CPM providers

```
uint32_t *providers;
APProviderInfoType providerInfo;
uint16_t temp = 0;
status_t err;

err = CPMLibOpen(&temp);

if (err) {
   DbgPrintf("SSFD: Error on CPMLibOpen 0x%x\n", err);
} else {
   DbgPrintf("SSFD: CPMLibOpen - 0x%x providers returned\n",
      temp providers = MemPtrNew(sizeof(uint32_t) * temp);
  err = CPMLibEnumerateProviders(providers, &temp);

   providers = MemPtrNew(sizeof(uint32_t) * temp);
   err = CPMLibEnumerateProviders(providers, &temp);
```

```
    if (err) {
        DbgPrintf("SSFD: Error on CPMLibEnumerateProviders
            0x%x\n", err);
    } else {
        DbgPrintf("SSFD: CPMLibEnumerateProviders - 0x%x
            providers returned\n", temp);
        for (i=0; i < temp; i++) {
            err = CPMLibGetProviderInfo(providers[i],
                &providerInfo);
            if (err) {
                DbgPrintf("SSFD: Error on CPMLibGetProviderInfo
                    0x%x\n", err);
           } else {
                uint32_t provider provider = providers[i];

                provider = providers[i];
                DbgPrintf("SSFD: CPMLibGetProviderInfo -
                    provider['%c%c%c%c']\n",
                    (char)((provider >> 24) & 0x000000FF),
                    (char)((provider >> 16) & 0x000000FF),
                    (char)((provider >> 8) & 0x000000FF),
                    (char)((provider & 0x000000FF)));
                DbgPrintf("\t%s\n", providerInfo.name);
                DbgPrintf("\t%s\n", providerInfo.other);
                DbgPrintf("\tAlgs: %d\n",
                    providerInfo.numAlgorithms);
                DbgPrintf("\tHardware?: %s\n",
                    providerInfo.bHardware?"yes":"no");
           }
    }
}
```

The order that the providers are called is arbitrary. The CPM orders the providers as they are found and calls each one in turn until one returns that it can handle the request. Subsequent calls on the same context of operations will go to the same provider that handled the first request. For example, say the first operation in an encryption context is generating a key to use for encryption. The provider that handles the key generation will also be used to do the encryption unless the application explicitly changes it.

For any given initial operation, or one for which a provider has not yet been selected, the CPM tries each provider in turn until one returns that the operation has been handled. Subsequent providers are not called. Due to this design, the CPM does not readily handle providers that include similar functionality.

The application is free to select a different provider for any operation for which the provider has already been set. The application is also free to set the first provider that the CPM will call, ensuring that all operations will go to a particular provider for the initial context. To set the default provider, do something like what is shown in Listing 1.2.

### Listing 1.2 Setting the default CPM provider

```
status_t err;

/*
* set the default provider to the provider with id 'foop'
*/
err = CPMLibSetDefaultProvider((uint32_t) 'foop');
if (err)
   DbgPrintf("SSFD: Error on CPMLibSetDefaultProvider
      0x%x\n", err);
```

All `CPMInfoType` structures have a common structure header which includes provider information about the provider that handled the structure. To change the provider for a given `CPMInfoType` structure, the application must copy the structure and reset the provider information. The application is then responsible for making sure that the original structure is passed to the original provider for cleanup. This operation is *not* recommended without specific knowledge of the operation and functionality of the various providers utilized. See Listing 1.3 for an illustration of how this might be done.

### Listing 1.3 Changing the provider

```
status_t err;
APKeyInfoType keyInfo, newkeyInfo;

MemSet(&keyInfo, sizeof(APKeyInfoType), 0);

err = CPMLibGenerateKey(NULL, 0, &keyInfo);

MemSet(&newkeyInfo, sizeof(APKeyInfoType), 0);
```

```
MemMove(&newkeyInfo, &keyInfo, sizeof(APKeyInfoType));
newkeyInfo.providerContext.localContext = NULL;
newkeyInfo.providerContext.providerID = 0;

CPMLibReleaseKeyInfo(&keyInfo);
/* now go on to use newkeyInfo as you please */
```

## Key Functions

The CPM provides several ways of introducing keys to the cryptographic functions. The CPM has concepts of either generating a brand new key, deriving a key from some initial data, or importing a previously generated key.

Generating a new key implies that every time a request for a new key to be generated is made, a brand new key is generated. If this brand new key is utilized for any cryptographic operations, it must be exported and saved in order to be used again. It is statistically improbable that a generated key could be regenerated.

Deriving a key means that given the same input data, the same key is derived from the data. This is useful for operations like Password Based Encryption (PBE) where the password is used to derive a key for a particular cryptographic operations (usually encryption or decryption).

Importing a key means that a previously generated key which was exported and saved by an application is now being imported for further cryptographic operations. Importing a key can also mean that key data from a derive key operation is now being used to create a `APKeyInfoType` object. In general, an application would not export and save a derived key since it could be re-derived by using the same input data. A generated key, however, must be exported and saved if it is to be used for later cryptographic operations.

Note that the CPM is designed to work with very little information about the specific cryptographic operation requested. Especially for data that remains on the originating device, most of the input and output parameters for CPM APIs can be ignored.

To generate a new key (the application does not care about the type of key since it is to be used to encrypt data that remains on the device), do something along the lines of what is shown in Listing 1.4.

### Listing 1.4 Generating a new key

```
status_t err;
APKeyInfoType keyInfo;

MemSet(&keyInfo, sizeof(APKeyInfoType), 0);

err = CPMLibGenerateKey(NULL, 0, &keyInfo);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibGenerateKey 0x%x\n", err);
} else {
   DbgPrintf("SSFD: CPMLibGenerateKey - key of length 0x%x
      returned\n", keyInfo.length);
}
```

The call to generate a key allows the application to specify some data to use as seed data for the pseudo-random number generator before the pseudo-random number generator is used. Listing 1.5 illustrates how to generate a key of a specific type and length.

### Listing 1.5 Generating a key of a specific type and length

```
status_t err;
APKeyInfoType keyInfo;
uint8_t *seedData;
uint32_t seedDataLength;

MemSet(&keyInfo, sizeof(APKeyInfoType), 0);
keyInfo.type = apSymmetricTypeRijndael;
keyInfo.length = 256/8; /* 256 bit key or 32 byte key */

/* provide some seed data to the random generator for
   generating the key */
GetSomeSeedData(seedData, seedDataLength);
err = CPMLibGenerateKey(seedData, seedDataLength, &keyInfo);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibGenerateKey 0x%x\n", err);
} else {
   DbgPrintf("SSFD: CPMLibGenerateKey - key of length 0x%x
      returned\n", keyInfo.length);
}
```

All of the CPM APIs that include an output buffer for results allow the application to specify a `NULL` output buffer and a valid output buffer size pointer to receive the required output buffer size. In this way an application can request the required output buffer size before making the actual call.

Derived keys can have some complicated parameters like iterations and salts which are better described elsewhere. In general, the CPM, and providers will provide "sane" functionality when parameters are left unspecified.

Deriving a key just returns exportable key data. To actually use a derived key, the application must import the key data to get an `APKeyInfoType`. The `APKeyInfoType` is used in subsequent cryptographic operations. See Listing 1.6 for sample code that derives a key.

### Listing 1.6 Deriving a key

```
status_t err;
APDerivedKeyInfoType dki;
APKeyInfoType keyInfo;
uint32_t size;
uint32_t *key_data;

/*
   * this is provider dependent
   */
struct {
   unsigned long length;
   unsigned char *data;
} kdInfo;

MemSet(&dki, sizeof(APDerivedKeyInfoType), 0);
MemSet(&kdInfo, sizeof(kdInfo), 0);
GetUserPassword(kdInfo.data, kdInfo.length)
dki.kdInfo = &kdInfo;
size = 0;
err = CPMLibDeriveKeyData(&dki, NULL, &size);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibDeriveKeyData 0x%x\n",
      err);

if (err == cpmErrBufTooSmall) {
   DbgPrintf("SSFD: cpmErrBufTooSmall with
      CPMLibDeriveKeyData returning %d\n", size);
```

```
   key_data = MemPtrNew(size);
   if (key_data != NULL) {
      err = CPMLibDeriveKeyData(&dki, key_data, &size);
      if (err) {
         DbgPrintf("SSFD: Error on CPMLibDeriveKeyData
            0x%x\n", err);
      } else {
         DbgPrintf("SSFD: CPMLibDeriveKeyData - 0x%x bytes
            returned\n", size
/* now we can use key_data as import data to get a key */
         MemSet(&keyInfo, sizeof(APKeyInfoType), 0);
         err = CPMLibImportKeyInfo(IMPORT_EXPORT_TYPE_RAW,
            key_data, size, &keyInfo);
         if (err) {
            DbgPrintf("SSFD: Error on CPMLibImportKeyInfo
               0x%x\n", err);
         } else {
            DbgPrintf("SSFD: CPMLibImportKeyInfo - key of
               length 0x%x returned\n", keyInfo.length);
         }
      }
   }
}
```

Import is used with keys from various sources such as a saved database, a static application key or a key sourced from a protocol negotiation. Typically an application must specify the type of key that is being imported and the import format[1]. Since import data is essentially raw byte streams, its important that the application specify something.

### Listing 1.7 Importing a key

```
status_t err;
APKeyInfoType keyInfo;
uint8_t key[] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD,
   0xEF };

MemSet(&keyInfo, sizeof(APKeyInfoType), 0);
keyInfo.type = apSymmetricTypeDES;
```

1. A given CPM import/export format, such as XML, is only supported if the provider supports it. IMPORT_EXPORT_TYPE_RAW is always supported.

```
err = CPMLibImportKeyInfo(IMPORT_EXPORT_TYPE_RAW, key,
   sizeof(key), &keyInfo);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibImportKeyInfo 0x%x\n",
      err);
} else {
   DbgPrintf("SSFD: CPMLibImportKeyInfo - key of length 0x%x
      returned\n", keyInfo.length);
}
```

## Message Digest Functions

Message digests or hashes are cryptographically strong one way functions. A one way function yields a series of bits that represent the input message. Nothing about the input message can be gleaned from the message digest. The same input message always generates the same hash. Typically hashes are used slow operations are to be performed on long messages. Rather than performing the slow operation on the entire message, the long operation is performed on the hash of the message which is much shorter.

The CPM has two modes of operation for message digests. One mode takes the input message as whole and outputs a digest. The other mode takes the input message as parts and doesn't output the digest until the final part of the message is submitted.

In the all-in-one-shot mode of operation, no context is required for the hashing operation. The application can safely ignore the `APHashInfoType` parameter for the AIO operations. Listing 1.8 shows how to do a hashing operation in a single pass.

### Listing 1.8 A single-pass hashing operation

```
status_t err;
uint32_t size;
uint8_t data[] = ( 'f', 'o', 'o' );
uint8_t *md;

size = 0;
err = CPMLibHash(apHashTypeSHA1, NULL, data, sizeof(data),
   NULL, &size);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibHash 0x%x\n", err);
```

```
   if (err == cpmErrBufTooSmall) {
      DbgPrintf("SSFD: cpmErrBufTooSmall with CPMLibHash
         returning %d\n", size);
      md = MemPtrNew(size);
      if (md != NULL) {
         err = CPMLibHash(apHashTypeSHA1, NULL, data,
            sizeof(data), md, &size);
      if (err) {
         DbgPrintf("SSFD: Error on CPMLibHash 0x%x\n", err);
      } else {
         DbgPrintf("SSFD: CPMLibHash - 0x%x bytes
            returned\n", size);
      }
   }
}
```

For the multi-part mode of operation, a context is required to pass from initial operation to subsequent operations. Upon return from the final operation the context must be cleaned up. It is the application's responsibility to pass the context to the Release function for cleanup by the CPM and providers. See Listing 1.9 for a sample illustrating the multi-part hashing operation.

#### Listing 1.9 A multi-part hashing operation

```
status_t err;
uint32_t size;
uint8_t data[] = ( 'f', 'o', 'o' );
uint8_t *md;
APHashInfoType hashInfo;

MemSet(&hashInfo, sizeof(APHashInfoType), 0);
hashInfo.type = apHashTypeSHA1;

err = CPMLibHashInit(&hashInfo); /* initialize the context */
if (err) {
   DbgPrintf("SSFD: Error on CPMLibHashInit 0x%x\n", err);
} else {
   /*update the operation; can do this any number of times */
   err = CPMLibHashUpdate(&hashInfo, data, sizeof(data));
   if (err) {
      DbgPrintf("SSFD: Error on CPMLibHashUpdate 0x%x\n",
         err);
   } else {
      size = 0;
      err = CPMLibHashFinal(&hashInfo, NULL, 0, NULL, &size);
```

```
      if (err) {
         DbgPrintf("SSFD: Error on CPMLibHashFinal 0x%x\n",
            err);

         if (err == cpmErrBufTooSmall) {
            DbgPrintf("SSFD: cpmErrBufTooSmall with
                CPMLibHash returning %d\n", size);
            md = MemPtrNew(size);
            if (md != NULL) {
               /* finalize the operation */
               err = CPMLibHashFinal(&hashInfo, NULL, 0, md,
                  &size);
               if (err) {
                  DbgPrintf("SSFD: Error on CPMLibHash
                     0x%x\n", err);
               } else {
                  DbgPrintf("SSFD: CPMLibHashFinal - 0x%x
                     bytes returned\n", size);
               }
            }
         }
      }
   }
   /* release the context */
   CPMLibReleaseHashInfo(&hashInfo);
}
```

Certain applications may require the hashing context to be saved off and returned to at a later time. The `APHashInfoType` structures may be exported and imported in much the same way as keys are imported and exported.

## Encryption and Decryption Functions

The CPM supports encryption and decryption in much the same was as hashing is supported. That is encryption and decryption have two modes of operation. An application can either encrypt or decrypt a message as a whole, or in parts. Providers are not required to support both modes.

Encryption algorithms either work on data a byte at a time or in blocks of bytes (usually blocks of 8 bytes) at a time. The former is called stream encryption while the latter is called, appropriately enough, block encryption. Typically with block encryption, some padding is added to the data to make the data an integral number of blocks. Providers are not required to support padding. If the provider does not support padding the application must pad data for a block encryption algorithm or an error occurs.

As with the hashing operation, no context is required for the operation if you perform the encryption in a single step. This is illustrated in Listing 1.10.

#### Listing 1.10 A single-pass encryption operation

```
status_t err;
uint8_t key[] = {0x7C, 0xA1, 0x10, 0x45, 0x4A, 0x1A, 0x6E,
   0x57};
uint8_t plain[] = {0x01, 0xA1, 0xD6, 0xD0, 0x39, 0x77, 0x67,
   0x42};
uint8_t *output;
uint32_t index, size;
APKeyInfoType keyInfo;

MemSet(&keyInfo, sizeof(APKeyInfoType), 0);
keyInfo.type = apSymmetricTypeDES;
err = CPMLibImportKeyInfo(IMPORT_EXPORT_TYPE_RAW, key, 8,
   &keyInfo);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibImportKeyInfo 0x%x\n",
      err);
} else {
   DbgPrintf("SSFD: CPMLibImportKeyInfo - key of length 0x%x
      returned\n", keyInfo.length);
   size = 0;
   err = CPMLibEncrypt(&keyInfo, NULL, plain, 8, NULL,
      &size);
   if (err) {
      DbgPrintf("SSFD: Error on CPMLibEncrypt 0x%x with size
         set to 0x%x\n", err, size);
   } else {
      DbgPrintf("SSFD: CPMLibEncrypt - cipher data of length
         0x%x returned\n", size);
   }

   if (err = cpmErrBufTooSmall) {
      output = MemPtrNew(size);
      if (output != NULL) {
         err = CPMLibEncrypt(&keyInfo, NULL, plain, 8,
            output, &size);
         if (err) {
```

```
            DbgPrintf("SSFD: Error on CPMLibEncrypt 0x%x\n",
               err);
         } else {
            DbgPrintf("SSFD: CPMLibEncrypt - cipher data of
               length 0x%x returned\n", size);
         }
         MemPtrFree(output);
      }
   }
   CPMLibReleaseKeyInfo(&keyInfo);
}
```

With the multi-part encryption, the application must pass the context from the initial operation through to the final operation. The multi-part encryption is much the same as the multi-part hashing. The difference is that the provider does not maintain the state of the encrypted data during an update. The application must supply an output buffer for each update. The final operation will handle the last input data and any padding that is required to make a full encryption block of data. This is illustrated in Listing 1.11.

#### Listing 1.11 A multi-part encryption operation

```
status_t err;
uint32_t size;
uint8_t key[] = {0x7C, 0xA1, 0x10, 0x45, 0x4A, 0x1A, 0x6E,
   0x57};
uint8_t plain1[] = {0x01, 0xA1, 0xD6, 0xD0, 0x39, 0x77, 0x67,
   0x42};
uint8_t plain2[] = {'f', 'o', 'o', 'b', 'a', 'z', 'a', 'r'};
uint8_t *output;
APKeyInfoType keyInfo;
APCipherInfoType cipherInfo;

MemSet(&keyInfo, sizeof(APKeyInfoType), 0);
keyInfo.type = apSymmetricTypeDES;
err = CPMLibImportKeyInfo(IMPORT_EXPORT_TYPE_RAW, key, 8,
   &keyInfo);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibImportKeyInfo 0x%x\n",
      err);
} else {
   DbgPrintf("SSFD: CPMLibImportKeyInfo - key of length 0x%x
      returned\n", keyInfo.length);
   MemSet(&cipherInfo, sizeof(APCipherInfoType), 0);
```

```
/* initialize the context */
err = CPMLibEncryptInit(&keyInfo, &cipherInfo);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibEncryptInit 0x%x\n",
      err);
} else {
   /* update the operation; can do this any number of
      times */
   size = 0;
   err = CPMLibEncryptUpdate(&keyInfo, &cipherInfo,
      plain1, sizeof(plain1), NULL, size);
   if (err)
      DbgPrintf("SSFD: Error on CPMLibEncryptUpdate
         0x%x\n", err);

   if (err == cpmErrBufTooSmall) {
      output = MemPtrNew(size);
      if (output != NULL) {
         err = CPMLibEncryptUpdate(&keyInfo, &cipherInfo,
            plain1, sizeof(plain1), output, size);
         if (err) {
            DbgPrintf("SSFD: Error on CPMLibEncryptUpdate
               0x%x\n", err);
         } else {
            /* do something with output */
            err = CPMLibEncryptFinal(&keyInfo,
               &cipherInfo, plain2, sizeof(plain2),
               output, &size);
            if (err)
               DbgPrintf("SSFD: Error on
                  CPMLibEncryptFinal 0x%x\n", err);

            if (err == cpmErrBufTooSmall) {
               DbgPrintf("SSFD: cpmErrBufTooSmall with
                  CPMLibEncryptFinal returning %d\n",
                     size);
               output = MemPtrRealloc(output, size);
               if (output != NULL) {
                  err = CPMLibEncryptFinal(&keyInfo,
                     &cipherInfo, plain2, sizeof(plain2),
                     output, &size);
                  if (err) {
                     DbgPrintf("SSFD: Error on
                        CPMLibEncryptFinal 0x%x\n", err);
                  } else {
                        /* do something with output */
                     DbgPrintf("SSFD: CPMLibEncryptFinal -
                        0x%x bytes returned\n", size);
```

```
                        }
                     }
                  }
               }
               MemPtrFree(output);
            }
         }
         CPMLibReleaseCipherInfo(&cipherInfo);
      }
      CPMLibReleaseKeyInfo(&keyInfo);
   }
```

Decryption is almost exactly the same as encryption. All in one and multi-part decryption is supported if the provider supports the two modes. Padding is required if the provider does not perform padding and the application must ensure that the contexts are released correctly.

As with encryption, the cipher context can be safely ignored if the operation is performed in a single step, as shown in Listing 1.12.

### Listing 1.12 A single-pass decryption operation

```
status_t err;
uint8_t key[] = {0x7C, 0xA1, 0x10, 0x45, 0x4A, 0x1A, 0x6E,
   0x57};
uint8_t cipher[] = (0x69, 0x0F, 0x5B, 0x0D, 0x9A, 0x26, 0x93,
   0x9B};
uint8_t *output;
uint32_t index, size;
APKeyInfoType keyInfo;

MemSet(&keyInfo, sizeof(APKeyInfoType), 0);
keyInfo.type = apSymmetricTypeDES;
err = CPMLibImportKeyInfo(IMPORT_EXPORT_TYPE_RAW, key, 8,
   &keyInfo);
if (err) {
   DbgPrintf("SSFD: Error on CPMLibImportKeyInfo 0x%x\n",
      err);
} else {
   DbgPrintf("SSFD: CPMLibImportKeyInfo - key of length 0x%x
      returned\n", keyInfo.length);
   size = 0;
   err = CPMLibDecrypt(&keyInfo, NULL, cipher, 8, NULL,
      &size);
   if (err) {
```

```
      DbgPrintf("SSFD: Error on CPMLibDecrypt 0x%x with size
         set to 0x%x\n", err, size);
   } else {
      DbgPrintf("SSFD: CPMLibDecrypt - deciphered data of
         length 0x%x returned\n", size);
   }

   if (err == cpmErrBufTooSmall) {
      output = MemPtrNew(size);
      if (output != NULL) {
         err = CPMLibDecrypt(&keyInfo, NULL, cipher, 8,
            output, &size);
         if (err) {
            DbgPrintf("SSFD: Error on CPMLibDecrypt 0x%x\n",
               err);
         } else {
            DbgPrintf("SSFD: CPMLibDecrypt - deciphered data
               of length 0x%x returned\n", size);
         }
         MemPtrFree(output);
      }
   }
   CPMLibReleaseKeyInfo(&keyInfo);
}
```

# Authentication Manager

The Authentication Manager (AM) is an abstraction layer between applications and authentication methods. The framework provided by the AM allows modules (plug-ins) to be written that implement specific authentication scenarios. Users of the AM deal with generic interfaces and opaque objects that define an authentication context.

The Authentication Manager is the authority that can answer the question "Are you X?" reliably, by utilizing some method of identity verification such as a password. The question "Are you X?" may be asked about a user or an application.

The services provided by the AM handle the following tasks: credential (Token) management (creation, deletion, modification, and storage), authentication against stored credentials (querying user for system password), and a framework for run-time extensibility via plug-ins.

The Palm OS Cobalt implementation of the AM includes three authentication models:

- Password based authentication (as in OS5)
- Signed code (PKI) based authentication
- Code fingerprint (hashed code) based authentication

## Authentication Tokens

A token is a reference to an authentication requirement. The structure that represents a token contains credentials. Tokens can either be system tokens or non-system tokens. The only difference is in how the AM behaves when it destroys a token. When a system token is destroyed the AM takes all of the same actions as when destroying a non-system token, except that the named entry for the token is not removed from the AM's list of tokens. This is due to the fact that system tokens should always exist: they are "well known" tokens, such as the user token (password), or the admin token (password). Tokens can be marked as system tokens at the time they are created.

Every token has a unique system ID.

### Token Types

`AmTokenEnum` is an enumeration of the different types of tokens that can be requested from the system. These are the most common type of tokens that the device will deal with. The custom type (`AmTokenCustom`) allows the plug-in to announce a custom type of token that it will service. If an application requests a custom token, the Authentication Manager examines all plug-ins and finds all that match that custom type. Out of all the matches the best fit is picked to create the token.

```
typedef enum  {
   AmTokenUnknown = 0,
   AmTokenCustom,
   AmTokenPassword,
   AmTokenSignedCode,
   AmTokenCodeFingerprint
} AmTokenEnum
```

To authenticate a token, the AM invokes a plug-in that implements a specific authentication method.

### Token Strength

Associated with each token is the concept of "strength." Tokens can be either strong or weak; weak tokens are authentication tokens that can be easily guessed or broken, such as dictionary words for passwords, or weak cryptography keys. Within the token structure is the minimum level of strength that the plug-in supports for token creation. The following levels are defined:

**AmTokenStrengthLow:** The lowest level. There are no requirements for token creation.

**AmTokenStrengthMedium:** Some measures are taken to reject weak tokens.

**AmTokenStrengthHigh:** The generated token should be guaranteed to not be a weak token.

## Token Management Functions

The Authentication Manager APIs include functions to create, destroy, modify, and authenticate tokens:

- `AmCreateToken()`
- `AmDestroyToken()`
- `AmModifyToken()`
- `AmAuthenticateToken()`

To get information about a token, you use one of these functions:

- `AmGetTokenBySystemId()`
- `AmGetTokenExtendedInfo()`
- `AmGetTokenInfo()`

Finally, when manipulating the plug-ins themselves you work with these Authentication Manager functions:

- `AmGetPluginInfo()`
- `AmGetPluginReferences()`
- `AmRegisterPlugin()`

- AmRemovePlugin()

The Authentication Manager also supports the “legacy” APIs from earlier versions of Palm OS. These are the functions declared in `Password.h`. (`PwdExists()`, `PwdRemove()`, `PwdSet()`, and `PwdVerify()`). These functions all act on the user token. and only work if the user token is of type `AmTokenPassword`.

The Authorization Manager functions are easy to use. For instance, the code excerpt in Listing 1.13 shows how to authenticate the user token.

#### Listing 1.13 Authenticating the user token

```
AmTokenType token;
status_t err;

AmGetTokenBySystemId(&token, SysUserToken);
err = AmAuthenticateToken(token, NULL, AmAuthenticationOther,
   NULL, NULL);
if (err == errNone){
   // Authentication succeeded. Do something here.
} else {
   // Authentication failed.
}
```

`AmAuthenticateToken()` can take hints about what type of authentication is being performed (database access, device unlock, and so on). In the above example it is `AmAuthenticationOther`. This function can also take and optional title and description strings. These can be used by the AM plug-in to clarify to the user just why they are being prompted to enter a password (or provide a thumbprint, or whatever).

## Using the Authentication Manager

The Authentication Manager can be used either to authenticate a user or to authenticate code.

### User Authentication

User authentication requires that the user knows or has on his possession a secret that identifies him to the system. This secret comes in the form of a PIN, a password, biometrics, and so forth.

The Authentication Manager collects the credentials being presented by the user and compares them to the stored credentials, thereby authenticating the user.

## Code Authentication

There are two major scenarios when it comes to code authentication: signed code and unsigned code.

Signed code is usually a third-party application or a system patch. It was signed with a certificate, which was assigned by a certificate authority. The AM can verify the signature of the code and authenticate the identity of the certificate that was used when the code was signed. This is how the system protects patchable or replaceable objects: by requiring that the application patching or replacing an object be signed by a well-known certificate.

Code that is not signed is treated differently. It is expensive to acquire certificates from a certificate authority, and most shareware developers will not go through the trouble, yet the system is still able to protect data from access by any other application. In order to provide a non-interactive authentication method, the system creates a token that uniquely identifies an application when that application is installed on the device. An application may use this token to protect objects, and the AM can then verify the identity of the application by re-calculating the identity of the application and matching it against the identity that was calculated at install time.

### *Signature Verification Library*

The Signature Verification Library does the bulk of the work for the following code authentication tasks:

- Interpreting the sign resource in an application's resource database.
- Extracting the X.509 certificate block from the sign resource.
- Verifying the validity of a digital signature in a PRC file.

This library enables any application on the device to verify a signed PRC file. The Authentication Manager also uses this shared library to authenticate signed code. The AM's task is to authenticate currently running applications. Any other type of authentication that needs to be done can be accomplished by using this shared library.

The Signature Verification Library is covered in detail under "Signature Verification Library" on page 46.

## Creating an Authentication Manager Plug-In

An Authentication Manager plug-in is a shared library of type `'ampl'` that extends the authentication services provided by the AM. Each plug-in implements one authentication model, and is responsible for implementing any UI associated with that model. Authentication Manager plug-ins are loaded by the security process when the new plug-in is registered with the AM.

**IMPORTANT:** Plug-ins execute in privileged space and have access to the whole system. Bugs in these modules can create security holes that can potentially expose all of the data on the device.

### Working With Tokens

Plug-ins publish information about the type of tokens they can create when registering with the AM. The AM then uses that information to find the most appropriate plug-in to use when an application creates a token. An application that wishes to use specific features supported by a specific plug-in is able to request that plug-in when creating a new token.

The plug-in that creates the token defines the structure of the token's data. Since the data for the token must be tamper proof, it is stored in system space (owned by the Authentication Manager), and the application is only given a reference: an `AmTokenType`. When the AM is asked to authenticate with a token, the plug-in that created the token is asked to collect a new token and compare it with the stored token. If the two tokens match, the authentication passes.

Plug-ins may define their own internal data structures to use for storing information about the token. The memory may be controlled by the AM or by the plug-in. If the memory allocation and deallocation is to be done by the AM, the plug-in must specify how much memory to allocate per token in the registration structure of the plug-in.

The custom token type (`AmTokenCustom`) allows the plug-in to announce a custom type of token that it will service. If your plug-in creates custom tokens, be sure that it fills in the `identifier` field of the `AmTokenPropertiesType` structure with the identifier for the plug-in. Otherwise, the `identifier` field can be set to 0.

Each token has a public info block that can be shared with applications through `AmGetTokenInfo()`. This info, defined by the `AmTokenInfoType` structure, is set by the plug-in when the token is created. Plug-ins may use their own discretion on how much information to divulge. Note that not all fields are applicable to all types of tokens.

Token attributes (defined by the `AmTokenAttributesType` structure) are flags that specify information about the token. The plug-in sets these flags and uses them later to allow or reject certain actions.

**destroy:** the token may be destroyed.

**modify:** the token may be modified.

**interactive:** the token is user interactive. That is, it is a password, PIN, or the like.

**empty:** the token is empty.

**system:** the token is a system token.

The remaining fields of the `AmTokenInfoType` structure are set or filled in by the Authentication Manager, except for the “friendly name” which should be supplied by the plug-in.

The function `AmAuthenticateToken()` can take hints about what type of authentication is being performed (database access, device unlock, and so on) in the form of an `AmAuthenticationEnum` value. This function can also take optional title and description strings. These can be used by the AM plug-in to clarify to the user just why they are being prompted to enter a password (or provide a thumbprint, or whatever).

### Plug-In Entry Points

A plug-in is a shared library that has a main entry point (see “The Main Entry Point” on page 33) which receives launch codes. During the processing of the `sysAppLaunchCmdNormalLaunch` launch code it fills in an initialization data structure which lets the AM know the address of its entry points. These entry points constitute a protocol for capturing, replacing, verifying, destroying, importing, and exporting tokens. The AM invokes these entry points in a defined sequence when carrying out a task such as creating a token. Some of the entry points have a context argument (an `AmApplicationCtxType`) that lets the plug-in know the context in which it is executing. This allows the plug-in to implement the correct UI associated with a given action under different contexts.

An AM plug-in implements entry points for the following actions:

**Open and Close:** Called at load and unload time, respectively.

**Capture:** Called by the AM during the capture of token information.

**Match:** Called by the AM to compare two tokens.

**Destroy Notify:** Called by the AM when a token is being destroyed.

**Get Extended Info:** Called to get extended information about a plug-in.

**Import and Export:** Called to import or export a plug-in.

**Get Derived Data:** Called to get derived data from a token.

**Admin:** Called by the AM to administer a plug-in.

Except for the `PluginOpen` and `PluginClose` functions, the plug-in exports the above listed functions in an `AmPluginFunctionsType` structure.

### Open and Close

`PluginOpen` is generally called upon receipt of a `sysAppLaunchCmdNormalLaunch` launch code, while `PluginClose` is usually called upon receipt of a `sysLaunchCmdFinalize` launch code.

`PluginOpen` and `PluginClose` are not directly invoked by the AM. Instead, the shared library support is used to let the plug-in know about these actions. That is, the main entry point for the shared library receives the following launch codes:

**sysAppLaunchCmdNormalLaunch:** Instructs the plug-in to initialize itself. This is the "PluginOpen" functionality. Along with this launch code the plug-in is passed a pointer to an AmPluginPrivType structure (in the command block pointer argument). The plug-in should initialize the `ftn` field with pointers to those functions that the plug-in exports. It should also set up the `info` field with pertinent information about the plug-in (friendly name, vendor, version, and so on) and the token properties, and set the `tokenDataLength` and `tokenExtendedInfoLength` fields as appropriate. Finally, the plug-in should open any needed libraries (such as the CPM) and then return `errNone`.

**sysLaunchCmdFinalize:** Instructs the plug-in to close. It should perform any necessary cleanup and close any libraries opened by the `PluginOpen` function.

### *Capture*

The Capture function is called whenever the AM needs your plug-in to create a new token, or to verify or replace an existing token created by the plug-in. Capture is invoked with one of four defined modes (AmCallMode): **Enrollment**, **Verification**, **Replacement Start**, and **Replacement End**. Enrollment is used when a new token is being created. Verification is used when capturing tokens for authentication. Replacement is a two-phase protocol: first the AM may need to authenticate access to modify a token (start), and then it captures a new token to replace the old (end) with.

During the processing of this function the plug-in may implement UI to gather tokens. The mode passed in to this entry point can help determine the exact UI to present.

Your Capture function should use the following prototype:

```
status_t (*pluginCaptureFtn)(AmCallMode, AmApplicationCtxType *,
AmTokenPrivType *, AmAuthenticationEnum, char *, char *)
```

See the description of AmPluginFunctionsType for a description of this function's parameters.

Your Capture function should return `errNone` if the operation completed successfully. Otherwise, return an appropriate Authentication Manager (or other) error code.

#### *Match*

When the AM needs to verify a token it invokes the associated plug-in's match entry point, passing in two token structures for comparision. Any success or failure UI is implemented by the plug-in.

Your Match function should use the following prototype:

```
status_t (*pluginMatchFtn)(AmApplicationCtxType *, AmTokenPrivType *,
AmTokenPrivType *)
```

See the description of AmPluginFunctionsType for a description of this function's parameters.

Your Match function should return `errNone` if the operation completed successfully. Otherwise, return an appropriate error code such as `amErrAuthenticationFailed`.

#### *Destroy Notify*

The destroy notification is sent to the plug-in that created the token when the token is destroyed.  Destroying a token is an action that may be taken if the user has lost the ability to authenticate against that token (as in the case of a lost password). When creating a token the plug-in sets a flag that allows or disallows its destruction.

Your Destroy Notify function, if your plug-in implements one, should use the following prototype:

```
status_t (*pluginDestroyNotifyFtn)(AmTokenPrivType *)
```

See the description of AmPluginFunctionsType for a description of this function's parameters.

Your Destroy Notify function should return `errNone` if the operation completed successfully. Otherwise, return an appropriate Authentication Manager (or other) error code.

#### *Get Extended Info*

This function is used to answer the query for extended info by an application. A plug-in is not required to support this entry point, though it can be useful for certain types of tokens. The Palm OS PKI plug-in returns the certificate ID of the token (in an AmPluginSignedCodeExtInfoType structure). The Palm OS

Code Fingerprint plug-in returns the type, creator, and name of the database that was fingerprinted (in an `AmPluginCodePrintExtInfoType` structure). The Palm OS password plug-in, however, doesn't implement this function.

Your Get Extended Info function, if your plug-in implements one, should use the following prototype:

```
status_t (*pluginGetTokenExtendedInfoFtn)(AmTokenPrivType *, uint8_t *,
uint32_t)
```

See the description of `AmPluginFunctionsType` for a description of this function's parameters.

Your Get Extended Info function should return `errNone` if the operation completed successfully. Otherwise, return an appropriate Authentication Manager (or other) error code.

#### *Import and Export*

If the Authentication Manager does all of the memory management for a particular plug-in, then the export and import of that plug-in's tokens is mostly taken care of by the AM. The AM will make sure that the buffer it has allocated for internal data for each token is exported and imported correctly. Plug-ins only need to worry about exporting or importing data that they have allocated themselves.

The Import and Export entry points are for copying internal data about a token for import or export. In your export function, memory that is associated with a token *and is managed by the plug-in* should be copied to the provided buffer and returned to the AM. The import function should do the opposite: copy the contents of a passed-in buffer into the token memory managed by the plug-in. Note that import and export functions are needed only for tokens that have associated data not managed by the AM itself. Because the Authentication Managert knows how to import and export the buffer that is allocated by the AM for the token data, simple plug-ins such as the password plug-in don't need to implement import and export functions.

Your Import and Export functions, if your plug-in implements them, should use the following prototypes:

```
status_t (*pluginImportTokenFtn)(AmTokenPrivType *, uint8_t *, uint32_t)
```

```
status_t (*pluginExportTokenFtn)(AmTokenPrivType *, uint8_t *, uint32_t *)
```

See the description of `AmPluginFunctionsType` for a description of the function parameters.

Your Import and Export functions should return `errNone` if the operation completed successfully. Otherwise, return an appropriate Authentication Manager (or other) error code.

#### *Get Derived Data*

This function is used solely by the operating system to get seed data for a cryptographic key derived from an authentication token (such as password derived keys). Currently the only user of this feature is the Data Manager; it uses this feature to generate password-derived keys for the backup function.

Your Get Derived Data function, if your plug-in implements one, should use the following prototype:

```
status_t (*pluginGetDerivedData)(AmTokenPrivType *, uint8_t *, uint32_t *)
```

See the description of `AmPluginFunctionsType` for a description of the function parameters.

Your Get Derived Data function should return `errNone` if the operation completed successfully. Otherwise, return an appropriate Authentication Manager (or other) error code.

#### *Admin*

This function is the admin entry point for the plug-in. Some plug-ins may have settings that can be changed (a biometric plug-in, for instance, might allow the user to tweak the settings it uses to match tokens); accordingly, the plug-in can implement an admin UI in its implementation of this function.

Your Admin function, if your plug-in implements one, should use the following prototype:

```
status_t (*pluginAdminFtn)(AmPluginType *)
```

See the description of `AmPluginFunctionsType` for a description of the function parameters.

Your Admin function should return `errNone` if the operation completed successfully. Otherwise, return an appropriate Authentication Manager (or other) error code.

### The Main Entry Point

The main entry point must have a definition as follows:

**Prototype**

```
uint32_t AmPluginMain (uint16_t cmd,
    MemPtr cmdPBP, uint16_t launchFlags)
```

**Parameters**

`cmd`
: The launch code. Of particular interest are `sysAppLaunchCmdNormalLaunch` and `sysLaunchCmdFinalize`.

`cmdPBP`
: When the launch code is `sysAppLaunchCmdNormalLaunch`, this parameter points to an `AmPluginPrivType` structure. The plug-in must fill in this structure before returning.

`launchFlags`
: Not used.

**Returns**

Return `errNone` to successfully register the plug-in. Otherwise, return one of the error codes declared in `Am.h`.

The following is a sample implementation of a plug-in entry point and initialization function:

**Listing 1.14 Sample plug-in entry point function**

```
uint32_t AmPkiPluginMain(uint16_t cmd, MemPtr cmdPBP,
   uint16_t launchFlags){
   switch(cmd) {
      case sysLaunchCmdInitialize:
      {
         // Do custom initialization here
         break;
      }

      case sysAppLaunchCmdNormalLaunch:
      {
         // For a PkiPlugin -- do the open
         AmPluginPrivType *pPlugin =
            (AmPluginPrivType *)cmdPBP;
```

```
            return (AmPkiPluginOpen(pPlugin));
         }

         case sysLaunchCmdFinalize:
         {
            // Do custom de-initialization here
            break;
         }
         default:
            break;
      }

      return 0;
   }

   status_t AmPwPluginOpen(AmPluginPrivType *pPlugin){
       uint16_t  numProviders= 0;
       status_t     err;

       // Setup the function array
       pPlugin->ftn.pluginCaptureFtn = AmPwPluginCapture;
       pPlugin->ftn.pluginMatchFtn = AmPwPluginMatch;
       pPlugin->ftn.pluginDestroyNotifyFtn =
         AmPwPluginDestroyNotify;
       pPlugin->ftn.pluginAdminFtn = AmPwPluginAdmin;
       pPlugin->ftn.pluginGetDerivedData =
         AmPwPluginGetDerivedData;

       // Setup the info piece
       strcpy(pPlugin->info.friendlyName, PwPluginFriendlyName);
       strcpy(pPlugin->info.vendor, PwPluginVendor);
       pPlugin->info.version = PwPluginVersion;

       pPlugin->info.tokenProperties.type = AmTokenPassword;
       pPlugin->info.tokenProperties.strength =
         AmTokenStrengthLow;
       pPlugin->info.tokenProperties.identifier =
         PwPluginCreator;

       // Setup the token length
       pPlugin->tokenDataLength = sizeof(PwPluginTokenType);

       return (err);
   }
```

### Sample Plug-In Implementations

The following sections provide some details about how the sample implementations of the three standard plug-ins provided to Palm OS licencees are implemented.

#### *Password Plug-In*

The password plug-in doesn't store plain-text passwords. Instead, it stores hashes of the passwords (SHA1 or MD5). Comparisons are done using the hashes.

The Password plug-in implements the AM plug-in interface in the following manner:

**Open:** Initializes the entry point function array, sets the plug-in properties (`friendlyName`, `vendor`, and `version`), sets the token properties (`type` = `AmTokenPassword`, `strength` = `AmTokenStrengthLow`, `identifier`), sets the token data length to `sizeof(PwPluginTokenType)`, and opens the CPM.

**Close:** Closes the CPM.

**Capture:** Supports each mode as described under "Capture" on page 29.

**Match:** Compares the supplied password hashes.

**Destroy Notify:** Frees the memory allocated to the password hint.

**Get Token Extended Info:** Not implemented.

**Import/Export:** Not implemented.

**Get Derived Data:** Copies the password hash into the supplied buffer.

**Admin:** Does nothing.

#### *Certificate Plug-in*

The certificate plug-in implements a code-signing authentication model. This plug-in can verify whether a specific application has been signed by a specific certificate. Tokens associated with this plug-in authenticate when the executing application has a signature from this certificate (the certificate ID is stored in the token).

The Certificate plug-in implements the AM plug-in interface in the following manner:

**Open:** Initializes the entry point function array, sets the plug-in properties (`friendlyName` , `vendor`, and `version`), sets the token properties (`type` = `AmTokenSignedCode`, `strength` = `AmTokenStrengthHigh`, `identifier`), sets the token data length to `sizeof(PkiPluginTokenType)`, sets the `tokenExtendedInfoLength` to `sizeof(AmPluginSignedCodeExtInfoType)`, and opens the CPM.

**Close:** Closes the CPM.

**Capture:** Implements `AmEnrollment` and `AmVerification`. `AmReplacementStart` and `AmReplacementEnd` simply return `amErrActionnotSupported`.

**Match:** Compares the supplied tokens.

**Destroy Notify:** Not implemented. Tokens created by this plug-in cannot be destroyed.

**Get Token Extended Info:** Copies the certificate ID to the supplied buffer.

**Import/Export:** Not implemented.

**Get Derived Data:** Not implemented.

**Admin:** Not implemented.

### *Application Fingerprint Plug-in*

A application fingerprint is a cryptographic hash of an application's resources. This plug-in implements an authentication model where current application must match a previously-stored cryptographic hash. This allows the system to set up access control where a specific application is granted access.

The Application Fingerprint plug-in implements the AM plug-in interface in the following manner:

**Open:** Initializes the entry point function array, sets the plug-in properties (`friendlyName` , `vendor`, and `version`), sets the token properties (`type` = `AmTokenCodeFingerprint`, `strength` = `AmTokenStrengthLow`, `identifier`), sets the token data length to `sizeof(CodePrintPluginTokenType)`, sets the `tokenExtendedInfoLength` to `sizeof(AmPluginCodePrintExtInfoType)`, and opens the CPM.

**Close:** Closes the CPM.

**Capture:** Implements `AmEnrollment` and `AmVerification`. `AmReplacementStart` and `AmReplacementEnd` simply return `amErrActionnotSupported`.

**Match:** Compares the supplied tokens.

**Destroy Notify:** Not implemented. Tokens created by this plug-in cannot be destroyed.

**Get Token Extended Info:** Copies the database name to the supplied buffer.

**Import/Export:** Not implemented.

**Get Derived Data:** Not implemented.

**Admin:** Not implemented.

## Manipulating Authentication Manager Plug-Ins

The header file `Am.h` defines functions that allow you to install and remove Authentication Manager plug-ins, as well as get information about an installed plug-in and find all references to an installed plug-in.

### Installing a Plug-in

Call `AmRegisterPlugin()` to install, or **register**, a plug-in. This function loads and opens the plug-in shared library. If you attempt to register a plug-in that is already registered you will be notified of that fact unless you set the *`force`* parameter to `true`, in which case the plug-in is closed and unloaded, then loaded and reopened. In this case the reference to the plug-in doesn't change; it is re-used. This means that all tokens still have a valid reference to their creator.

As explained in "Plug-in Security" on page 38, the plug-in system space is protected by a token which must be authenticated against prior to installation; this keeps rogue plug-ins from being allowed onto the device that could circumvent all authentication security.

### Removing a Plug-in

You remove an installed Authentication Manager plug-in by calling `AmRemovePlugin()`. Note that you can only remove a plug-in if there are no tokens on the device that have been created by that plug-in.

### Other Plug-in Manipulation Operations

Call `AmGetPluginInfo()` to get the public info block for a registered plug-in. The returned `AmPluginInfoType` data structure contains information about the plug-in, such as vendor name, friendly name, and information about the type of tokens that the plug-in can create.

To get a list of all currently registered Authentication Manager plug-ins, call `AmGetPluginReferences()`. You must allocate the array into which the list of references (each is an `AmPluginType`) is written; call `AmGetPluginReferences()` with a `NULL` pointer for the array to have returned to you the number of elements that would be written to the array.

### Plug-in Security

The storage area used to hold plug-in information is protected by a token. Authentication against this token is necessary before a plug-in can be installed or removed. So, for instance, if the plug-in storage space is protected by a password token, the user would need to enter a password before a plug-in could be installed or removed.

The device manufacturer can create a policy that controls how Authentication Manager plug-ins are installed. If no policies are set, the device behaves as follows when installing a plug-in:

- If the user’s current security level is set to None, the plug-in is installed.
- If the user’s current security level is set to Medium, the user is prompted to choose whether or not the plug-in should be installed.
- If the user’s current security level is set to High, the plug-in is not installed.

# Authorization Manager

The Authorization Manager (AZM) manages access control lists that are based on authentication tokens. These control lists are called rule sets. Figure 1.2 illustrates the basic rule-set syntax.

**Figure 1.2 Rule-set syntax**

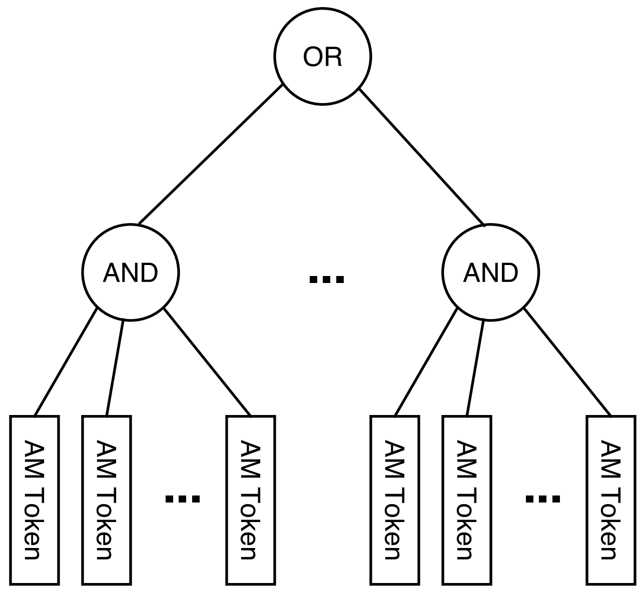
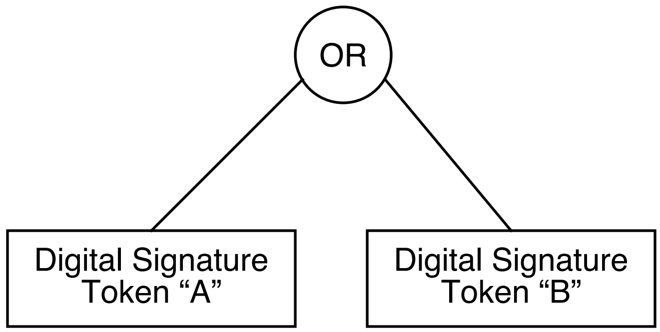



Figure 1.3 illustrates a simple rule set. When this rule set is evaluated, it will be satisfied if the currently running application's signature can be verified with either certificate "A" or certificate "B".

**Figure 1.3 A simple rule-set**




The set of APIs exposed by the Authorization Manager to third-party developers is quite small, consisting only of:

- `AzmAddRule()`, which lets you add an access rule to an existing rule-set container for a specific action.
- `AzmNonInteractiveAuthorize()`, which authorizes an action given a rule-set reference.
- `AzmGetSyncBypass()` and `AzmSetSyncBypass()`, which let you get and set the **sync bypass flag** in an existing rule-set container for a specific action. Sync-bypass must enabled for a specific action in order for an authenticated sync agent to be able to complete that action successfully.

Note that as a third-party developer you cannot create or destroy rule sets.

Applications use `AzmAddRule()` to add rules to a rule set. Listing 1.15 presents one common operation: authenticating the user to perform a particular action.

**Listing 1.15 Adding a rule to a rule set**

```
AmTokenType usertoken;
AzmRulesetType ruleSet;

AmGetTokenBySystemId(&usertoken, SysUserToken);

ruleSet = CreateProtectedResource();
err = AzmAddRule(ruleSet, action, "%t", usertoken);
```

The contents of the action parameter depend upon what is being protected. Many applications will use access rules to control access to schema databases. See "Schema Database Access Rule Action Types" on page 301 of *Exploring Palm OS: Memory, Databases, Files* for the access rules that can be applied to schema databases.

The above shows the addition of a very simple rule. More complex rules can be constructed using AND and OR logic operations. To construct a rule set that is valid if either of two tokens is authorized, use the logic operator OR when specifying the rule format string, as in "`%t OR %t`". An AND operation is even simpler, since you don't supply the word "AND." Simply specify the format like this: "`%t %t`".

# Certificate Manager

The Certificate Manager provides a secure server for the storing and parsing of DER-encoded X.509 digital certificates. It exposes functions that allow you to import, export, parse, and verify those certificates.

You can use the Certificate Manager in either of two different ways: as a certificate verifier and parser, and as a certificate store. In the verifier/parser mode, the Certificate Manager takes data as input and parses it as a digital certificate. The user can then verify the certificate and access its internal fields. In certificate store mode, the Certificate Manager can securely store a tree of digital certificates (with multiple roots) and make the fields of those certificates available to users.

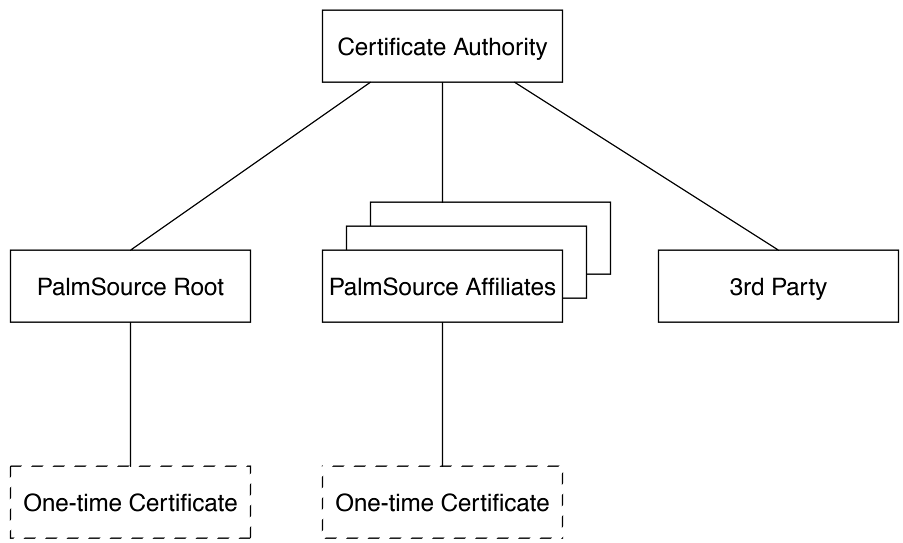
The Certificate Manager is a system server with a client-side library. To securely store certificates, the Certificate Manager makes use of the Data Manager's vault facilities. This allows the Certificate Manager to guarantee the integrity of any certificate added to its certificate store.

Note that very few applications use the Certificate Manager directly. As was shown in Figure 1.1 on page 4, both SSL and the Signature Verification Library make use of the Certificate Manager on the application's behalf. The Certificate Manager only exposes a fairly low-level set of APIs.

## Certificate Store Operations

The Certificate Manager can securely store a tree of digital certificates (with multiple roots). Figure 1.4 shows the basic certificate hierarchy.

**Figure 1.4 Certificate Hierarchy**




At boot time the certificate store is seeded with the list of root certificates that were stored in ROM by the device manufacturer. These ROM certificates are used to authenticate RAM certificates.

To get a certificate from the store, call `CertMgrFindCert()`. This function can be used in one of two modes: to find a particular certificate by ID or by subject RDN, or to iterate through all of the certificates in the certificate store. You control this function's operation through the use of the *`searchFlag`* parameter.

To add and remove certificates from the store, you use `CertMgrAddCert()` and `CertMgrRemoveCert()`, respectively. Note that you can only add a certificate if its authentication chain already resides in the certificate store, or if the certificate is self-signed. Also note that removing a certificate that is part of an authentication chain may prevent new certificates from being authenticated.

The code excerpt shown in Listing 1.16 shows how you can use `CertMgrAddCert()` to add a self-signed certificate to the certificate store.

**Listing 1.16 Adding a self-signed certificate**

```
while (true) {
   err = CertMgrAddCert(&certInfo, false, &verifyResult);
   if (err) {
      CertMgrReleaseCertInfo(&certInfo);
      goto exit;
   }

   if (verifyResult.failureCode == 0) {
      break;
   } else {
      if (verifyResult.failureCode ==
         CertMgrVerifyFailSelfSigned) {
         verifyResult.failureCode = 0;
         continue;
      }

      /* Another type of failure */
      break;
   }
}
```

## Certificate Verification and Parsing

Use `CertMgrImportCert()` to import a DER-encoded x509 certificate and get back a `CertMgrCertInfoType` structure. This structure represents a certificate object. You then verify this certificate's contents by calling `CertMgrVerifyCert()`. Once you have a verified certificate, use `CertMgrGetField()` to get fields out of the certificate. Most commonly, applications will want to get the key from the certificate.

Once you are done with a certificate, be sure to call `CertMgrReleaseCertInfo()` to release these resources that were allocated by the Certificate Manager during the call to `CertMgrFindCert()` or `CertMgrImportCert()`.

## Certificate Backup and Restore

All certificates in the certificate store are backed up and restored.

# Security Services

In Palm OS Cobalt the security services allow the user of the Palm Powered device to specify a level of "paranoia." This maps directly to the paradigm of private records being visible, masked, or hidden but in Palm OS Cobalt this security setting extends to more than just private records. The security services also control the automatic locking and unlocking of the device. Finally, they also allow licensees to specify basic restrictive policies for various managers and services on the device, and provide APIs that let third-party developers examine those policies.

## Current Security Setting

`SecSvcsGetDeviceSetting()` and `SecSvcsSetDeviceSetting()` allow you to get and set the device's current security setting. The security setting is an indication of how much the user wants to be bothered with security and how "paranoid" the user is. The manager or service reading this setting should follow these guidelines with regard to the security setting:

**SecSvcsDeviceSecurityNone:** The user does not want to be bothered at all and the device is totally open. Everything is "ok" by the user.

**SecSvcsDeviceSecurityMedium:** The user should be bothered with a Yes/No question about the pending operation with as much information about the operation as is reasonably possible (whether the code is signed or unsigned, which manager is performing the operation, details about the operation being performed, and so on). A "Yes" from the user indicates that the operation should proceed. "No" means that the operation should not be performed.

**SecSvcsDeviceSecurityHigh:** The user does not want to be bothered at all and the device is essentially closed. In general, no operations should be performed.

The following table shows how certain aspects of Palm OS Cobalt react, by default, to the various security settings:

| | None | Medium | High |
|---|---|---|---|
| **Trusted Desktop** | No UI | Ask user | Ask user |
| **Sync Clients** | No UI | Ask user | Ask user |
| **Token Caching** | Global | Application | Application |
| **AM Plug-in** | No UI; always allowed | Ask user | No UI; always denied |

## Lockout Settings

`SecSvcsGetDeviceLockout()` and `SecSvcsSetDeviceLockout()` get and set the device's current lockout scheme. These functions are intended to be used by security applications that control the locking and unlocking of the device.

Use `SecSvcsEncodeLockoutTime()` to encode the lockout parameters into a 32-bit value for use by `SecSvcsSetDeviceLockout()`. As you might expect, you use `SecSvcsDecodeLockoutTime()` to decode the lockout parameters from the 32-bit value returned from `SecSvcsGetDeviceLockout()`.

Low-level modules that control whether or not the device is locked can use `SecSvcsIsDeviceLocked()`. This function returns a boolean value: `true` if the device is locked, `false` if it is not.

## Security Policies

Palm OS Cobalt licensees can set security policies for the various managers in their Palm OS system. Specifically, the following can be gated by separate security policies:

- Processes
- FEPs and locale modules
- The BSP key
- IOS drivers

- Sync clients
- Authentication Manager plug-ins
- CPM providers

`SecSvcsGetDevicePolicies()` obtains the security policies defined for the device. It checks, in order, the ROM token area of the device, any ROM-based PDB files, and then any RAM-based PDB files. The PDB files and ROM tokens that this function checks are of a specific format. The format used by the policies is a non-terminated list of 20 byte IDs of certificates against which code must be checked for signed status. These IDs can be directly used with the Signature Verification Library using the `SignVerifySignatureByID()` function. The Signature Verification Library is responsible for checking that code is signed appropriately before it is used.

Application developers can build a secure application by including a set of security policies in the PRC of the application (using PRCCert and PRCSign; see the book *Working with Resource Tools*) and then signing the PRC digitally. Before the Program Loader launches the application, it will make sure the application's integrity has not been compromised. When such an application is running, the Program Loader also makes sure that any shared library loaded into that application's process meets the requirement of the security policies carried by the application's PRC.

# Signature Verification Library

The Signature Verification Library provides an interface through which applications can access signature and certificate resources (`'sign'` and `'cert'`, respectively) in a PRC. With the exposed APIs you can:

- Get the number of signatures or certificates
- Get a certificate or signature by index or certificate ID
- Get an overlay validation certificate list
- Get a shared library validation certificate list
- Validate a signature

Reference documentation for the APIs exposed by this library can be found in Chapter 15, "Signature Verification Library," on page 313.

## Signature Verification

Signature verification is perhaps the most common Signature Verification Library operation. Listing 1.17 contains a code excerpt that shows how you can verify a signature.

#### Listing 1.17 Validating a signature

```
DatabaseID dbID;
DmOpenRef dbP;
status_t err;

dbID = DmFindDatabase("Test_SignedCode", 'scta');
dbP = DmOpenDBNoOverlay(dbID, dmModeReadOnly);
err = SignVerifySignatureByIndex(dbP, 0);
if (err)
   DbgPrintf("Error in Signature\n");
else
   DbgPrintf("Signature validation succeeded");
DmCloseDatabase(dbP);
```

To verify a PRC's digital signature, its sign resource block must be interpreted. Each signature block in a sign resource contains a reference to its verifying certificate. This reference is the certificate's ID (the SHA1 digest of the public key).

The code verifying the signature can get the RSA verify key (the public key) from the Certificate Manager by referencing the certificate ID. If the certificate is not found in the certificate store, search for a matching certificate in the PRC file. If a certificate in the PRC file matches the ID, import it into the certificate store prior to using it for verification. The Certificate Manager verifies the integrity, validity, and suitability of the certificate during the process of importing it into the certificate store.

---

**NOTE:** It is possible to import expired certificates into the certificate store for purposes of verifying digital signatures. If a certificate has expired, the signature verification code is responsible for verifying that the PRC file was signed prior to the expiration date of the certificate.

---

The Authentication Manager, running in the System process, makes use of the signature verification shared library to provide signed code authentication.

# Signing Code

Applications are signed when code integrity is a concern. Depending on the device, some code—such as a patch—may need to be signed. Or, a secure database may be configured in such a way as to only allow access by a particular group of signed applications.

## What can be Signed

Signed code in Palm OS Cobalt is used to validate the authenticity of a program resource. There are several types of resources that could be signed in Palm OS Cobalt. Applications, system patches, shared libraries, system components, system drivers, and the like. All of these resources are packaged as PRC files and then loaded onto the device.

Unlike the desktop world where digital signatures are used to indicate the author of a piece of software, Palm OS uses digital signatures to represent endorsements. For example, an enterprise can use its own certificate to sign Palm OS applications that it has tested for usefulness and interoperability with other core enterprise applications. This makes it easy for employees to know they are getting good applications.

An application can have multiple endorsements. For example, an application created by a major software vendor could be signed by the vendor as well as by the enterprise. It is also possible for user groups to endorse software that they have reviewed favorably. These endorsements help the user decided whether to install a piece of downloaded software.

### Overlay and Shared Library Validation Certificate List

A signed application can define two lists of certificate IDs: an overlay list, and a shared library list. These lists authenticate the integrity of overlays and shared libraries. They are defined at the time that the first signature is applied to the application, and cannot be changed without invalidating that signature.

## Signing Algorithm

The algorithm for digital signatures in Palm OS Cobalt is the RSA/SHA1 signing algorithm. This means that Palm OS Cobalt uses RSA private keys to sign a SHA1 digest, and RSA public keys to verify the signature. To be compatible with the widest range of cryptographic hardware vendors, the padding format is PKCS #1 Block Type 1 from version 1.5 of the PKCS #1 specification.

The signing keys can be either self-issued or assigned by a Certificate Authority (CA). These are RSA keys with 1024 bits and can have either double-prime or triple-prime modulus. The exponent can be 3 or 216+1. Verifying a signature produced with exponent 3 is about three times faster than with the larger exponent.

## Signing Tools

There are two tools that are required to do code-signing of PRC's: PRCCert and PRCSign. You use **PRCCert** to create your own RSA key pairs and digital certificates. You may create self-signed certificates for testing, or certificates that are signed by other private keys. PRCCert creates RSA public/private key pairs at 1024-bit length in PEM format. PRCCert also generates a public certificate file in DER format.

The output files from PRCCert are used as input files to PRCSign.

**PRCSign** is a tool that you use to digitally sign your applications or to embed digital signature certificates in your applications. PRCSign creates a digital signature for a particular PRC using an asymmetric key cipher, storing the signature into the PRC as a resource of type `'sign'`. The signature can be verified as authentic by using your public key to decipher the signature resource. Each application has at most one `'sign'` resource with a resource ID of 1000.

PRCSign takes your private key and signs a SHA1 hash of all of the static (unchanging) resources in the PRC along the signature attributes. PRCSign then adds the resulting output as the `'sign'` resource to your application PRC file. PRCSign also takes a digital signature and adds it to the PRC as a `'cert'` resource in such a way that the Security Manager can retrieve it for application certification.

PRCSign supports keys in regular files, and smart cards.

The tools can handle any kind of X.509 certificate as long as the certificate's key usage constraints include the ability to sign data. Certificates that can only sign certificates, and certificates that can only be used for encryption, are not acceptable. A certificate issued for signing e-mail, however, is acceptable even though it is not marked as being able to sign code. This allows for the widest possible range of certificate-issuing systems or infrastructures to be used to sign Palm OS software. It is up to the Certificate Manager acting on behalf of the user or the administrator to further restrict the suitability of certificates. The tools do not enforce this.

For detailed instructions on using PRCCert and PRCSign, see *Palm OS Resource Tools Guide*.

## Signed Code and Shared Libraries

Palm OS Cobalt can guarantee the integrity of the shared libraries that are loaded by an application while it runs. In order to do this the application has associated with it a list of certificates (the "shared library certificate ID list") that authenticate the shared libraries that are loaded at runtime. The operating system then uses this list of certificates to authenticate any shared library that is loaded by the system for the application. If a shared library has a valid signature (one that can be verified by one of the certificates in the list), then it is loaded. Otherwise the load is cancelled, and the application stops running.

The feature in PRCSign that allows the setting of this list is the `–scert` parameter. Multiple `–scert` parameters can be supplied. Note that the list of shared library certificates must be set when the PRC is being signed; it cannot be modified after the first signature is applied. Previous signatures will be invalidated if the list is modified; the Application Manager requires that all signatures be valid when it checks the integrity of the certificate list. If any signature is determined to be invalid, the application is stopped.

### Shared Library Scenarios

Whether or not a given shared library is loaded depends on whether the application is signed, and whether the shared library's signature appears in the application's shared library certificate ID list.

#### *Unsigned PRC*

If the application isn't signed, the application is run and any needed shared libraries are loaded without any verification of signatures.

#### *Signed PRC, Empty Shared Library Certificate ID List*

If the application is signed but the shared library certificate ID list is empty, all signatures on the PRC must be valid, or the PRC execution is halted. Any needed shared libraries are loaded without signature verification.

#### *Signed PRC, Shared Library Certificate ID List has Entries*

If the application is signed and there are entries in the shared library certificate ID list, all signatures on the PRC must be valid. Any needed shared libraries must have a signature that can be validated by one of the certificates in the shared library certificate ID list.

## Signed Code and Overlays

Overlays for signed PRC's must be authenticated before they are applied. A signed PRC may contain a list of overlay certificate IDs. This list contains the IDs of certificates that may be used to authenticate overlays.

The list of overlay certificate IDs is included in the signing hash for each signature, so if a user or application changes this list all previous signatures are invalidated.

Prior to applying an overlay to a base PRC, the Locale Manager must verify that the overlay can be authenticated by one of the certificates in the "overlay certificate ID list" (from the `'sign'` resource).

### Overlay Scenarios

Whether or not a given overlay is applied depends on whether the application is signed, and whether the overlay's signature appears in the application's overlay certificate ID list.

#### *Unsigned PRC*

If the application isn't signed, the application is run and any overlays are applied without any verification of signatures.

Note that signed overlays may be applied to unsigned PRC files because the operating system doesn't check the signature of the overlay.

#### *Signed PRC, Empty Overlay Certificate ID List*

If the application is signed, but its `'sign'` resource doesn't contain any overlay certificate IDs, any overlay is allowed.

#### *Signed PRC, Overlay Certificate ID List has Entries*

If the application is signed and its 'sign' resource contains an overlay certificate ID list, an overlay must be verified with the certificate(s) in the list before it will be applied.

## Securing Databases

The Data Manager in Palm OS Cobalt supports secure schema databases. These databases have an Authorization Manager rule set associated with them that is evaluated when the database is opened or removed.

The Data Manager defines several different actions that the application can define access rules for:

- `dbActionRead`
- `dbActionWrite`
- `dbActionDelete`
- `dbActionBackup`
- `dbActionRestore`
- `dbActionEditSchema`

There are two functions for creating secure databases: `DbCreateSecureDatabase()`, and `DbCreateSecureDatabaseFromImage()`.

When you first create a secure database, access is only granted to the creator of that database, and the creator is only allowed to modify the database's access-control rule set. Accordingly, an application that creates a secure database must then set rules for those actions for which it wants to grant access.

A rule specifies a series of tokens that must be satisfied prior to access being granted.

The following code shows how to create a secure database that requires a user password for any action performed on that database.

**Listing 1.18 Creating a secure database**

```
AzmRuleSetType dbRuleSet;
AmTokenType usertoken;
UInt32 action = dbActionRead | dbActionWrite | dbActionDelete
   | dbActionBackup | dbActionRestore | dbActionEditSchema;
status_t err;

// Create DB — get AZM ruleset reference
err = DbCreateSecureDatabase(“My DB”, ‘crea’, ‘type’,
   numSchemas, schemaList, &dbRuleSet, &dbID);

// Set user password required for ALL actions
err = AmGetTokenBySystemId(&usertoken, SysUserToken);
err = AzmAddRule(dbRuleSet, action , "%t", usertoken);
```

## Synchronization and Backup of Secure Databases

“Sync bypass” must enabled for a specific action in order for an authenticated HotSync agent to be able to complete that action successfully. You use `AzmGetSyncBypass()` and `AzmSetSyncBypass()` to get and set the sync bypass flag in an existing rule-set container for a specific action. Sync bypass may be flagged for any action; it tells the sync server to grant access to registered sync clients. Sync bypass is used in both synchronization and backup operations.

### HotSync and Secure Databases

If sync bypass has been granted for the appropriate actions, any HotSync client that has registered with the HotSync server may access a secure database on the device, and any HotSync conduit may access a secure database from the desktop. Any application can register with the HotSync Manager (although user confirmation may be required).

Be aware that synchronizing a secure database exposes it on the desktop and on the device. The secure database is synchronized "in the clear," meaning that the security of the link between the desktop and the device is up to the HotSync client/server setup. For truly secure data synchronization is not recommended.

### Backing up Secure Databases

The database itself is always backed up encrypted, and the encryption key is backed up as well. In order to allow the database to be backed up, the backup action must be set in the bypass vector of the rule set. As well, all actions must be protected by the user token: only data that is protected solely by the user password can be backed up. This is enforced by the Data Manager.

On the desktop the backup image is protected by the user password. That means that the security of the backup depends on strength of the password. However, because it is protected by a user password on both the device and the desktop, the data is no more at risk on the desktop than it is on the device.

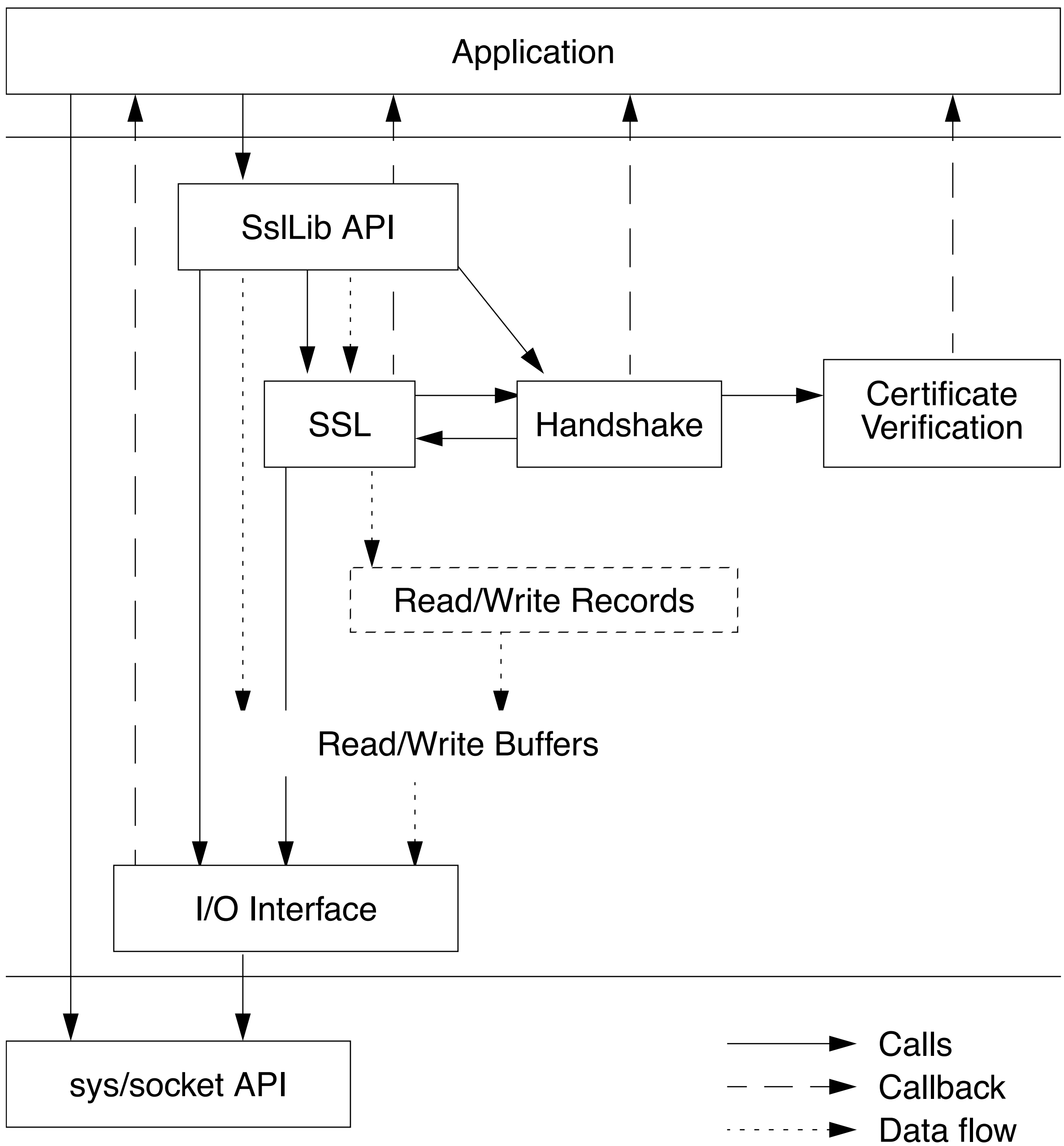
2

# SSL Concepts

## SSL Library Architecture

The SslLib library is an implementation of the SSL protocol for use under Palm OS. The API implements an interface that can be used to perform SSL and non-SSL network I/O. Figure 2.1 is intended to help show the relationship between the different components of SslLib and how they interact with the user's application.

**Figure 2.1 SSL library architecture**




In this diagram, the following items are labeled.

- Application – This is the user's application that will be using the SslLib library to secure its network connections.

- `sys/socket` API – This is the standard sockets API. This box represents calls into that library.
- SslLib API – This is the Palm OS SslLib API. This box represents calls into that library via it's public interfaces.
- SSL – The SSL protocol which is under the SslLib API. This represents the code that performs the SSL encapsulation of the application's data.
- Handshake – The SSL protocol, during the initial connection, performs a message exchange with the remote SSL server. This box represents the part of the SSL protocol that implements this exchange.
- Certificate Verification – As part of the SSL handshake, certificates need to be verified. This box represents the logic that performs the certificate verification.
- Read/Write Records – The SSL protocol sends and receives SSL records. This box represents the data structures used to keep track of the last record read and the next record to be written.
- Read/Write Buffers – SslLib buffers incoming and outgoing data. This box represents the data structures used to hold this data.
- I/O Interface – This is the code that sends data from a write buffer to the network, or the code that reads data from the network and puts it in the read buffer.

The application will call `sys/socket` directly to configure and establish a network connection (a descriptor referencing the socket). Once the socket has been configured, it is passed into SslLib by associating the socket with an SslContext (`SslContextSet_Socket()`). When a read or write call is made to SslLib, depending on the mode of operation the SslContext is configured to operate in (`SslContextSet_Mode()`), either the data bytes will be directly sent, or they will under go SSL processing to encrypt and MAC the data. The diagram shows how the data bytes always go via the SslContext's read/write buffers. These buffers are used to store bytes waiting to be sent to the socket and any extra bytes read from the socket that have not yet been processed. The SSL protocol initially enters a handshake state, where the security parameters to use to encrypt and MAC the application's data bytes are determined. As part of this process, some certificates need to be verified.

The callback arrows indicate where the application can register to receive notification of activity in those relevant subsystems. The I/O Interface can return via the info callback (`SslContextSet_InfoCallback()`) information about the calls to the socket APIs. The SSL box callback indicates the notification of SSL Protocol Alerts that are received (via the info callback). The handshake callback arrow indicates the calls to the info callback when-ever the SSL handshake protocol changes state (`SslContextGet_HsState()`). The information returned from these three access points is mostly of interest for debugging reasons. The final callback, the Verify callback (`SslContextSet_VerifyCallback()`) is often used to modify the policies regarding certificates.

# Critical Extensions

Extensions are an optional set of fields in X.509 certificates. Certain certificates, including SSL certificates, may have extensions. Some of these extensions may be classified as "critical," which means that they should be processed by the entity trying use the certificate. These extensions are defined by a Certificate Authority ("CA") to clarify and restrict the role of the certificate and its purpose. The SSL library in Palm OS Cobalt version 6.1 and earlier ignores two of these critical extensions:

**Basic Constraints:** The `BasicConstraints` extension is used to clarify the role and position of the certificate in the CA hierarchy. That is, root and sub CA certificates may contain a `BasicConstraints` extension that identifies them as CA certificates while end-entity certificates may be clearly identified as not being such.

**Key Usage:** `KeyUsage` extensions define the purpose of the public key contained in a certificate. The public key may be used for digital signature, non repudiation, key encipherment, data encipherment, key agreement, certificate signing, and certificate revocation list ("CRL") signing.

In Palm OS Cobalt version 6.1 and earlier, the SSL library does not process the `BasicConstraints` or `KeyUsage` extensions. If the SSL library finds a critical extension of any type, the error `CertMgrVerifyFailCriticalExtension` is returned to the application. In practice, this means that an application running on Palm OS Cobalt version 6.1 that tries to connect to a website that has a certificate with a critical `KeyUsage` or `BasicConstraints` extension will get this error even if the extension is correct and valid for the connection.

If a web server is using a certificate that has a `KeyUsage` extension indicating a usage not appropriate for SSL, this means that the certificate is being used for a different purpose than that stipulated by the Certificate Authority. This basically waives the liability of the CA for that particular certificate usage. If the root certificate of a particular end-entity certificate does not have a basic constraint specifying itself as a CA, or if the end-entity certificate specifies itself as a CA, that is a potential misuse of the certificate(s). However, regardless of whether a certificate has a `KeyUsage` or `BasicConstraints` extension that is appropriate or inappropriate for SSL, if the extension is marked critical, on Palm OS Cobalt version 6.1 the error `CertMgrVerifyFailCriticalExtension` is returned.

On Palm OS Cobalt 6.1 and earlier, it is up to the application to decide whether to continue or abort the SSL connection when it encounters a `CertMgrVerifyFailCriticalExtension` error. This issue is more related to the liability of a CA for the usage of a certificate not originally intended for SSL, rather than the actual SSL processing and the security of the connection.

# Attributes

The SslLib library uses two main structures to hold information: the SslLib structure and the SslContext. The SslContext is used to hold all information associated with a single SSL network connection. It contains various flags that govern how the SSL protocol will operate, and also contains a read buffer and a write buffer where SSL protocol packets are assembled and disassembled. As part of the SSL handshake, various structures are created . These include the security parameters associated with the particular connection and the certificate from the SSL server that is on the other end of the network connection. Quite a large number of these attributes can be retrieved for debugging and informational reasons. Others can be set by the application to modify the behavior of the SSL protocol. The SslLib can be though of as a template for many of these options. The SslLib can have many of its attributes set, and then when an SslContext is created using the SslLib, these attributes are inherited directly. These values are copied into the SslContext, so subsequent changes to the SslLib's attributes will not modify any existing SslContext's.

Attribututes can be broken into two main classes; integer values, and pointer values. The integer values are numbers that can be set or retrieved via the `SslLibGetLong()`, `SslLibSetLong()`, `SslContextGetLong()` and `SslContextSetLong()` calls. These functions are not normally called directly; instead, applications typically employ those macros declared in `SslLibMac.h`. The pointer-based attributes are similarly set or retrieved using macros; those macros evaluate to calls to `SslLibGetPtr()`, `SslLibSetPtr()`, `SslContextGetPtr()` and `SslContextSetPtr()`. Whenever an attribute is passed in via a pointer, the type of the pointer is defined by the attribute being used. The object that the pointer is pointing to is always copied into the SslLib or SslContext, so the data element that is passed in does not need to be preserved. There are some exceptions to this rule. Pointer-based attributes that are retrived from an SslLib or an SslContext will always be references to objects held inside the SslLib or SslContext. If the application wishes these values to be preserved, it should copy them into local storage.

The attributes can be grouped into several categories: some will always be used, some will be regularly used and will profoundly modify the behavior of some of SslLib core functions. Some are to help debugging, and some are used to configure more subtle protocol specific internal configuration parameters. The following sections detail each attribute, grouping them by these categories.

## Always-Used Attributes

### AutoFlush

This attribute affects the behavior of `SslSend()` and `SslWrite()`. When enabled, these functions will attempt to immediately send the supplied data bytes to the network. If the application performs 200 one-byte `SslWrite()` calls, this will generate 200 network packets, each about 80 bytes in size (assuming TCP over Ethernet), for a total of 16,000 bytes. If this data was buffered, it would have been sent in a single packet of about 280 bytes. When buffering, there is an additional advantage in that the write calls will not generate errors unless the buffer fills. This can be used to simplify routines that package data for transmission. It is very important to remember to use the `SslFlush()` call when AutoFlush is disabled. `SslFlush()` will write any data that is in the SslContext's write buffer. If an application does not flush this data to the network, the server application at the other end will not reply, so the application will probably deadlock, awaiting a response from the server that will never come because the client has not yet sent its data to the server.

The internal logic in SslLib is as follows:

```
Int32 SslWrite(...) {
write_data_to_output_buffer(...);
if (ssl->autoflush)
   flush_output_buffer(...);
}
```

Auto-flush is enabled by default.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_AutoFlush()`

**SslLib Write:** `SslLibSet_AutoFlush()`

**SslContext Read:** `SslContextGet_AutoFlush()`

**SslContext Write:** `SslContextSet_AutoFlush()`

### CipherSuites

This attribute is used to specify the SSL cipher suites that the SSL protocol will attempt to use. The pointer refers to an array of UInt8 bytes that specify the SSLv3 cipher suite values, in the order desired, to be sent to the SSL Server. The first two bytes, in network byte order, contain the number of bytes that follow. Following these two bytes are values selected from "Cipher Suites" on page 344. Note that each `sslCs_RSA_`*xxx* #define is two bytes long.

This value is inherited from the SslLib when an SslContext is created. Setting `CipherSuites` with a value of `NULL` will restore the use of the default cipher suite list. The default cipher suites list (including the size bytes) is:

```
{0x00, 0x08, sslCs_RSA_RC4_128_MD5, sslCs_RSA_RC4_128_SHA1,
sslCs_RSA_RC4_56_SHA1, sslCs_RSA_RC4_40_MD5}
```

To ensure that an application only uses strong encryption, it should make the following call:

```
static UInt8 cipherSuites[]={
   0x00,0x04,       /* Number of following bytes
                       (each value is two bytes) */
   sslCs_RSA_RC4_128_MD5,
   sslCs_RSA_RC4_128_SHA1
};

SslLibSet_CipherSuites(lib, cipherSuites);
/* To change the cipher suite for an existing SslContext */
SslContextSet_CipherSuites(lib, cipherSuites);
```

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_CipherSuites()`

**SslLib Write:** `SslLibSet_CipherSuites()`

**SslContext Read:** `SslContextGet_CipherSuites()`

**SslContext Write:** `SslContextSet_CipherSuites()`

### Error

When a fatal error occurs while using an SslContext, the internal error attribute is set to the error value. The application can retrieve this error value and change it if it desires. Normally an application will not change this value, but once the error attribute is set, the SslLib network APIs will continue to return this error (unless the error is a non-fatal error) until either an SSL Reset is performed on the SslContext or the error is cleared, at which point the Error attribute will be zero. A SSL Reset can be performed with `SslContextSet_Mode()`:

```
SslContextSet_Mode(ssl,SslContextGet_Mode(ssl));
```

Note that `SslErrIo` is a non-fatal error.

Use the following macros to read and write this attribute:

**SslContext Read:** `SslContextGet_Error()`

**SslContext Write:** `SslContextSet_Error()`

### Mode

This attribute is used to turn the SSL protocol on or off. It applies to the SslContext, and when set to `sslModeClear`, causes the SSL protocol to be bypassed. This can be useful for an application since it can be written to use the SslLib API, and still perform normal non-SSL data transfers via that API. This will let an application take advantage of the buffering provided in an SslContext so that it can perform buffer reads and buffer writes to the network. When an SslContext has its Mode attribute changed, an SSL Reset occurs. This clears any SSL state information and sets the SslContext back to a state ready to establish a new SSL connection. The SSL Session information is not cleared. This means that an application can start in `sslModeClear`, and then switch to `sslModeSslClient`. If the application switches back to `sslModeClear`, and again over to `sslModeSslClient`, a new handshake will be performed.

The `SslModeSsl` is a subset value of `sslModeSslClient`. In a future release of SslLib, the server side of the SSL protocol may be supported in which case `sslModeSslServer` would be added.

An application can do the following in order to determine if the SSL protocol is being used:

```
If (SslContextGet_Mode(ssl) & sslModeSsl)
   /* SSL protocol enabled */
else
   /* Using cleartext */
```

A comparison with `sslModeSslClient` could be used to determine if the client or server side of the protocol is being used for that particular SslContext.

The `sslModeFlush` flag is special. When supplied to `SslContextSet_Mode()`, it causes any data in the internal data buffers to be cleared. This is normally required when reusing an SslContext for a new connection. If an application is using an SslContext for cleartext, and then wants to enable SSL on the same connection, this flag should not be used.

By default, the `mode` attribute is set to `sslModeSslClient`.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_Mode()`

**SslLib Write:** `SslLibSet_Mode()`

**SslContext Read:** `SslContextGet_Mode()`

**SslContext Write:** `SslContextSet_Mode()`

"Mode Attribute Values" on page 339 lists the values that this attribute can have.

### RbufSize

The read and write buffers are used in the SslContext to buffer incoming and outgoing data. When these values are set for an SslLib, SslContexts that are created against the SslLib will inherit the SslLib's values.

The write buffer size is the maximum number of bytes that can be buffered before a network write operation is performed. The number of application data bytes that can be buffered is less than this number when in SSL mode—approximately 30 bytes less due to SSL record overheads. If the application writes a 16 kb block of data and the write buffer is about 1 kb in size, about 16 network packets will be sent.

The read buffer is a little different from the write buffer in that it may be automatically increased is size depending on other configuration information. The SSLv3 protocol supports SSL records up to 16 Kbytes in size. Depending on the encryption cipher being used, the protocol may need to decrypt the record in a single operation. In this case the read buffer will be increased in size to buffer the incoming record. See the ReadStreaming option for advanced usage of the read buffer to decrease latency of data availability for the application.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_RbufSize()`

**SslLib Write:** `SslLibSet_RbufSize()`

**SslContext Read:** `SslContextGet_RbufSize()`

**SslContext Write:** `SslContextSet_RbufSize()`

The read buffer's default size is 2048 bytes. You can change the size of the read buffer to any value from 0 to 16384 bytes.

### Socket

This call is used to specify the socket that the SslContext should use to perform its network I/O operations. An SslContext is unable to perform any network operation until the application creates and supplies a suitable socket descriptor. The SslLib library does not perform any operations on the supplied descriptor other than `sendto()` and `recvfrom()`. All socket creation and shutdown operations must be performed by the application.

Use the following macros to read and write this attribute:

**SslContext Read:** `SslContextGet_Socket()`

**SslContext Write:** `SslContextSet_Socket()`

### VerifyCallback

The callback function is used to assist with certificate verification. See `SslCallbackFunc()` (documented on page 373) for more details on the `SslCallback` structure and its usages, specifically when used to assist in certificate verification.

When a new Verify callback is specified, the application passes in a pointer to an `SslCallback` structure. This structure is copied into an internal `SslCallback` structure. The callback and data fields are preserved. When the Verify callback structure is copied into an SslLib, or copied into an SslContext, the callback function is called with a command of `sslCmdNew`. When the parent SslLib or SslContext is destroyed, a `sslCmdFree` command is issued.. If a

SSL Reset is performed, a `sslCmdReset` command is issued. Outside of these situations, the callback will be called during the certificate verification process as outlined in the documentation for the `SslCallbackFunc()` function.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_VerifyCallback()`

**SslLib Write:** `SslLibSet_VerifyCallback()`

**SslContext Read:** `SslContextGet_VerifyCallback()`

**SslContext Write:** `SslContextSet_VerifyCallback()`

### WbufSize

The read and write buffers are used in the SslContext to buffer incoming and outgoing data. When these values are set for an SslLib, SslContexts that are created against the SslLib will inherit the SslLib's values.

The write buffer size is the maximum number of bytes that can be buffered before a network write operation is performed. The number of application data bytes that can be buffered is less than this number when in SSL mode—approximately 30 bytes less due to SSL record overheads. If the application writes a 16 kb block of data and the write buffer is about 1 kb in size, about 16 network packets will be sent.

The read buffer is a little different from the write buffer in that it may be automatically increased is size depending on other configuration information. The SSLv3 protocol supports SSL records up to 16 Kbytes in size. Depending on the encryption cipher being used, the protocol may need to decrypt the record in a single operation. In this case the read buffer will be increased in size to buffer the incoming record. See the ReadStreaming option for advanced usage of the read buffer to decrease latency of data availability for the application.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_WbufSize()`

**SslLib Write:** `SslLibSet_WbufSize()`

**SslContext Read:** `SslContextGet_WbufSize()`

**SslContext Write:** SslContextSet_WbufSize()

The write buffer's default size is 1024 bytes. You can change the size of the write buffer to any value from 0 to 16384 bytes.

## Debugging and Informational Attributes

### AppInt32

The AppInt32 attribute is a 32-bit integer value that the application can read or write as it sees fit. It is present so the application can attach an arbitrary value to an SslLib or a SslContext. SslLibDestroy() and SslContextDestroy() do not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

Use the following macros to read and write this attribute:

**SslLib Read:** SslLibGet_AppInt32()

**SslLib Write:** SslLibGet_AppInt32()

**SslContext Read:** SslContextGet_AppInt32()

**SslContext Write:** SslContextSet_AppInt32()

### AppPtr

The AppPtr attribute is a pointer value that the application can read or write as it sees fit. It is present so the application can attach an arbitrary pointer to an SslLib or a SslContext. SslLibDestroy() and SslContextDestroy() do not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself. The value of the AppPtr attribute is NULL by default.

Use the following macros to read and write this attribute:

**SslLib Read:** SslLibGet_AppPtr()

**SslLib Write:** SslLibGet_AppPtr()

**SslContext Read:** SslContextGet_AppPtr()

**SslContext Write:** SslContextSet_AppPtr()

### CipherSuite

Pass a pointer to a uint8_t pointer in order to retrieve this attribute. The returned value points to two bytes which identify the cipher suite being used by the current connection. Possible values for the cipher suites are:

`0x00, 0x00`
: No cipher suite

`0x00, 0x04`
: sslCs_RSA_RC4_128_MD5

`0x00, 0x05`
: sslCs_RSA_RC4_128_SHA1

`0x00, 0x64`
: sslCs_RSA_RC4_56_SHA1

`0x00, 0x03`
: sslCs_RSA_RC4_40_MD5

Also see the `CipherSuites` attribute for instructions on specifying which cipher suites SslLib should advertise as available for use when it initially connects to the SSL server.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_CipherSuite()`

### CipherSuiteInfo

This function differs from most others in that the application passes in a structure to be populated from the SslContext. Normally the SslContext returns a pointer to an internal data structure. This call returns the information relevant to the current cipher suite.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_CipherSuiteInfo()`

### ClientCertRequest

The SSL protocol allows the SSL server to request a certificate from the client. This attribute will be set if the server requested a client certificate.

**SslContext Read:** `SslContextGet_ClientCertRequest()`

### Compat

Turn on compatibility with incorrect SSL protocol implementations. These bugs will not normally be encountered while using the SSL protocol, but if desired, it is worth enabling the compatibility in case old buggy servers are being accessed.

See "Compatibility Flags" on page 342 for the defined constants that correspond to the compatibility flags. By default, none of these compatibility flags are set.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_Compat()`

**SslLib Write:** `SslLibSet_Compat()`

**SslContext Read:** `SslContextGet_Compat()`

**SslContext Write:** `SslContextSet_Compat()`

### HsState

This attribute is the state that the SSL protocol is currently in. Possible values are defined under "SSL Protocol States" on page 349. This information is generally only of use during debugging. See the SSL protocol specification for clarification on what the values mean.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_HsState()`

### InfoCallback

This callback is called when various situations occur during the usage of an SslContext. It is primarily intended for debugging and feedback purposes. If the callback returns a non-zero value, this error will be returned back out to the SslLib API. The callback will be called with a command argument of `sslCmdInfo`.

A single Info callback is used for notification of four different types of events. The `InfoInterest` attribute controls which of these events will invoke the Info callback.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_InfoCallback()`

**SslLib Write:** `SslLibSet_InfoCallback()`

**SslContext Read:** `SslContextGet_InfoCallback()`

**SslContext Write:** `SslContextSet_InfoCallback()`

### InfoInterest

This value is used to specify the events for which the `InfoCallback` will be called. The value is the logical ORing of the `sslFlgInfo`*xxx* values listed under "InfoInterest Values" on page 347. The `sslFlgInfoIo` value controls the notification of the four different Info Callbacks. By default, the `InfoInterest` attribute value is zero.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_InfoInterest()`

**SslLib Write:** `SslLibSet_InfoInterest()`

**SslContext Read:** `SslContextGet_InfoInterest()`

**SslContext Write:** `SslContextSet_InfoInterest()`

### IoFlags

Since we will normally be using TCP connections with SSL, this attribute is more included for completeness rather than utility. Read about this flags value in the `sendto()` and `recvfrom()` man pages.

---

**NOTE:** The `MSG_OOB` and `MSG_PEEK` values are not valid and will be silently removed.

---

Use the following macros to read and write this attribute:

**SslContext Read:** `SslContextGet_IoFlags()`

**SslContext Write:** `SslContextSet_IoFlags()`

### IoStruct

The SslContext's internal SslSocket structure.

Use the following macros to read and write this attribute:

**SslContext Read:** `SslContextGet_IoStruct()`

**SslContext Write:** `SslContextSet_IoStruct()`

### IoTimeout

The SslContext contains internally a default timeout value to pass to `sys/socket` calls. When a call is made into the SslLib API which does not specify a timeout, this internal value is used. If the API call has a timeout value, it overrides this internal value.

By default, the SslContext's internal timeout value is 10 seconds.

Use the following macros to read and write this attribute:

**SslContext Read:** `SslContextGet_IoTimeout()`

**SslContext Write:** `SslContextSet_IoTimeout()`

### LastAlert

The alert values are received from the server and are either fatal or non-fatal. Non-fatal alerts have a value of the form `0x01`*XX*, while fatal alerts have the form `0x02`*XX*. SslLib will fail on fatal alerts and continue on non-fatal alerts. See "SSL Server Alerts" on page 350 for the complete list of alerts.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_LastAlert()`

### LastApi

This attribute is the last SslLib API call that was made. `sslLastApiRead` is set if `SslRead()`, `SslPeek()` or `SslReceive()` was called. `sslLastApiWrite` is set if `SslWrite()` or `SslSend()` was called. This attribute can be useful in event driven programs.

See "LastApi Attribute Values" on page 347 for the set of values that this attribute can have.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_LastApi()`

### LastIo

This function can be called to determine the last network operation. If SslLib, while performing a network operation, encounters an error, the error value will be returned to the application. Since most of the SslLib API I/O functions can cause an SSL handshake to be performed, it is often not possible to know if the reason that a `SslSend()` returned `netErrWouldBlock` is because the send operation failed or a receive operation failed (because a SSL Handshake was being performed). This attribute allows the application to determine which I/O operation was being called if an network error is returned. If the application is using `select()`, this attribute is very important. This attribute returns the last network operation performed. This means that `sslLastIoNone` will only be returned if the SslContext has not performed any I/O operations since its last reset.

See "LastIO Attribute Values" on page 348 for the set of values that this attribute can have.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_LastIo()`

### PeerCert

If the certificate supplied by the other end of the SSL connection is available, the certificate is returned. The returned pointer references a data structure which is internal to the SslContext and will be disposed of by the SslContext. If a new connection is established with the SslContext, previously returned `PeerCert` pointers will become invalid. If the application wishes to preserve the certificate for an extended period, it should make a local copy.

The `SslExtendedItems` structure is described in "The SslExtendedItems Structure" on page 380.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_PeerCert()`

### PeerCommonName

This call will return a pointer to an `SslExtendedItems` structure which contains the common name for the server's certificate. If using SSL in an https context, the client application should ensure that the common name contained in the servers certificate matches the URL requested. This function facilitates this functionality. The pointer returned refers to a data structure from inside the peer certificate; the offset field in the returned value is relative to the value returned by `SslContextGet_PeerCert()`.

The following code shows how to access the common name from within the `SslExtendedItems` structure (see "The SslExtendedItems Structure" on page 380 for a description of this structure):

```
SslExtendedItems *cert;
SslExtendedItem *commonName;
uint16_t length;
uint8_t *bytes;

SslContextGet_PeerCert(ssl, &cert);
if (cert == NULL) goto err;
SslContextGet_PeerCommonName(ssl,&commonName);
length=commonName->len;
bytes=((Int8 *)cert)+commonName->offset;
// bytes now points to the common name, and length contains
// the length of the common name string.
```

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_PeerCommonName()`

### ProtocolVersion

The version of the SSL protocol to use. There are 3 versions of the SSL protocol. SSLv2 which is deprecated due to security flaws, SSLv3 which is the most widely deployed, and TLSv1, or SSLv3.1. SslLib sends a TLSv1 ClientHello message by default. Note that in Palm OS Cobalt version 6.0 an attempt to change this protocol version to SSLv3 via `SslContextSet_ProtocolVersion()` has no effect—SslLib continues to send a TLSv1 ClientHello message.

See "Protocol Versions" on page 340 for the defined constants that correspond to the SSL protocol versions.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_ProtocolVersion()`

**SslLib Write:** `SslLibSet_ProtocolVersion()`

**SslContext Read:** `SslContextGet_ProtocolVersion()`

**SslContext Write:** `SslContextSet_ProtocolVersion()`

### SessionReused

The SSL protocol has the capability to re-establish a secure connection with a truncated handshake. This can be performed if both end-points have communicated previously and share an SSL Session. An SSL Session is a collection of security attributes that are normally determined as part of the SSL Handshake. If the SSL handshake was able to perform a truncated handshake by re-using the SSL session values in the SslContext, this attribute will have a non-zero value. See the `SslSession` attribute.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_SessionReused()`

### SslSession

This attribute is either the `SslSession` currently being used, or the `SslSession` for this SslContext to use to establish its next connection. The `SslSession` holds all the security information associated with a particular SSL connection. If an SslContext is configured to use the same `SslSession` as a previous connection to the same server, the SSL protocol can perform a truncated handshake that involves less network traffic and a smaller CPU load on the server.

If a new connection is performed on the SslContext, or another call is made to retrieve the `SslSession`, any previously returned `SslSession` pointers will become invalid. If the program wants to keep the `SslSession` for an extended period, it should make a local copy.

Use the following macros to read and write this attribute:

**SslContext Read:** `SslContextGet_SslSession()`

**SslContext Write:** `SslContextSet_SslSession()`

### SslVerify

During certificate verification, an `SslVerify` structure (see "The SslVerify Structure" on page 377 for a definition of this structure) is used in the SslContext to preserve state. The application can retrieve this structure to help it resolve any problems that SslLib may have encounterd during certificate verifcation.

When a certificate is being verified and a verification error occurs, if the application has registered a `VerifyCallback` the callback will be called with an *argv* value pointing to the `SslVerify` structure. If there is no callback, or if the callback still reports an error, SslLib will return the error back to the application. The application can then decide to look at the certificate verification state (by calling `SslContextGet_SslVerify()`) and, if it determines that the error is not fatal, clear the error and re-call the SslLib API that just returned the error.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_SslVerify()`

## Advanced Protocol Attributes

The following attributes are not normally used. They give access to various internal aspects of the SSL protocol and or SslLib.

### BufferedReuse

The SSL protocol is capable of performing a truncated handshake if both endpoints share an SslSession from a previous connection. The truncated handshake finishes with SslLib sending a SSL handshake message to the SSL server. If the application then sends a message, say a URL, under some network stacks a significant delay can be incurred as the TCP protocol waits for a response from the SSL server's TCP stack. This option, if enabled, will buffer the last message in an SslSession-reused handshake instead of sending it over the network. The application *must* send data before it tries to read any, or more to the point, it must make sure the data is flushed, ether by having `AutoFlush` enabled, or by explicitly calling `SslFlush()`. There are security implications in that a "man in the middle" attack would only be detected once the first data bytes are read from the server. This would mean an attacker could have read all the bytes in the first message sent to the server. For this reason this option should not be normally used. By default, this attribute is set to zero, disabling the buffered reuse option.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_BufferedReuse()`

**SslLib Write:** `SslLibSet_BufferedReuse()`

**SslContext Read:** `SslContextGet_BufferedReuse()`

**SslContext Write:** `SslContextSet_BufferedReuse()`

### DontSendShutdown

During the SSL protocol shutdown sequence, the two SSL endpoints swap shutdown messages. This can incur a time penalty since extra messages need to be exchanged over the network. If DontSendShutdown is set, then a `SslClose()` will not send a shutdown message to the server. If `DontWaitForShutdown` is set, then SslLib will not wait for a shutdown message in `SslClose()`. To perform a correct SSL shutdown, these options should not be on.

This attribute has a default value of zero. A non-zero value indicates that the SSL protocol should be modified.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_DontSendShutdown()`

**SslLib Write:** `SslLibSet_DontSendShutdown()`

**SslContext Read:** `SslContextGet_DontSendShutdown()`

**SslContext Write:** `SslContextSet_DontSendShutdown()`

### DontWaitForShutdown

During the SSL protocol shutdown sequence, the two SSL endpoints swap shutdown messages. This can incur a time penalty since extra messages need to be exchanged over the network. If DontSendShutdown is set, then a `SslClose()` will not send a shutdown message to the server. If `DontWaitForShutdown` is set, then SslLib will not wait for a shutdown message in `SslClose()`. To perform a correct SSL shutdown, these options should not be on.

This attribute has a default value of zero. A non-zero value indicates that the SSL protocol should be modified.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_DontWaitForShutdown()`

**SslLib Write:** `SslLibSet_DontWaitForShutdown()`

**SslContext Read:** `SslContextGet_DontWaitForShutdown()`

**SslContext Write:** `SslContextSet_DontWaitForShutdown()`

### ReadBufPending

This attribute is the number of data bytes that are currently buffered for reading from the SslContext. This number of bytes also include bytes used for encoding SSL records. This attribute is mostly for debugging purposes.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_ReadBufPending()`

### ReadOutstanding

This attribute is the number of bytes in the current record that have not been read from the network. If this value is 0, then all bytes that have been read from the network have had their MAC checked. If it is not 0, then the last bytes that have been read have not had their MAC value checked yet. See the Streaming and ReadStreaming attributes to see why this value can be useful.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_ReadOutstanding()`

### ReadRecPending

Unlike ReadBufPending, this attribute is the number of application data bytes that are buffered, awaiting the application to read. If this number of bytes is 0, then the next `SslRead()` or `SslReceive()` will cause a `recvfrom()` call.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_ReadRecPending()`

### ReadStreaming

The SSL protocol exchanges records between its endpoints. A SSL record can contain up to 16K bytes of data. This record is encrypted and protected with a cryptographic checksum call a MAC. If the network is very low speed (300 baud modem), it can be desirable to allow data to be returned to the application from the SSL connection before the full record has been downloaded. If the ReadStreaming flag is on, this protocol modification is enabled. There are security implications behind this modification. The record MAC is used to ensure that the data bytes downloaded have not been modified. If the application has been sent a 16K record, and it is read-streaming and only processing 300 bytes at a time, those bytes could be corrupted or forged without SslLib notifiying the application of this error until the last bytes of the 16K of data is sent. This attribute can be useful if the application is displaying or saving the downloaded data and does not want to be stuck in a `SslRead()` for an extended period of time. Remember that if read-streaming is turned on, the data may be invalid and you will only receive notification when the last bytes are read from the record.

This attribute has a default value of zero. A non-zero value indicates that the SSL protocol should be modified.

Use the following macros to read and write this attribute:

**SslLib Read:** `SslLibGet_ReadStreaming()`

**SslLib Write:** `SslLibSet_ReadStreaming()`

**SslContext Read:** `SslContextGet_ReadStreaming()`

**SslContext Write:** `SslContextSet_ReadStreaming()`

### Streaming

This attribute returns 1 if the current SslContext is doing read-streaming. Just because the ReadStreaming attribute is set, that does not mean the SslLib will use read-streaming.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_Streaming()`

### WriteBufPending

This attribute returns the number of bytes in the SslContext's write buffer waiting to be sent to the remote machine. This value will normally be zero unless AutoFlush is disabled and/or non-blocking I/O is being used. A `SslFlush()` will attempt to write these bytes to the network.

Use the following macro to read this attribute:

**SslContext Read:** `SslContextGet_WriteBufPending()`

# Sample Code

The following is a simple example that demonstrates the usage of some of the SslLib libraries functions by way of listing subroutines that could be used by an application utilizing the SSL protocol.

```
#include <SslLib.h>

/* We will perform the initial SslLib setup.  The SslLib would be
 * created with reasonable default values, which can be modified.
 * Quite a few of these values are
 * 'inherited' during SslContext creation.
 */
Err InitaliseSSL(libRet)
SslLib **libRet;
   {
   SslLib *lib;
   Int16 err;
   lnt32 lvar;

   /* Create the structure */
   if ((err=SslLibCreate(&lib)) != 0)
      return(err);

   /* Make sure we use the SSL protocol by default and increase
 * the write buffer size */
   SslLibSet_Mode(lib,sslModeSslClient);
   SslLibSet_WbufSize(lib,1024*8);

   *libRet=lib;
   return(0);
   }

/* This function would be called to create an sslContext from an open socket
 */
Err CreateSslConnection(SslLib *lib, int socket,SslContext **sslRet)
   {
   SslContext ssl=NULL;
   Int16 err;

   /* We first create a new SslContext.
    * This context will inherit various internal configuration
    * details from the SslLib.
    */
   if ((err=SslContextCreate(lib,&ssl)) != 0)
      return(err);
```

```
    /* We now specify the socket to use for IO */
    SslContextSet_Socket(ssl,socket);

    /* At this point we could specify the SSL Mode of operation to use,
     * but since we already specified this for the SslLib, we do not
     * need to do it again.
     */
    //SslContextSet_Mode(ssl,sslModeSslClient);

    /* For this example, we will perform the SSL handshake now. */
    err=SslOpen(ssl,0,30*SysTicksPerSecond());

    *sslRet=ssl;
    return(Err);
    }

/* Shutdown the SSL protocol and return the socket */
Err CloseSslConnection(SslContext ssl, int *retSock)
    {
    int socket;
    SslSession *sslSession;
    MemHandle ssHandle;

    /* We will perform a full SSL protocol shutdown.  We could have
     * set a flag against the SslContext earlier, or even against the
     * SslLib to specify the shutdown behavior.
     */
    err=SslClose(ssl,0,10*SysTicksPerSecond());

    /* We have now closed the SSL protocol, but the socket is still open
     * and the SslContext still has SslSession information that
     * other connections to the same site may want to use.
     * In this case we ask for a reference to the SslSession.
     * Since this structure is variable in size, once we have a
     * reference to it, we can duplicate it if we want to keep it.
     */
    SslContextGet_SslSession(ssl,&sslSession);

    ssHandle=MemHandleNew(sslSession->length);
    Memcpy(MemHandleLock(ssHandle),sslSession,sslSession->length)
    MemHandleUnlock(ssHandle);
    /* We now have a handle to a SslSession that we can store
     * for later use with a new connection.  We would need to store
     * this SslSession with the relevant hostname/url information
     * to ensure we try to reuse it on only the relevant SSL server.
     * This mapping is application/protocol-specific (urls for https).
     */
```

```
   /* We will return the Socket */
   *retSock=SslContextGet_Socket(ssl);

   /* Throw away the SslContext structure */
   SslContextDestroy(ssl);
   return(0);
   }

Err HTTPS_call(SslContext ssl,char *send,Uint16 len,char *reply,Uint32 *outlen)
   {

   Err err;
   Int16 ret;

   /* We will send the 'send' data, and then wait for the response */
   ret=SslSend(ssl,send,len,0,NULL,0,60*SysTicksPerSecond(),&err)
   if (ret <= 0) goto end;

   ret=SslReceive(ssl,reply,*outlen,0,NULL,0,60*SysTicksPerSecond(),&err);
   if (ret < 0) goto end;
   *outlen=ret;
end:
   return(err);
   }
```

# Part II
# Reference

This part contains reference documentation for the following:

# 3 Authentication Manager

The Authentication Manager provides abstract methods for authenticating access to protected objects. It allows modules (plug-ins) to be written that implement specific authentication scenarios, such as PIN, PKI, or password.  Users of the Authentication Manager deal with generic interfaces and opaque objects that define an authentication context.

The Authentication Manager is the authority that can answer the question "Are you X?" reliably, by utilizing some method of identity verification. The question "Are you X?" may be asked either about a user or an application.

The services provided by the Authentication Manager handle the following tasks: credential (token) management (creation, deletion, modification, and storage), authentication against stored credentials (querying user for system password), and a framework for run-time extensibility via plug-ins.

Every authentication model is a plug-in. Palm OS Cobalt includes three authentication models: password, signed code, and hashed code.

The remainder of this chapter documents the Authentication Manager APIs. It is organized into the following sections:



The header file `Am.h` declares the API that this chapter describes.

# Authentication Manager Structures and Types

## AmApplicationCtxType Struct

**Purpose** A data structure that is prepared by the caller of the Authentication Manager, it holds information about the application that needs to be authenticated and other private data specific to the plug-in. The data for the plug-in is defined by plug-in type.

**Declared In** `Am.h`

**Prototype**

```
typedef struct {
   uint32_t processIDKey;
   AmTokenEnum dataType;
   uint8_t padding1;
   uint16_t padding2;
   union {
      struct {
         char *password;
         uint32_t passwordLength;
         char *hint;
         uint32_t hintLength;
      } passwordCtxType;
      struct {
         uint8_t *certificateId;
         uint32_t certificateIdLength;
      } signatureCtxType;
      struct {
         uint32_t type;
         uint32_t creator;
         char *dbname;
         uint32_t dbnameLength;
      } appFingerprintCtxType;
      struct {
         uint8_t *dataPtr;
         uint32_t dataLength;
      } customCtxType;
   } data;
} AmApplicationCtxType, *AmApplicationCtxPtr
```

**Fields**

`processIDKey`
: Used during authentication. The key ID of the application being authenticated must be placed here. Mostly used by the

Authorization Manager, but anyone calling `AmAuthenticateToken()` should fill this member in.

`dataType`
: The type of data being passed. This is one of the `AmTokenEnum` values, and it controls which of the `data` enum data structures applies.

`padding1`
: Padding bytes.

`padding2`
: Padding bytes.

`passwordCtxType`
: Data structure used when creating a password token.

  `password`
  : The clear-text password. Depending on the call, this field is used to verify or set the password.

  `passwordLength`
  : Length of the password buffer, bytes.

  `hint`
  : Hint to save in the new password token.

  `hintLength`
  : Length of the hint buffer, in bytes.

`signatureCtxType`
: Data structure used when creating a signed-code token.

  `certificateId`
  : ID of the certificate. This field is used during creation.

  `certificateIdLength`
  : Length of the certificate ID buffer, in bytes.

`appFingerprintCtxType`
: Data structure used when creating a code-fingerprint token.

  `type`
  : The type of the application’s resource database.

  `creator`
  : The creator ID of the application’s resource database.

`dbname`
The name of the application's resource database.

`dbnameLength`
The length, in bytes, of the application's resource database name.

`customCtxType`
Data structure used when both creating and verifying a custom token.

`dataPtr`
A data buffer passed into the plug-in.

`dataLength`
The length, in bytes, of the data buffer passed into the plug-in.

## AmPluginInfoType Struct

**Purpose** A structure through which the Authorization Manager shares information about a plug-in with applications. Use `AmGetPluginInfo()` to get plug-in information.

**Declared In** `Am.h`

**Prototype**

```
typedef struct {
   char friendlyName[amPluginFriendlyNameLength];
   char vendor[amPluginVendorLength];
   uint32_t version;
   AmTokenPropertiesType tokenProperties;
} AmPluginInfoType, *AmPluginInfoPtr
```

**Fields**

`friendlyName`
A "friendly" name for the plug-in that can be used for display purposes.

`vendor`
A string that identifies the vendor of this plug-in.

`version`
The plug-in's version.

`tokenProperties`
The properties of the tokens that this plug-in creates. A `AmTokenPropertiesType` value.

## AmPluginType Typedef

**Purpose** Reference to a token.

**Declared In** `Am.h`

**Prototype** `typedef uint32_t AmPluginType`

**Comments** Supply a token reference of this type when getting information about a plug-in (`AmGetPluginInfo()`). When you retrieve a list of all plug-ins (with `AmGetPluginReferences()`), you receive a set of `AmPluginType` values.

## AmTokenAttributesType Struct

**Purpose** Flags that indicate various token attributes. The plug-in sets these values and uses them to allow or reject certain actions.

**Declared In** `Am.h`

**Prototype**

```
typedef struct {
   int destroy:1;
   int modify:1;
   int interactive:1;
   int empty:1;
   int system:1;
   int reserved:11;
   uint16_t padding;
} AmTokenAttributesType, *AmTokenAttributesPtr
```

**Fields**

`destroy`
The token may be destroyed.

`modify`
The token can be modified.

`interactive`
The token is user interactive. That is, it is a password, PIN, or the like.

`empty`
The token is empty.

`system`
The token is a system token.

`reserved`
Reserved for future use.

`padding`
Padding bits.

## AmTokenInfoType Struct

**Purpose** The public info block that the plug-in fills in. Applications may request this info block by calling `AmGetTokenInfo()`.

**Declared In** `Am.h`

**Prototype**

```
typedef struct {
   AmTokenType ref;
   AmTokenEnum type;
   AmTokenCacheSettings cacheSettings;
   AmTokenStrength strength;
   uint8_t rfu;
   AmTokenAttributesType attributes;
   uint8_t systemId[amTokenSystemIdLength];
   char friendlyName[amTokenFriendlyNameLength];
} AmTokenInfoType, *AmTokenInfoPtr
```

**Fields**

`ref`
Reference to the token.

`type`
The token's type. One of the `AmTokenEnum` values.

`cacheSettings`
The cache settings that govern this token. One of the `AmTokenCacheSettings` values.

`strength`
The strength of the token. One of the `AmTokenStrength` values.

`rfu`
Reserved for future use.

`attributes`
Token attribute flags. See `AmTokenAttributesType` for the attribute flag values.

`systemId`
: The token's system ID. Token system IDs will never exceed `amTokenSystemIdLength` bytes in length.

`friendlyName`
: A "friendly" name for the token that can be used for display purposes.

## AmTokenPropertiesType Struct

**Purpose** Structure containing various token properties. Used when creating or modifying a token with `AmCreateToken()` or `AmModifyToken()`.

**Declared In** `Am.h`

**Prototype**

```
typedef struct {
   uint32_t identifier;
   AmTokenEnum type;
   AmTokenStrength strength;
   AmTokenCacheSettings cacheSettings;
   uint8_t rfu;
} AmTokenPropertiesType, *AmTokenPropertiesPtr
```

**Fields**

`identifier`
: Identifier for the plug-in that will service this token.

`type`
: The token's type. One of the `AmTokenEnum` values.

`strength`
: The strength of the token. One of the `AmTokenStrength` values.

`cacheSettings`
: The cache settings that will govern this token. One of the `AmTokenCacheSettings` values.

`rfu`
: Reserved for future use.

**Comments** This structure is used during token creation. The application creating the token fills in the token type, strength and cache settings.

If the token is supported by a custom plug-in, the application should fill in the custom identifier.  The Authentication Manager will match the custom identifier to the custom identifiers registered with the Authentication Manager.

### AmTokenType Typedef

**Purpose** Reference to a token.

**Declared In** `Am.h`

**Prototype** `typedef uint32_t AmTokenType, *AmTokenPtr`

**Comments** Supply a token reference of this type when creating, modifying, destroying, or getting information about a token.

## Authentication Manager Constants

### Well-Known Tokens

**Purpose** System IDs of various well-known tokens.

**Declared In** `Am.h`

**Constants** `#define SysAdminToken "SysAdminToken"`
The Administrator token.

`#define SysEmptyToken "SysEmptyToken"`
The empty token.

`#define SysLockOutToken "SysLockOutToken"`
The “lockout” token.

`#define SysUserToken "SysUserToken"`
The User token.

## Miscellaneous Authentication Manager Constants

**Purpose** The header file Am.h also declares these constants.

**Declared In** `Am.h`

**Constants**

`#define amInvalidToken 0xFFFFFFFF`
An invalid token value. If `AmCreateToken()` fails, it returns `amInvalidToken` as a token value.

`#define amPluginFriendlyNameLength 48`
The maximum length, in bytes, of a plug-in's "friendly" name--a name that can be displayed to the user.

`#define amPluginVendorLength 48`
The maximum length, in bytes, of a plug-in's vendor string.

`#define AmServiceName "psysAuthenticationMgr"`
The name under which the Authentication Manager is registered with the Service Manager.

`#define amTokenFriendlyNameLength 36`
The maximum length, in bytes, of a token's "friendly" name--a name that can be displayed to the user.

`#define amTokenSystemIdLength 20`
The maximum length, in bytes, of a token's system ID.

`#define amTokenTypeIdentifierLength 8`

## Authentication Manager Error Codes

**Purpose** Error codes returned by the various Authentication Manager functions.

**Declared In** `Am.h`

**Constants**

`#define amErrActionNotSupported (amErrorClass | 0x16)`
The required action is not suppoted for this token by its plug-in.

`#define amErrAlreadyRegistered (amErrorClass | 0x0D)`
The specified plug-in was already registered.

`#define amErrAuthenticationFailed (amErrorClass | 0x10)`
The authentication operation failed.

`#define amErrBackupInProgress (amErrorClass | 0x17)`
A backup is in progress.

`#define amErrBufferTooSmall (amErrorClass | 0x14)`
The supplied buffer is too small.

`#define amErrInvalidImportBuffer (amErrorClass | 0x15)`
The supplied import buffer is invalid.

`#define amErrInvalidParam (amErrorClass | 0x02)`
One or more function parameters is invalid.

`#define amErrInvalidPlugin (amErrorClass | 0x0B)`
The specified plug-in reference is invalid.

`#define amErrInvalidToken (amErrorClass | 0x09)`
The specified token reference is invalid.

`#define amErrLibNotOpen (amErrorClass | 0x03)`
The library has not been opened.

`#define amErrLibStillOpen (amErrorClass | 0x04)`
An attempt to close the library returned without closing.

`#define amErrMaxPlugins (amErrorClass | 0x0A)`
The maximum number of plug-ins allowed in the system has been reached.

`#define amErrMaxTokens (amErrorClass | 0x08)`
The maximum number of tokens allowed in the system has been reached.

`#define amErrMemory (amErrorClass | 0x06)`
An internal memory error occurred.

`#define amErrNoPluginsAllowed (amErrorClass | 0x11)`
Security is set to high: no plug-ins are allowed.

`#define amErrNotFound (amErrorClass | 0x0C)`
The named resource was not found.

`#define amErrNotImplemented (amErrorClass | 0x01)`
The plug-in does not implement the requested function.

`#define amErrOutOfMemory (amErrorClass | 0x05)`
There was insufficient memory to complete the requested operation.

`#define amErrResourceNotFound (amErrorClass | 0x0E)`
An Authentication Manager resource or plug-in is missing.

`#define amErrTokenDestroyed (amErrorClass | 0x12)`
The token was destroyed during the action.

`#define amErrTokenExists (amErrorClass | 0x13)`
The named token already exists.

`#define amErrUnsupportedTokenType (amErrorClass | 0x07)`
The requested token type is unsupported.

`#define amErrUserCancel (amErrorClass | 0x0F)`
The user cancelled the action (such as password gathering).

## AmAuthenticationEnum Enum

**Purpose** An enumeration of the different types of authentication situations. When calling `AmAuthenticateToken()` you can pass a situation so that the plug-in will have an idea of the type of UI to put up for the user.

**Declared In** `Am.h`

**Constants**

`AmAuthenticationNone = 0`
System use only.

`AmAuthenticationOther`
Other authentication.

`AmAuthenticationDataAccess`
Database access.

`AmAuthenticationDeviceUnlock`
The device is being unlocked.

`AmAuthenticationTokenModify`
The token is being modified. For instance, the password is changing.

## AmTokenCacheSettings Enum

**Purpose** Different policies that can be applied to a token in the system. The application creating the token defines the cache settings.

**Declared In** `Am.h`

**Constants** `AmTokenCacheNever`
The token is not cached.

`AmTokenCacheSystem`
The token is kept in the system's token cache.

## AmTokenEnum Enum

**Purpose** An enumeration of the different types of tokens that can be requested from the system.

**Declared In** `Am.h`

**Constants** `AmTokenUnknown = 0`
The token type is unknown.

`AmTokenCustom`
A custom token. See the Comments section, below, for more on custom tokens.

`AmTokenPassword`
A password token.

`AmTokenSignedCode`
A signed code token.

`AmTokenCodeFingerprint`
A code-fingerprint (hash) token.

**Comments** These are the most common type of tokens that the device will deal with. The custom type allows the plug-in to announce a custom type of plug-in that it will service. If an application requests a custom type token, the Authentication Manager examines all plug-ins and find all that match that custom type. Out of all the matches the best fit is picked to create the token.

### AmTokenStrength Enum

**Purpose** Token strengths: the minimum level of strength that a plug-in supports for token creation.

**Declared In** `Am.h`

**Constants** `AmTokenStrengthLow`
Lowest level. No requirements for token creation.

`AmTokenStrengthMedium`
Some measures are taken to reject weak tokens.

`AmTokenStrengthHigh`
The generated token should be guaranteed to not be a weak token.

**Comments** A weak token is an authentication token that can be easily guessed or broken, such as dictionary words for passwords, or weak cryptography keys.

## Authentication Manager Functions and Macros

### AmAuthenticateToken Function

**Purpose** Authenticates a token.

**Declared In** `Am.h`

**Prototype**
```
status_t AmAuthenticateToken (AmTokenType token,
    AmApplicationCtxPtr pAppCtx,
    AmAuthenticationEnum authType,
    char *titleString, char *descriptionString)
```

**Parameters** → *token*
The token to be authenticated.

↔ *pAppCtx*
A pointer to the application context information. The application context contains data that the application wishes to pass to the plug-in. The plug-in may also return data in this structure.

→ *authType*
A hint to the Authentication Manager about why the token is being authenticated—one of the `AmAuthenticationEnum` values. This hint will be passed onto the plug-in that handles this type of token. The plug-in is free to use the hint as it sees fit. The hint is useful when the plug-in displays UI to the user.

→ *titleString*
An optional string that is passed into the plug-in. The plug-in may choose to display the string to the user if it is interactive. The string is meant to be a reason for the authentication request. Pass `NULL` if the plug-in doesn't display a string or if you don't want one displayed.

The plug-in will typically display this string on the title of the modal window for authentication. Accordingly, the string must fit in the title of the window.

→ *descriptionString*
An optional string that is passed into the plug-in. The plug-in may choose to display the string to the user if it is interactive. The string is meant to be a more in-depth description of the reason for the authentication request. Pass `NULL` if the plug-in doesn't display a string or if you don't want one displayed.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrInvalidToken`
The token reference is invalid.

`AmErrAuthenticationFailed`
The authentication failed.

`AmErrOutOfMemory`
The Authentication Manager ran out of memory while attempting to authenticate the token.

**Comments** The Authentication Manager invokes the plug-in to gather a new token, and then calls the plug-in to compare the new token with the supplied token. If the plug-in decides that they are a match, the authentication is successful.

## AmCreateToken Function

**Purpose** Create a new token.

**Declared In** `Am.h`

**Prototype**
```
status_t AmCreateToken (AmTokenPtr pToken,
    char *pSystemId, char *pFriendlyName,
    AmTokenPropertiesPtr pProperties,
    AmApplicationCtxPtr pAppCtx,
    Boolean systemToken)
```

**Parameters**

← *pToken*
Pointer to an `AmTokenType` variable that receives the reference to the newly-created token. If an error occurs, `*pToken` is set to `amInvalidToken`.

→ *pSystemId*
Optional system name for this token. This is a system unique name; applications can later get a reference to this token by looking up this ID. If this argument is set to `NULL`, a system ID will be assigned by the Authentication Manager to this token.

The name is copied into system space, so it may be de-allocated by the caller as soon as this function returns.

→ *pFriendlyName*
Optional pointer to a string buffer containing the "friendly" name that will be associated with this token. This is a name that can be displayed to the user.

→ *pProperties*
A description of the parameters the new token should meet. The system will attempt to meet all requirements, but this is not guaranteed. The only parameter that must be met is the type of token.

↔ *pAppCtx*
A pointer to the application context information. The application context contains data that the application wishes to pass to the plug-in. The plug-in may also return data in this structure.

→ *systemToken*
`true` if this token should be marked as a system token. The only difference between a non-system token and a system token is that when they are destroyed, the reference to the

token is not invalidated. The Authentication Manager will keep the token's reference valid, but the token will be empty after being destroyed. The notification about the token being destroyed is still sent to the Authorization Manager.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrInvalidParam`
One of the input parameters is invalid (most likely a `NULL` *`pToken`*).

`AmErrSystemIDInUse`
The specified token system ID is already in use.

`AmErrSystemIDTooLong`
The name was larger than allowed. The token was not created.

`AmErrOutOfMemory`
The Authentication Manager ran out of memory

`AmErrUnsupportedTokenType`
The specified token type is not supported

**Comments** The token property parameter is used by the caller to describe the desired properties of the new token. The system will find the plug-in that best meets these requirements, but does not guarantee that all (or any) requirements will be met. Once a token has been created, the caller may call `AmGetTokenInfo()` to get the properties of the created token.

**See Also** `AmDestroyToken()`, `AmModifyToken()`

## AmDestroyToken Function

**Purpose** Frees all resources associated with a specified token.

**Declared In** `Am.h`

**Prototype**
```
status_t AmDestroyToken (AmTokenType token,
   AmApplicationCtxPtr pAppCtx)
```

**Parameters** → *`token`*
Reference to the token to destroy.

↔ *`pAppCtx`*
A pointer to the application context information. The application context contains data that the application wishes to pass to the plug-in. The plug-in may also return data in this structure.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrInvalidParam`
One of the input parameters is invalid.

`AmErrInvalidToken`
The referenced token does not exist.

`AmErrTokenDestructionRejected`
Destruction of the token failed, either due to the user rejecting it, or the plug-in vetoing the destruction, or the authentication of the protecting token failing.

**Comments** Call this function when you would like to remove the token from the system, but cannot authenticate against it prior to its deletion. (That is, the token is lost).

The Authentication Manager will verify if there are other tokens protecting the destruction of this token. If there are, those tokens must be authenticated prior to destruction of the specified token.

The Authentication Manager will verify if the action is allowed with the plug-in. If it is allowed then the token will be removed, *along with all the data that it protected*. Care must be taken when a plug-in allows destruction of tokens, as the deletion of certain data objects may leave the system in a non-useful state.

Any application may call this function when a token is lost, even if they did not create the token. (All data protected by this token is deleted though, so this doesn't introduce a security loophole). The Authentication Manager will display a dialog informing the user that the destruction of the token will lead to possible data loss.

When the token is destroyed, a notification is broadcast throughout the system about the token being destroyed.

**See Also** `AmCreateToken()`, `AmModifyToken()`

## AmGetPluginInfo Function

**Purpose** Get the public info block for a plug-in.

**Declared In** `Am.h`

**Prototype**
```
status_t AmGetPluginInfo (AmPluginType plugin,
   AmPluginInfoPtr pInfo)
```

**Parameters**

→ *plugin*
: `AmPluginType` that references the plug-in for which you want information.

← *pInfo*
: Pointer to an `AmPluginInfoType` structure, allocated by the caller, that is filled in by this function.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrInvalidParam`
: *pInfo* is `NULL`.

`AmErrInvalidPlugin`
: The reference to the plug-in is invalid.

**Comments** The `AmPluginInfoType` data structure contains information about the plug-in, such as vendor name, friendly name, and information about the type of tokens that the plug-in can create.

## AmGetPluginReferences Function

**Purpose** Get references to all of the plug-ins registered on the system.

**Declared In** `Am.h`

**Prototype**
```
status_t AmGetPluginReferences
   (AmPluginType *refList, uint16_t *pSize)
```

**Parameters**

← *refList*
: A caller-allocated array where references to the installed plug-ins are returned, or `NULL` to determine how large the array should be. Each array element is an `AmPluginType` that references a single plug-in.

↔ *pSize*
The number of elements in the *refList* array. If *refList* is `NULL`, or if the supplied array isn't large enough, this function sets **pSize* to the required array size.

**Returns** Returns `errNone` if the operation completed successfully, or `AmErrBufferTooSmall` if the supplied buffer is too small.

**Comments** When calling this function you must pre-allocate an array of references and pass the address of this array in *refList*. If the buffer is too small or if *refList* is `NULL`, the *pSize* parameter is set to the required number of entries in the array. Accordingly, you may want to call this function twice. First, call it with *refList* set to `NULL` in order to obtain the size of the needed buffer. Then, after allocating a buffer of the proper size, call this function again to obtain the plug in references.

**See Also** `AmGetPluginInfo()`

## AmGetTokenBySystemId Function

**Purpose** Find a token reference given its system ID.

**Declared In** `Am.h`

**Prototype**
```
status_t AmGetTokenBySystemId (AmTokenPtr pToken,
    char *pSystemId)
```

**Parameters**

← *pToken*
Pointer to an `AmTokenType` variable that receives the reference to the token.

→ *pSystemId*
The token's system ID.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrSystemIDUnknown`
The requested token was not found

`AmErrInvalidParameter`
*pToken* is `NULL`

**Comments** This function allows for the creation of "well known" tokens, such as the system password, and the admin password on a device.

Applications that wish to protect data with the system password can get a reference to it by invoking this function. See "Well-Known Tokens" on page 92 for the predefined set of well-known tokens.

## AmGetTokenExtendedInfo Function

**Purpose** Get extra information about this token.

**Declared In** `Am.h`

**Prototype**
```
status_t AmGetTokenExtendedInfo
   (AmTokenType token, uint8_t *pExtInfo,
   uint32_t *pExtInfolen)
```

**Parameters**

→ *token*
A reference to the token about which information is needed.

← *pExtInfo*
Pointer to the buffer into which the information is written, or `NULL` to determine how large the buffer should be.

← *pExtInfolen*
The size of the *pExtInfo* buffer. If *pExtInfo* is `NULL` or if this parameter's value indicates that the *pExtInfo* buffer isn't large enough to hold the information to be returned, the size of the needed buffer is written to **pExtInfolen*.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrInvalidParam`
*pExtInfo* is NULL.

`AmErrInvalidToken`
The referenced token is invalid.

`AmErrBufferTooSmall`
*pExtInfo* is too small to hold the extended information. The correct size is returned in **pExtInfolen*.

`AmErrNotSupported`
The underlying plug-in doesn't provide any extra information about the token.

**Comments** The plug-in responds to this call directly. For PKI tokens managed by the Palm OS PKI plug-in, the returned data is the certificate ID of the token (in an `AmPluginSignedCodeExtInfoType` structure).

For tokens managed by the Palm OS Code Fingerprint plug-in, the returned data contains the type, creator, and name of the database that was fingerprinted (in an `AmPluginCodePrintExtInfoType` structure). For password tokens managed by the Palm OS password plug-in, the returned data is the hint. No extended information is available for code-fingerprint tokens managed by the Palm OS code-fingerprint plug-in.

**See Also** `AmGetTokenInfo()`

## AmGetTokenInfo Function

**Purpose** Get the public info block for the referenced token.

**Declared In** `Am.h`

**Prototype**

```
status_t AmGetTokenInfo (AmTokenType token,
    AmTokenInfoPtr pInfo)
```

**Parameters**

→ *token*
A reference to the token about which information is needed.

← *pInfo*
Pointer to a location into which the contents of the token's public info block are written. This should be a `AmTokenInfoType` structure.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrInvalidParam`
*pInfo* is `NULL`.

`AmErrInvalidToken`
The referenced token is invalid.

**Comments** Applications call this function after creating a token to examine the properties of the generated token.

**See Also** `AmCreateToken()`, `AmGetTokenExtendedInfo()`

## AmModifyToken Function

**Purpose** Replace or modify an existing token.

**Declared In** `Am.h`

**Prototype**
```
status_t AmModifyToken (AmTokenType token,
   AmTokenPropertiesPtr pProperties,
   AmApplicationCtxPtr pAppCtxOld,
   AmApplicationCtxPtr pAppCtxNew)
```

**Parameters**

→ *token*
Reference to the token to be modified. The new token replaces the old token, using the same reference.

→ *pProperties*
An optional description of the parameters the new token should meet. The system will attempt to meet all requirements, but that is not guaranteed. The only parameter that must be met is the type of token.

If no new properties are specified (pass `NULL` for this parameter), the properties of the token being modified are not changed.

→ *pAppCtxOld*
A pointer to the application context information as it relates the token being modified. The caller may place information required to authenticate the token—such as a password, for a password token—in the application context.

→ *pAppCtxNew*
A pointer to the application context information for the new token that will be created to replace the token being modified, or `NULL`. The caller may place information needed to create the token in this optional parameter.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrInvalidParam`
One of the input parameters is invalid (most likely a `NULL` *pToken*).

`AmErrInvalidToken`
The specified token was not found.

`AmErrOutOfMemory`
The Authentication Manager ran out of memory

`AmErrUnsupportedTokenType`
Token type is not supported

`AmErrModifyRejected`
The modification action was rejected: authentication of the token protecting the to-be-modified token must have failed.

**Comments** Applications wishing to change tokens, (for instance, to change the password, clear the password, and so on) use this function. This function can be used to replace or even remove (by making the token empty) an authentication token. The token being modified is destroyed and a new token is created which takes its place.

The actual operation of this function depends on the plug-in.

If the token properties are specified, and a different token type is requested, then the replacement token that is created will be of a different type. (For example, you may replace a password token with a biometric token). This means that the authentication model is changed.

Tokens may be protected from modification by authentication contexts. With the help of the Authorization Manager the Authentication Manager authenticates the user prior to modification of the token.

After authentication for modify, the Authentication Manager creates a new token, given the properties supplied in *`pProperties`*. The new token will replace the old token.

**See Also** `AmCreateToken()`, `AmDestroyToken()`

## AmRegisterPlugin Function

**Purpose** Register a plug-in in the Authentication Manager.

**Declared In** `Am.h`

**Prototype**
```
status_t AmRegisterPlugin (uint32_t creator,
    Boolean force)
```

**Parameters** → *`creator`*
The creator ID of the plug-in being registered.

→ *force*
: `true` to force the registration, even if the plug-in has already been loaded.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrOutOfMemory`
: Out of memory.

`AmErrAlreadyRegistered`
: The plug-in was already registered and *force* is set to `false`.

**Comments** Note that although you specify the plug-in's creator, you don't specify its type. The plug-in type is implicit: all plug-ins are of type `'ampl'`.

The plug-in's shared library need not be loaded prior to registration. This function loads and opens the shared library. If you set *force* to `true`, forcing a re-registration, the shared library is closed and unloaded. Then, it is reloaded and reopened. The reference to the plug-in doesn't change: it is re-used. Thus, all tokens still have a valid reference to their creator.

**See Also** `AmRemovePlugin()`

## AmRemovePlugin Function

**Purpose** Remove a registered plug-in.

**Declared In** `Am.h`

**Prototype** `status_t AmRemovePlugin (uint32_t creator)`

**Parameters** → *creator*
: The creator ID of the plug-in being removed.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AmErrNotFound`
: A plug-in with the specified creator ID has not been registered.

`AmErrHasTokens`
Tokens exist for this plug-in, and removing the plug-in would invalidate them.

**Comments** In order for this function to work, there must not be any tokens with references back to this plug-in.

**See Also** `AmRegisterPlugin()`

# 4

# AmPlugin

AmPlugin.h declares a set of APIs that are used by Authentication Manager plug-ins to implement a particular authentication model.. These plug-ins are shared libraries of type 'ampl'. After initialization, which is done by sending the plug-in standard launch codes, the Authentication Manager interacts with functions exported by the plug-in through the AmPluginFunctionsType structure.

The contents of this chapter are divided into the following sections:



The header file AmPlugin.h declares the API that this chapter describes.

For information on writing Authentication Manager plug-ins, see "Creating an Authentication Manager Plug-In" on page 26. Information on registering a plug-in with the Authentication Manager or removing a registered plug-in can be found under "Manipulating Authentication Manager Plug-Ins" on page 37.

## AmPlugin Structures and Types

### AmMemHandle Typedef

**Purpose** Handle to a memory chunk allocated in the vault.

**Declared In** AmPlugin.h

**Prototype** `typedef uint32_t AmMemHandle`

**Comments** You must use the AmMemHandle... functions to manipulate Authentication Memory handles. All dynamic memory used by a plug-in that is associated with a token should be done through a call to one of the `AmMemHandle...` functions.

## AmPluginFunctionsType Struct

**Purpose** Identifies those functions in the set that makes up the interface between the Authentication Manager and a plug-in that are implemented by the plug-in.

**Declared In** `AmPlugin.h`

**Prototype**

```
typedef struct {
   status_t (*pluginCaptureFtn)(AmCallMode mode,
      AmApplicationCtxType *pAppContext,
      AmTokenPrivType *pPrivToken,
      AmAuthenticationEnum authType,
      char *title, char *description);
   status_t (*pluginMatchFtn)
      (AmApplicationCtxType *pAppContext,
      AmTokenPrivType *pToken1,
      AmTokenPrivType *pToken2);
   status_t (*pluginDestroyNotifyFtn)
      (AmTokenPrivType *pPrivToken);
   status_t (*pluginGetTokenExtendedInfoFtn)
      (AmTokenPrivType *pPrivToken,
      uint8_t *pExtInfo, uint32_t extInfoLen);
   status_t (*pluginImportTokenFtn)
      (AmTokenPrivType *pPrivToken,
      uint8_t *pBuffer, uint32_t bufferLen);
   status_t (*pluginExportTokenFtn)
      (AmTokenPrivType *pPrivToken,
      uint8_t *pBuffer, uint32_t *bufferLen);
   status_t (*pluginGetDerivedData)
      (AmTokenPrivType *pPrivToken,
      uint8_t *data, uint32_t *dataLength);
   status_t (*pluginAdminFtn)
      (AmPluginType *plugin);
} AmPluginFunctionsType
```

**Fields**

`pluginCaptureFtn`
: The Capture function is called whenever the AM needs your plug-in to create a new token, or to verify or replace an

existing token created by the plug-in. Parameters passed to this function are:

*mode*
: An `AmCallMode` that indicates whether the plug-in should create, verify, or replace a token.

*pAppContext*
: An `AmApplicationCtxType` structure that holds information about the application that needs to be authenticated and other private data specific to the plug-in.

*pPrivToken*
: An `AmTokenPrivType` data structure that holds information regarding the credentials that must be matched for a valid authentication to occur.

*authType*
: A `AmAuthenticationEnum` value that identifies the authentication situation—the reason that the plug-in is being called.

*title*
: An optional string that the plug-in may choose to display to the user. The string is meant to be a reason for the authentication request. This parameter is `NULL` if the caller doesn't want a string to be displayed.

*description*
: An optional string that the plug-in may choose to display to the user. The string is meant to be a more in-depth description of the reason for the authentication request. This parameter is `NULL` if the caller doesn't want such a string to be displayed.

`pluginMatchFtn`
: When the AM needs to verify a token it invokes the associated plug-in's Match entry point, passing in two token structures for comparision. Parameters passed to this function are:

*pAppContext*
: An `AmApplicationCtxType` structure that holds information about the application that needs to be

authenticated and other private data specific to the plug-in.

*pToken1*
An `AmTokenPrivType` data structure that holds information regarding the credentials for the first token being matched.

*pPToken2*
An `AmTokenPrivType` data structure that holds information regarding the credentials for the second token being matched.

`pluginDestroyNotifyFtn`
The Destroy Notify entry point is called when a token is destroyed. Destroying a token is an action that may be taken if the user has lost the ability to authenticate against that token (as in the case of a lost password). This function takes a single parameter:

*pPrivToken*
An `AmTokenPrivType` data structure that holds information about the token being destroyed.

`pluginGetTokenExtendedInfoFtn`
The Get Extended Info function is used to answer the query for extended info by an application. A plug-in is not required to support this entry point, though it can be useful for certain types of tokens. The Palm OS PKI plug-in returns the certificate ID of the token (in an `AmPluginSignedCodeExtInfoType` structure). The Palm OS Code Fingerprint plug-in returns the type, creator, and name of the database that was fingerprinted (in an `AmPluginCodePrintExtInfoType` structure). The Palm OS password plug-in, however, doesn't implement this function. Parameters passed to this function are:

*pPrivToken*
An `AmTokenPrivType` data structure that holds information about the token for which extended information is being requested.

*pExtInfo*
Pointer to the buffer into which the information is to be written.

*extInfoLen*
The size, in bytes, of the *pExtInfo* buffer.

**pluginImportTokenFtn**
The Import and Export entry points are for copying internal data about a token for import or export. In your import function, the contents of the provided buffer should be copied to memory that is associated with a token *and is managed by the plug-in*. Parameters passed to this function are:

*pPrivToken*
An AmTokenPrivType data structure that holds information about the token being imported.

*pBuffer*
Pointer to the buffer containing the token info being imported.

*bufferLen*
The size, in bytes, of the *pBuffer* buffer.

**pluginExportTokenFtn**
The Import and Export entry points are for copying internal data about a token for import or export. In your export function, memory that is associated with a token *and is managed by the plug-in* should be copied to the provided buffer and returned to the AM. Parameters passed to this function are:

*pPrivToken*
An AmTokenPrivType data structure that holds information about the token being exported.

*pBuffer*
Pointer to the buffer into which the token info should be written.

*bufferLen*
The size, in bytes, of the *pBuffer* buffer.

**pluginGetDerivedData**
The Get Derived Data function is used solely by the operating system to get seed data for a cryptographic key derived from an authentication token (such as password derived keys). Currently the only user of this feature is the Data Manager; it uses this feature to generate password-

derived keys for the backup function. Parameters passed to this function are:

*pPrivToken*
An AmTokenPrivType data structure that holds information about the token for which derived data is being requested.

*data*
Buffer into which your function should write seed data derived from the authentication token.

*dataLength*
Size, in bytes, of the *data* buffer.

`pluginAdminFtn`
This function is the admin entry point for the plug-in. Some plug-ins may have settings that can be changed (a biometric plug-in, for instance, might allow the user to tweak the settings it uses to match tokens); accordingly, the plug-in can implement an admin UI in its implementation of this function. Parameters passed to this function are:

*plugin*
A reference to the plug-in.

**Comments** These functions may return any error defined in `Am.h`.

Upon receipt of a `sysAppLaunchCmdNormalLaunch` launch code, the plug-in should, among other things, identify which of these functions it implements. Accompanying the launch code is a `AmTokenPrivType` structure; this structure's *ftn* field is an `AmPluginFunctionsType` structure into which you write the addresses of those functions that your plug-in implements.

## AmPluginPrivType Struct

**Purpose** Describes a plug-in. This structure is used by the AM to reference a plug-in loaded onto the system.

**Declared In** `AmPlugin.h`

**Prototype**

```
typedef struct {
   AmPluginType pluginRef;
   uint32_t recordId;
   AmPluginInfoType info;
   uint32_t tokenDataLength;
   uint32_t tokenExtendedInfoLength;
   AmPluginFunctionsType ftn;
} AmPluginPrivType, *AmPluginPrivPtr
```

**Fields**

`pluginRef`
Reference to this plug-in. This field is set by the AM.

`recordId`
Record ID for this plug-in in the AM vault. This field is set by the AM.

`info`
An `AmPluginInfoType` structure that contains information about the plug-in to be shared with applications.

`tokenDataLength`
The size, in bytes, of the tokens managed by this plug-in.

`tokenExtendedInfoLength`
The length, in bytes, of any extended information that is returned to caller by the Get Extended Info entry point.

`ftn`
A list of plug-in entry points. Not all entry point function pointers need to be initialized: only those entry points that a plug-in chooses to implement need to be set in this structure. See `AmPluginFunctionsType` for the complete set of entry points a plug-in can export.

**Comments** An Authentication Manager plug-in fills in a data structure of this type during initialization. Within this data structure are entry points that provide the Authentication Manager with direct access to the plug-in functions.

See the discussion of the `sysAppLaunchCmdNormalLaunch` launch code under "Open and Close" on page 28 for more information on filling out this structure's fields.

## AmTokenDataType Struct

**Purpose** An optional linked list of record IDs that identify dynamic data associated with the token.

**Declared In** `AmPlugin.h`

**Prototype**

```
typedef struct _AmTokenDataTag {
   uint32_t recordId;
   struct _AmTokenDataTag *next;
} AmTokenDataType, *AmTokenDataPtr
```

**Fields**

`recordId`
: ID of the record that contains the dynamic data.

`next`
: Pointer to the next `AmTokenDataType` structure in the linked list, or `NULL` if there is no additional data.

## AmTokenPrivType Struct

**Purpose** An opaque (to AM clients) data structure that holds information regarding the credentials that must be matched for a valid authentication to occur.

**Declared In** `AmPlugin.h`

**Prototype**

```
typedef struct {
   AmTokenInfoType header;
   uint32_t pluginCreatorId;
   uint32_t tokenRecId;
   MemPtr dataPtr;
   uint32_t dataLength;
   AmTokenDataPtr dynamicData;
} AmTokenPrivType, *AmTokenPrivPtr
```

**Fields**

`header`
: Public information common to all tokens. See `AmTokenInfoType`.

`pluginCreatorId`
: The creator ID that manages this token. This is the creator ID of the plug-in.

`tokenRecId`
: Reference to the record ID for this token in the AM vault.

`dataPtr`
: Pointer to the token's data segment.

`dataLength`
: Length, in bytes, of the token't data segment.

`dynamicData`
: Pointer to an `AmTokenDataType` structure that holds dynamic data associated with the token.

**Comments** The creation and management of tokens (these structures) is left up to the authentication plug-in.

# AmPlugin Constants

## AmCallMode Enum

**Purpose** Indicates whether the `pluginCaptureFtn()` entry point function is being called to create new credentials, verify existing credentials, or replace existing credentials.

**Declared In** `AmPlugin.h`

**Constants**

`AmEnrollment = 0`
: The plug-in should create new credentials.

`AmVerification`
: The plug-in should verify credentials.

`AmReplacementStart`
: First phase of replacing a token. See the Comments section, below.

`AmReplacementEnd`
: Second phase of replacing a token. See the Comments section, below.

**Comments** The replacement mode is used when a credential is being modified. There are two stages. `ReplacementStart` is used first. In this stage the plug-in must authenticate whoever is trying to modify the token, be it a user or an application. `ReplacementEnd` is used last; this is when the plug-in creates the new replacement credentials.

# AmPlugin Functions and Macros

## AmInitializeUIContext Function

**Purpose** Locks the UI context and grabs the event queue from the calling thread.

**Declared In** `AmPlugin.h`

**Prototype** `status_t AmInitializeUIContext (void)`

**Parameters** None.

**Returns** Always returns `errNone`.

**Comments** If your plugin needs to interact with the user, at the point where you need to display some form of UI it should call `WinStartThreadUI()`, then it should call this function.

If the UI context is currently locked when this function is called, this function blocks until it is released.

**See Also** `AmReleaseUIContext()`

## AmMemHandleFree Function

**Purpose** Deallocates a dynamically-created memory chunk and disassociates it from the associated token.

**Declared In** `AmPlugin.h`

**Prototype** `void AmMemHandleFree (AmTokenPrivType *pPrivToken, AmMemHandle hMem)`

**Parameters**

→ *pPrivToken*
Pointer to the token with which the memory chunk is associated.

→ *hMem*
Pointer to the memory chunk to be freed. This memory chunk must have been allocated with `AmMemHandleNew()`.

**Returns** Nothing.

**Comments** The memory chunk being freed should not be locked when this function is called.

**See Also** `AmMemHandleNew()`, `MemHandleFree()`

## AmMemHandleLock Function

**Purpose** Obtain a pointer to a chunk of memory referenced by an `AmMemHandle`.

**Declared In** `AmPlugin.h`

**Prototype** `MemPtr AmMemHandleLock (AmMemHandle hMem)`

**Parameters** → *hMem*
Handle to the memory chunk. This handle must have been returned from `AmMemHandleNew()`.

**Returns** A pointer to the memory chunk, or `NULL` if the chunk couldn't be locked.

**Comments** This function does not return a pointer to the actual memory chunk in the vault. Instead, an ordinary memory buffer of the corresponding size is allocated using `MemPtrNew()` and the contents of the vault chunk are copied into it. The pointer to this new buffer is returned. When `AmMemHandleUnlock()` is called, the buffer's contents are copied back to the vault and the buffer allocated during the call to `AmMemHandleLock()` is freed.

**See Also** `AmMemHandleNew()`, `AmMemHandleUnlock()`, `MemHandleLock()`

## AmMemHandleNew Function

**Purpose** Allocates a memory chunk of a specified size in the vault and associates it with a specified token.

**Declared In** `AmPlugin.h`

**Prototype**
```
AmMemHandle AmMemHandleNew
    (AmTokenPrivType *pPrivToken, uint32_t size)
```

**Parameters** → *pPrivToken*
Pointer to the token with which the record is to be associated.

→ *size*
Size, in bytes, of the memory chunk to be allocated.

**Returns** Returns a handle to the newly-allocated memory chunk, or 0 if the chunk could not be allocated as specified.

**Comments** The newly-allocated memory chunk is actually a database record of the specified size.

You must call `AmMemHandleLock()` in order to obtain a pointer to a buffer into which you can write (or from which you can read).

**See Also** `AmMemHandleFree()`, `AmMemHandleLock()`, `MemHandleNew()`

## AmMemHandleUnlock Function

**Purpose** Unlocks a vault memory chunk previous locked with `AmMemHandleLock()`.

**Declared In** `AmPlugin.h`

**Prototype**
```
void AmMemHandleUnlock (AmMemHandle hMem,
   MemPtr pMem)
```

**Parameters** → *hMem*
Handle to the vault memory chunk previously allocated with `AmMemHandleNew()`.

→ *pMem*
Pointer to the memory buffer returned from a call to `AmMemHandleLock()`.

**Returns** Nothing.

**Comments** This function copies the contents of the memory buffer *pMem* to the memory chunk in the vault referenced by *hMem*. It then frees the buffer *pMem*.

**See Also** AmMemHandleLock(), AmMemHandleNew(), MemHandleUnlock()

## AmReleaseUIContext Function

**Purpose** Unlock the UI context, and release the event queue.

**Declared In** `AmPlugin.h`

**Prototype** `status_t AmReleaseUIContext (void)`

**Parameters** None.

**Returns** Always returns `errNone`.

**Comments** As soon as possible after the AM plugin is done interacting with the user, it should call this function and then it should call WinFinishThreadUI().

**See Also** AmInitializeUIContext()

# 5

# AmPluginCodePrint

This data structure is used by the Code Fingerprint plug-in to the Authentication Manager. It provides extra information about the tokens managed by the plug-in.



The header file `AmPluginCodePrint.h` declares the API that this chapter describes.

See Chapter 1, "Palm OS Cobalt Security," on page 3 for more information about the Authentication Manager and AM plug-ins.

## AmPluginCodePrint Structures and Types

### AmPluginCodePrintExtInfoType Struct

**Purpose** Data structure returned by the standard Code Fingerprint plug-in when extended token information is requested by calling `AmGetTokenExtendedInfo()`.

**Declared In** `AmPluginCodePrint.h`

**Prototype**

```
typedef struct {
   uint32_t type;
   uint32_t creator;
   char name[32];
} AmPluginCodePrintExtInfoType
```

**Fields**

`type`
: The type of the database that was fingerprinted.

`creator`
: The creator ID of the database that was fingerprinted.

`name`
: The name of the database that was fingerprinted.

# 6
# AmPluginSignedCode

This data structure is used by the Signed Code plug-in to the Authentication Manager. It provides extra information about the PKI tokens managed by the plug-in.

The header file `AmPluginSignedCode.h` declares the API that this chapter describes.

See Chapter 1, "Palm OS Cobalt Security," on page 3 for more information about the Authentication Manager and AM plug-ins.

## AmPluginSignedCode Structures and Types

### AmPluginSignedCodeExtInfoType Struct

**Purpose** Data structure returned by the standard PKI plug-in when extended token information is requested by calling `AmGetTokenExtendedInfo()`.

**Declared In** `AmPluginSignedCode.h`

**Prototype**

```
typedef struct {
   SignCertificateIDType certID;
} AmPluginSignedCodeExtInfoType
```

**Fields**

`certID`
The certificate ID.

# 7

# Authorization Manager

The Authorization Manager is the top-level component in the security suite that enables protection of objects in the operating system. The Authorization Manager is designed around the Access Control List paradigm.

The Authorization Manager maintains a list of rule-set containers that represent protected objects. Associated with each rule-set container is the protected object's fully-qualified name as defined by the creator of the protected object and one or more rules specifying how the object referenced by the Access Control List container is protected. Each rule is an association between one or more actions that could be performed on the object and the set of authentications required to perform those actions.

The Authorization Manager relies on the managers of objects to provide a fully-qualified name for the objects they want to protect, and to define what actions can be performed on those objects.

The remainder of this chapter documents the Authorization Manager APIs. It is organized into the following sections:



The header file `azm.h` declares the API that this chapter describes.

# Authorization Manager Structures and Types

## AzmActionType Typedef

**Purpose** Defines a bitmap of actions. Each bit in the `AzmActionType` value corresponds to a different rule.

**Declared In** `azm.h`

**Prototype**

```
typedef uint32_t AzmActionType
```

## AzmNotificationType Struct

**Purpose** Data structure that accompanies notifications sent by the Authorization Manager.

**Declared In** `azm.h`

**Prototype**

```
typedef struct {
   union {
      struct {
         uint8_t name [azmRuleSetNameMaxLength];
         uint32_t length;
      } ruleSetDestroyed;
   } data;
   uint16_t version;
   uint16_t padding;
} AzmNotificationType
```

**Fields**

`data`
: A union of structures, one for each of the notifications that can be sent.

  `ruleSetDestroyed`
  : The `data` variant that applies to `AzmNotificationRuleSetDestroyed` notifications. This structure has two fields: the name of the rule-set being destroyed, and the length of that name.

`version`
: The version of the `AzmNotificationType` structure. The structure as detailed above is version 1.

`padding`
Padding bytes.

### AzmRuleSetType Typedef

**Purpose** An opaque handle to an rule-set container managed by the Authorization Manager.

**Declared In** `azm.h`

**Prototype** `typedef uint32_t AzmRuleSetType`

## Authorization Manager Constants

### Miscellaneous Authorization Manager Constants

**Purpose** The header file `azm.h` also declares these constants.

**Declared In** `azm.h`

**Constants** `#define azmActionModify 0x80000000`
Predefined MODIFY action. This action may *not* be redefined. This is an Authorization Manager-specific action which gates the modification of a rule-set container. A modification of a rule-set container is defined as creation (always allowed), addition or modification of ACE entries, or destruction.

`#define azmCreator 'azm_'`
Creator ID used for the vault that contains secure databases.

`#define azmInvalidRuleSet 0xFFFFFFFF`
Rule-set handle value representing an invalid rule-set.

`#define azmMaxTokenNodes 2`
Maximum number of token nodes.

`#define azmMaxTokensInNode 8`
Maximum number of tokens that can be placed into a single node.

`#define azmMaxTokensInTree azmMaxTokenNodes * azmMaxTokensInNode`
The maximum number of tokens that can occur in an access rule.

`#define AzmNotificationRuleSetDestroyed 0x1`
"Rule-set destroyed" notification callback opcode.

`#define azmRuleFormatLength 60`
The maximum length in bytes (including the null terminator) of a rule format string.

`#define azmRuleSetNameMaxLength 20`
The maximum length, in bytes, of an Authorization Manager rule-set name.

`#define AzmServiceName "psysAuthorizationMgr"`
The name under which the Authorization Manager is registered with the Service Manager.

`#define azmSyncRuleSet 0x00800000`
Handle to the rule-set container for synchronization.

## Authorization Manager Error Codes

**Purpose** Error codes returned by the various Authorization Manager functions.

**Declared In** `azm.h`

**Constants**

`#define azmErrAlreadyExists (azmErrorClass | 19)`
The specified rule-set already exists.

`#define azmErrAuthorizationFailed (azmErrorClass | 9)`
The authorization request has failed.

`#define azmErrBackupInProgress (azmErrorClass | 17)`
A backup is in progress.

`#define azmErrBadParam (azmErrorClass | 1)`
One of the supplied parameters is invalid.

`#define azmErrInvalidParameter (azmErrorClass | 18)`
One of the supplied parameters is invalid.

`#define azmErrInvalidReference (azmErrorClass | 8)`
: The reference to the rule-set container is invalid.

`#define azmErrInvalidRuleSyntax (azmErrorClass | 16)`
: The syntax of the supplied rule definition is invalid.

`#define azmErrInvalidTokenReference (azmErrorClass | 15)`
: The token reference is invalid.

`#define azmErrMaxRuleSets (azmErrorClass | 5)`
: The system already has the maximum number of rule sets allowed.

`#define azmErrMemory (azmErrorClass | 6)`
: The Authorization Manager encountered a memory error. This may indicate a possible out-of-memory condition.

`#define azmErrMgrAlreadyRegistered (azmErrorClass | 13)`
: The manager creator ID is already registered.

`#define azmErrMgrNotRegistered (azmErrorClass | 10)`
: A request was recieved from an unregistered manager.

`#define azmErrNotFound (azmErrorClass | 12)`
: The rule-set being looked up was not found.

`#define azmErrNotImplemented (azmErrorClass | 2)`
: The Authorization Manager attempted to perform an unimplemented operation.

`#define azmErrNotOpen (azmErrorClass | 3)`
: The library has not been opened.

`#define azmErrOutOfMemory (azmErrorClass | 7)`
: There is insufficient memory to complete the requested operation.

`#define azmErrRestrictedAPI (azmErrorClass | 11)`
: This call can only be made from a registered manager.

`#define azmErrStillOpen (azmErrorClass | 4)`
: The library is opened by others and cannot be closed.

`#define azmErrTooManyTokensInRule (azmErrorClass | 14)`

You have exceeded the limit on the number of tokens per rule.

# Authorization Manager Functions and Macros

## AzmAddRule Function

**Purpose** Adds an access rule to an existing rule-set container for a specific action bitmap.

**Declared In** `azm.h`

**Prototype**

```
status_t AzmAddRule (AzmRuleSetType ruleset,
   AzmActionType action, char *rulefmt, ...)
```

**Parameters**

→ *`ruleset`*
A valid handle to a rule-set container managed by the Authorization Manager.

→ *`action`*
Bitmap of actions to apply these rules to.

→ *`rulefmt`*
A rule format string of the form: `[%t]+ (OR [%t]*)?`. This string should not exceed `azmRuleFormatLength` bytes in length.

→ *`...`*
A variable argument list containing a valid `AmTokenType` for each "`%t`" in the rule format string.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`azmErrParam`
The rule format string is invalid.

`azmErrMemory`
Out of memory, or the Authorization Manager detected an invalid internal memory structure.

`azmErrInvalidReference`
*`ruleset`* is invalid.

`azmErrAuthorizationFailed`
The modify rule-set rule was not authenticated properly.

`azmErrTooManyTokensInRule`
Your rule has too many tokens. A given rule cannot have more than `azmMaxTokensInTree` tokens.

**Comments** The rule format string is in the canonical form “`%t (OR %t)`”. That is, at least one “`%t`”. This may be followed by “ OR %t” to indicate the right hand side of the ACE rule. In the right hand side of the ACE rule there may be zero or more “%t” references.

There must be a valid `AmTokenType` in the variable argument list for each “%t” in the rule format string.

If the action referenced in the function call already has a rule associated with it, the rule is replaced. See “Schema Database Access Rule Action Types” on page 301 of *Exploring Palm OS: Memory, Databases, Files* for the set of constants that correspond to those schema database actions for which you can set access rules. For an example of how to use this function to secure a schema database, see “Securing Databases” on page 52.

Although this function is usually called by applications, it can be called from anywhere. However, in order for the call to succeed you must pass the rule-set container modify rule. To create a rule that allows access to anyone, use the well-known system token `amEmptyToken` and create a rule that only contains that token for the action you wish to allow free access to. For example,

```
AmGetTokenBySystemId(&token, “empty”);
AzmAddRule(ruleSetRef, action, “%t”,  token);
```

## AzmGetSyncBypass Function

**Purpose** Get the state of the sync bypass flags.

**Declared In** `azm.h`

**Prototype**
```
status_t AzmGetSyncBypass
   (AzmRuleSetType ruleset,
   uint32_t *statebitfield)
```

**Parameters**

→ *ruleset*
A reference to the rule-set container for which the bypass settings are being requested.

← *statebitfield*
A bitmap of the actions. If a bit is set for a specific action then its bypass flag is set.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`azmErrInvalidReference`
*ruleset* is invalid

`azmErrBadParam`
*statebitfield* is `NULL`.

**Comments** Upon return `*`*statebitfield* contains a 1 for each state that has bypass enabled. Checking for a flag is easy. For instance, to see if sync bypass it set for ACTION_READ, simply do the following:

```
if ((stateBitField & ACTION_READ) > 0)
```

Note that it is not possible to set a sync bypass for Authorization Manager actions such as ACTION_MODIFY.

**See Also** `AzmSetSyncBypass()`

## AzmNonInteractiveAuthorize Function

**Purpose** Authorize an action given a rule-set reference.

**Declared In** `azm.h`

**Prototype**
```
status_t AzmNonInteractiveAuthorize
    (AzmRuleSetType ruleSet, AzmActionType action,
    uint32_t appIdentityKeyId)
```

**Parameters**

→ *ruleSet*
: A valid handle to a rule-set container managed by the Authorization Manager.

→ *action*
: A bitmap of the action to authorize. Only one action may be authorized.

→ *appIdentityKeyId*
: The keyID for the application ID key that was passed in to the manager during the action request message.

**Returns** `errNone` if the authorization request succeeds. Otherwise, this function returns one of the following:

`azmErrInvalidReference`
: The reference to the Authorization Manager rule-set container is invalid.

`azmErrParam`
: The action paarameter is empty. That is, it has a value of 0.

`azmErrMemory`
: An internal memory error occurred.

`azmErrAuthorizationFailed`
: Authorization failed.

**Comments** The authorization function treats all interactive tokens as failed authentications (without calling the Authentication Manager for authentication), therefore the result is whether a non-interactive authorization succeeds. This means that the device won't bother the user while this function is in operation.

The rule-set container must have been created prior to calling this function.

## AzmSetSyncBypass Function

**Purpose** Set the state of the sync bypass flags.

**Declared In** `azm.h`

**Prototype**

```
status_t AzmSetSyncBypass
   (AzmRuleSetType ruleset, AzmActionType action,
   Boolean state)
```

**Parameters**

→ *ruleset*
A reference to the rule-set container for which the bypass settings are to be modified.

→ *action*
A bitmap of the actions. If a bit is set for a specific action then its bypass flag is set, otherwise it is cleared.

→ *state*
What state to set the relationship to: `true` corresponds to "allow sync," while `false` corresponds to "disallow sync."

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`AzmErrInvalidReference`
*ruleset* is invalid.

`AzmErrAuthorizationFailed`
Authorization of the rule-set container modify rule failed. You must be the manager that created the rule-set.

**Comments** Every rule-set can have a set of sync-bypass flags associated with it. These bypass flags can be enabled or disabled on a per-action basis.

When sync-bypass is enabled for a specific action, an authenticated sync agent will be able to complete that action successfully. (The sync-bypass rules takes care of authenticating sync agents). For example, if a data manager object (protected database) sets sync bypass for the READ action, a sync agent such as HotSync can access the contents of the database for READ only, thus enabling one-way sync.

Although this function can be called from anywhere, only callers that pass the ACTION_MODIFY rule will succeed in setting a bypass flag.

**See Also** `AzmGetSyncBypass()`

# 8

# Certificate Manager

The Certificate Manager manages digital certificates, handling such operations as import, export, parsing, secure storage, content authentication, and storage querying. You can use the Certificate Manager in two different ways: as a certificate verifier and parser, and as a certificate store. In the verifier/parser mode, the Certificate Manager takes data as input and parses it as a digital certificate. The user can then verify the certificate and access its internal fields. In certificate store mode, the Certificate Manager can securely store a tree of digital certificates (with multiple roots) and make the fields of those certificates available to users.

The Certificate Manager is a system server with a client-side library. To securely store certificates, the Certificate Manager makes use of the Data Manager's vault facilities.

The remainder of this chapter documents the Certificate Manager APIs. It is organized into the following sections:



The header file `CertificateMgr.h` declares the API that this chapter describes.

# Certificate Manager Structures and Types

## CertMgrCertChainType Struct

**Purpose** A certificate chain is used when calling `CertMgrVerifyCert()`. You may define a list of certificates to be used when trying to verify the certificate chain.

**Declared In** `CertificateMgr.h`

**Prototype**

```
typedef struct {
   CertMgrCertInfoType *certs;
   uint32_t count;
} CertMgrCertChainType
```

**Fields**

`certs`
Pointer to the first certificate in the chain.

`count`
Number of certificates in the chain.

## CertMgrCertElementEnum Typedef

**Purpose** Certificate element types, used in conjunction with `CertMgrGetField()`.

**Declared In** `CertificateMgr.h`

**Prototype** `typedef uint32_t CertMgrCertElementEnum`

**Constants**

`#define apCertMgrElementTypeRDN 34`

`#define apCertMgrElementTypeRSA 33`

`#define apCertMgrElementTypeX509Cert 32`

`#define apCertMgrElementTypeX509Extensions 35`

## CertMgrCertFieldEnum Typedef

**Purpose** Certificate element field types, used in conjunction with `CertMgrGetField()`.

**Declared In** `CertificateMgr.h`

**Prototype** `typedef uint32_t CertMgrCertFieldEnum`

**Comments** The values used with variables of this type depend on the certificate element. Depending on the certificate element, the element's field types are listed under one of the following:

- "X509Cert Element Fields" on page 146
- "RSA Element Fields" on page 148
- "RDN Element Fields" on page 148
- "X509Extensions Element Fields" on page 149

## CertMgrCertInfoType Struct

**Purpose** Abstracts a certificate object. An application uses a structure of this type to refer to a certificate in the memory address space of the Certificate Manager. You can get a reference to a certificate from a successful call to `CertMgrImportCert()` or `CertMgrFindCert()`

**Declared In** `CertificateMgr.h`

**Prototype**

```
typedef struct {
   uint32_t ref;
   uint16_t format;
} CertMgrCertInfoType
```

**Fields**

`ref`
: An opaque object.

`format`
: The format of the certificate. See "Certificate Formats" on page 150 for the defined set of certificate format values.

## CertMgrCertSearchEnum Typedef

**Purpose** Specifies the search mode to `CertMgrFindCert()`.

**Declared In** `CertificateMgr.h`

**Prototype** `typedef uint32_t CertMgrCertSearchEnum`

**Constants** `#define apCertMgrSearchCert 1000`

In this mode, repeated calls to `CertMgrFindCert()` iterate through the certificates in the store, each time returning the certificate that resides at the location indicated by the *`index`* parameter. Each time `CertMgrFindCert()` is called, the *`index`* parameter is incremented. Accordingly, you can use this mode to iterate through all of the certificates in the certificate store.

`#define apCertMgrSearchCertID 1001`

Causes `CertMgrFindCert()` to look for a certificate with a certID that matches that supplied in the *`reference`* parameter. The data in *`reference`* should be a 20-byte certID.

`#define apCertMgrSearchSubjectRDN apCertMgrFieldSubjectRDN`

Causes `CertMgrFindCert()` to look for a certificate whose SubjectRDN matches the one in the *`reference`* parameter.

## CertMgrElementListType Struct

**Purpose** Structure that represents a list of elements. `CertMgrGetField()` fills out this structure.

**Declared In** `CertificateMgr.h`

**Prototype**

```
typedef struct {
   uint32_t length;
   uint32_t count;
   CertMgrElementType element[1];
} CertMgrElementListType
```

**Fields** `length`

The length of this structure, including all of the `CertMgrElementType` structures needed to contain all of the list's fields.

`count`
: The number of fields—that is, the number of `CertMgrElementType` structures—in the list.

`element`
: The first element. Subsequent elements follow this one.

## CertMgrElementType Struct

**Purpose** Structure that represents a single field.

**Declared In** `CertificateMgr.h`

**Prototype**

```
typedef struct {
   uint16_t type;
   uint16_t field;
   uint16_t dataType;
   uint16_t length;
   uint32_t offset;
} CertMgrElementType
```

**Fields**

`type`
: The certificate element type. One of the `CertMgrCertElementEnum` values.

`field`
: The element field identifier. One of the values listed under "X509Cert Element Fields," "RSA Element Fields," "RDN Element Fields,"or "X509Extensions Element Fields."

`dataType`
: The field's data type. One of the values listed under "Data Types" on page 149.

`length`
: The length, in bytes, of the field data.

`offset`
: The offset, in bytes, to the beginning of the field data.

## CertMgrVerifyResultType Struct

**Purpose** If a certificate fails to verify during a call to `CertMgrVerifyCert()` or `CertMgrAddCert()`, this structure is filled in and returned to indicate the reason for the verification failure.

**Declared In** `CertificateMgr.h`

**Prototype**

```
typedef struct {
   uint32_t failureCode;
   CertMgrCertInfoType cert;
   uint32_t depth;
   uint32_t state;
   DateTimeType verifyTime;
} CertMgrVerifyResultType
```

**Fields**

`failureCode`
: The reason for the verification failure. See "Certificate Verification Failure Codes" on page 152 for the set of values that can be returned in this field.

`cert`
: The certificate that failed to verify.

`depth`
: How deep the failed certificate is.

`state`
: The verification state.

`verifyTime`
: The date and time against which the certificate was verified.

# Certificate Manager Constants

## X509Cert Element Fields

**Purpose** Fields in an X509Cert element.

**Declared In** `CertificateMgr.h`

**Constants**

```
#define apCertMgrElementFieldEntireCert 17
#define apCertMgrElementFieldExtension 200
#define apCertMgrElementFieldExtensions 23
#define apCertMgrElementFieldInnerDER 1
#define apCertMgrElementFieldIssuerID 15
```

```
#define apCertMgrElementFieldIssuerRDN 4
#define apCertMgrElementFieldIssuerUniqueID 21
#define apCertMgrElementFieldNotAfter 7
#define apCertMgrElementFieldNotBefore 6
#define apCertMgrElementFieldPubKeyBER 8
#define apCertMgrElementFieldSerialNumber 3
#define apCertMgrElementFieldSigAlgID 91
#define apCertMgrElementFieldSignature 14
#define apCertMgrElementFieldSigOID 12
#define apCertMgrElementFieldSigParams 13
#define apCertMgrElementFieldSubjectID 16
#define apCertMgrElementFieldSubjectRDN 5
#define apCertMgrElementFieldSubjectUniqueID 22
#define apCertMgrElementFieldVersion 2
#define apCertMgrFieldExtensions
  apCertMgrElementFieldExtensions
#define apCertMgrFieldIssuerID
  apCertMgrElementFieldIssuerID
#define apCertMgrFieldIssuerRDN
  apCertMgrElementFieldIssuerRDN
#define apCertMgrFieldIssuerUniqueID
  apCertMgrElementFieldIssuerUniqueID
#define apCertMgrFieldNotAfter
  apCertMgrElementFieldNotAfter
#define apCertMgrFieldNotBefore
  apCertMgrElementFieldNotBefore
#define apCertMgrFieldPubKeyBER
  apCertMgrElementFieldPubKeyBER
#define apCertMgrFieldSerialNumber
  apCertMgrElementFieldSerialNumber
```

```
#define apCertMgrFieldSignature
  apCertMgrElementFieldSignature
```

```
#define apCertMgrFieldSigOID
  apCertMgrElementFieldSigOID
```

```
#define apCertMgrFieldSigParams
  apCertMgrElementFieldSigParams
```

```
#define apCertMgrFieldSubjectID
  apCertMgrElementFieldSubjectID
```

```
#define apCertMgrFieldSubjectRDN
  apCertMgrElementFieldSubjectRDN
```

```
#define apCertMgrFieldSubjectUniqueID
  apCertMgrElementFieldSubjectUniqueID
```

```
#define apCertMgrFieldVersion
  apCertMgrElementFieldVersion
```

## RSA Element Fields

**Purpose** Fields in an RSA element.

**Declared In** `CertificateMgr.h`

**Constants**

```
#define apCertMgrElementFieldRSAModulus 16
```

```
#define apCertMgrElementFieldRSAPubExpo 17
```

## RDN Element Fields

**Purpose** Fields in an RDN element.

**Declared In** `CertificateMgr.h`

**Constants**

```
#define apCertMgrElementFieldRDNOID 4
```

First RDN OID.

```
#define apCertMgrElementFieldRDNValue 5
```

First RDN value.

**Comments** To get the second and subsequent OIDs and values, use the `apCertMgrElementFieldRDNOIDN()` and `apCertMgrElementFieldRDNValueN()` macros, respectively.

## X509Extensions Element Fields

**Purpose** Fields in an X509Extensions element.

**Declared In** `CertificateMgr.h`

**Constants**

`#define apCertMgrElementFieldX509ExBytes 2`
First data bytes.

`#define apCertMgrElementFieldX509ExCritical 1`
First critical flag.

`#define apCertMgrElementFieldX509ExOID 0`
First extension OID.

**Comments** To get the second and subsequent data bytes, critical flags, and extension OIDs, use the `apCertMgrElementFieldX509ExBytesN()`, `apCertMgrElementFieldX509ExCriticalN()`, and `apCertMgrElementFieldX509ExOIDN()` macros, respectively.

## Data Types

**Purpose** Data type of an element field. These values appear in the `CertMgrElementType` data structure.

**Declared In** `CertificateMgr.h`

**Constants**

`#define apCertMgrElementDataTypeASN1BitString 3`

`#define apCertMgrElementDataTypeASN1BmpString 30`

`#define apCertMgrElementDataTypeASN1Boolean 1`

`#define apCertMgrElementDataTypeASN1EmbeddedPDV 11`

`#define apCertMgrElementDataTypeASN1Enumerated 10`

`#define apCertMgrElementDataTypeASN1Eoc 0`

`#define apCertMgrElementDataTypeASN1External 8`

`#define apCertMgrElementDataTypeASN1GenString 27`

`#define apCertMgrElementDataTypeASN1GenTime 24`

`#define apCertMgrElementDataTypeASN1GraphicString 25`

`#define apCertMgrElementDataTypeASN1IA5String 22`

```
#define apCertMgrElementDataTypeASN1Integer 2
#define apCertMgrElementDataTypeASN1ISO64String 26
#define apCertMgrElementDataTypeASN1Null 5
#define apCertMgrElementDataTypeASN1NumericString
  18
#define apCertMgrElementDataTypeASN1ObjDesc 7
#define apCertMgrElementDataTypeASN1OctetString 4
#define apCertMgrElementDataTypeASN1OID 6
#define apCertMgrElementDataTypeASN1PrintString 19
#define apCertMgrElementDataTypeASN1Real 9
#define apCertMgrElementDataTypeASN1Sequence 16
#define apCertMgrElementDataTypeASN1Set 17
#define apCertMgrElementDataTypeASN1T61String 20
#define apCertMgrElementDataTypeASN1UnivString 28
#define apCertMgrElementDataTypeASN1UTCTime 23
#define apCertMgrElementDataTypeASN1UTF8String 12
#define apCertMgrElementDataTypeASN1VideoTexString
  21
```

## Certificate Formats

**Purpose** Specifies the certificate format. The `CertMgrCertInfoType` structure's `format` field takes one of these values.

**Declared In** `CertificateMgr.h`

**Constants** `#define apCertMgrFormatX509 1`
The certificate is a DER encoded x509 certificate.

`#define apCertMgrFormatXML 2`
The certificate is formatted as XML.

**NOTE:** XML-formatted certificates are not currently supported in Palm OS Cobalt.

## Certificate Manager Error Codes

**Purpose** Error codes returned by the various Certificate Manager functions.

**Declared In** `CertificateMgr.h`

**Constants**

`#define certMgrErrBackupInProgress (certErrorClass | 0x0C)`
The certificate vault could not be accessed because it is in the process of being backed up.

`#define certMgrErrBufTooSmall (certErrorClass | 0x07)`
The export buffer is too small. The required size is written into the variable pointed to by the length parameter.

`#define certMgrErrCertNotFound (certErrorClass | 0x09)`
A certificate matching the specified criteria was not found.

`#define certMgrErrDatabaseFail (certErrorClass | 0x0B)`
A Data Manager error occurred.

`#define certMgrErrFieldNotFound (certErrorClass | 0x08)`
The specified field could not be found.

`#define certMgrErrInvalidEncoding (certErrorClass | 0x02)`
The specified format encoding is invalid.

`#define certMgrErrInvalidParam (certErrorClass | 0x04)`
One of the function parameters is invalid.

`#define certMgrErrNotExportable (certErrorClass | 0x0A)`
The certificate is not exportable. It is probably stored in compressed form.

`#define certMgrErrNotImplemented (certErrorClass | 0x01)`
The requested certificate format is not supported.

`#define certMgrErrNotRemovable (certErrorClass | 0x0D)`
The certificate is not removable.

`#define certMgrErrOutOfMemory (certErrorClass | 0x03)`
There was insufficient memory to complete the operation.

`#define certMgrErrOutOfResources (certErrorClass | 0x06)`
The Certificate Manager ran out of resources.

`#define certMgrErrServiceNotStarted (certErrorClass | 0x05)`
The Certificate Manager process has not started.

## Certificate Verification Failure Codes

**Purpose** Indicates why a certificate failed to verify. These values are returned in the `CertMgrVerifyResultType` structure's `failureCode` field as the result of a call to `CertMgrVerifyCert()` or `CertMgrAddCert()`. These are also passed to the SSL Library's Verify callback; see "The Verify Callback" on page 376.

**Declared In** `CertificateMgr.h`

**Constants** `#define CertMgrVerifyFail (certErrorClass+0x80)`

`#define CertMgrVerifyFailBasicConstraints (CertMgrVerifyFail+8)`
There was a constraint violation.

`#define CertMgrVerifyFailCriticalExtension (CertMgrVerifyFail+9)`
The critical extension is unknown.

`#define CertMgrVerifyFailKeyUsage (CertMgrVerifyFail+7)`

`#define CertMgrVerifyFailNotAfter (CertMgrVerifyFail+6)`

`#define CertMgrVerifyFailNotBefore (CertMgrVerifyFail+5)`

`#define CertMgrVerifyFailSelfSigned (CertMgrVerifyFail+4)`

`#define CertMgrVerifyFailSignature (CertMgrVerifyFail+3)`
The signature is invalid.

`#define CertMgrVerifyFailUnknown (CertMgrVerifyFail+0)`

`#define CertMgrVerifyFailUnknownIssuer (CertMgrVerifyFail+1)`
The root cannot be trusted since the issuer is not known.

`#define CertMgrVerifyFailUnknownSigAlg (CertMgrVerifyFail+2)`

## Miscellaneous Certificate Manager Constants

**Purpose** These constants are also declared in `CertificateMgr.h`.

**Declared In** `CertificateMgr.h`

**Constants** `#define CertMgrServiceName "pSysCertificateManager"`
The name under which the Certificate Manager is registered with the Service Manager.

# Certificate Manager Element Field Macros

## apCertMgrElementFieldRDNOIDN Macro

**Purpose** Macro that evaulates to the field ID for the the second and subsequent OID fields for an RDN element.

**Declared In** `CertificateMgr.h`

**Prototype** `#define apCertMgrElementFieldRDNOIDN (n)`

**Parameters** → *n*

The OID field index. The second OID field's index would be 2.

**Returns** Evaluates to the RDN element's OID field ID.

## apCertMgrElementFieldRDNValueN Macro

**Purpose** Macro that evaulates to the field ID for the second and subsequent Value fields for an RDN element..

**Declared In** `CertificateMgr.h`

**Prototype** `#define apCertMgrElementFieldRDNValueN (n)`

**Parameters** → *n*

The Value field index. The second Value field's index would be 2.

**Returns** Evaluates to the RDN element's Value field ID.

## apCertMgrElementFieldX509ExBytesN Macro

**Purpose** Macro that evaulates to the field ID for the the second and subsequent Bytes fields for an X509Extended element.

**Declared In** `CertificateMgr.h`

**Prototype** `#define apCertMgrElementFieldX509ExBytesN (n)`

**Parameters** → *n*

The Bytes field index. The second Bytes field's index would be 2.

**Returns** Evaluates to the X509Extended element's Bytes field ID.

## apCertMgrElementFieldX509ExCriticalN Macro

**Purpose** Macro that evaulates to the field ID for the the second and subsequent Criticial fields for an X509Extended element.

**Declared In** `CertificateMgr.h`

**Prototype** `#define apCertMgrElementFieldX509ExCriticalN (n)`

**Parameters** → *n*

The Critical field index. The second Critical field's index would be 2.

**Returns** Evaluates to the X509Extended element's Critical field ID.

## apCertMgrElementFieldX509ExOIDN Macro

**Purpose** Macro that evaulates to the field ID for the the second and subsequent OID fields for an X509Extended element.

**Declared In** `CertificateMgr.h`

**Prototype** `#define apCertMgrElementFieldX509ExOIDN (n)`

**Parameters** → *n*

The OID field index. The second OID field's index would be 2.

**Returns** Evaluates to the X509Extended element's OID field ID.

# Certificate Manager Functions and Macros

## CertMgrAddCert Function

**Purpose** Add a certificate to the certificate store.

**Declared In** `CertificateMgr.h`

**Prototype**
```
status_t CertMgrAddCert
    (CertMgrCertInfoType *certInfoP,
    Boolean compress,
    CertMgrVerifyResultType *verifyResult)
```

**Parameters**

→ *certInfoP*
  Pointer to `CertMgrCertInfoType` structure for the certificate to be added.

→ *compress*
  If `true`, the certificate is stored in compressed form. This saves space, but note that you cannot export certificates that are compressed.

← *verifyResult*
  Supply a pointer to a `CertMgrVerifyResultType` structure that will be filled in if the certificate could not be verified.

**Returns** Returns `errNone` if the certificate was added successfully, or one of the following otherwise:

`certMgrErrInvalidParam`
  *certInfoP* or *verifyResult* is `NULL` .

`certMgrErrServiceNotStarted`
  The Certificate Manager process has not started.

`certMgrErrDatabaseFail`
  A Data Manager error occurred.

`certMgrErrBackupInProgress`
  The certificate vault could not be accessed because it is in the process of being backed up.

**Comments** This function can be used to add an imported certificate to the certificate store. When a certificate becomes part of the store it is verified and then saved in the Certificate Manager's secure vault. Other applications may then query for it.

If the certificate cannot be added due to a failure in the verification of the certificate, this function fills in the caller-supplied `CertMgrVerifyResultType` structure. In many cases the caller might choose to override the verification failure and request that the certificate be stored anyway. The only errors that cannot be overridden are signature failure and unknown issuer. To override a verification error, clear the failure code in the `CertMgrVerifyResultType` structure and then call `CertMgrAddCert()` once again.

When the *`compress`* parameter is set to `true`, some data is thrown away from the certificate to save space. Because of this, the stored certificate is not complete and cannot be exported at a later time.

**Example** The following code excerpt adds certificates that may be self-signed:

```
while (true) {
   err = CertMgrAddCert(&certInfo, false, &verifyResult);
   if (err) {
      CertMgrReleaseCertInfo(&certInfo);
      goto exit;
   }

   if (verifyResult.failureCode == 0) {
      break;
   } else {
      if (verifyResult.failureCode ==
         CertMgrVerifyFailSelfSigned) {
         verifyResult.failureCode = 0;
         continue;
      }

      /* Another type of failure */
      break;
   }
}
```

**See Also** `CertMgrImportCert()`, `CertMgrRemoveCert()`, `CertMgrVerifyCert()`

## CertMgrExportCert Function

**Purpose** Exports a certificate from the certificate store to a caller-supplied buffer.

**Declared In** `CertificateMgr.h`

**Prototype**
```
status_t CertMgrExportCert
    (CertMgrCertInfoType *certInfoP,
    uint8_t *certData, uint32_t *certDataLen)
```

**Parameters**

→ *certInfoP*
Reference to the certificate to be exported.

← *certData*
Pointer to a caller-allocated buffer into which the exported certificate data will be written, or `NULL` to determine how large this buffer should be (the needed size, in bytes, is returned via *certDataLen*).

↔ *certDataLen*
When calling this function, `*`*certDataLen* should be set to the size of the *certData* buffer, or 0 to determine how large the *certData* buffer should be. Upon return `*`*certDataLen* is set to the size of the exported certificate data.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`certMgrErrBufTooSmall`
The export buffer is too small. The required size is written into `*`*certDataLen*.

`certMgrErrNotExportable`
The certificate is not exportable. It is probably stored in compressed form.

`certMgrErrDatabaseFail`
A Data Manager error occurred.

`certMgrErrBackupInProgress`
The certificate vault could not be accessed because it is in the process of being backed up.

**Comments** This function attempts to fill the supplied buffer with the same data that was imported into the Certificate Manager.

---

**NOTE:** In Palm OS Cobalt only DER encoded X509 format certificates are supported. Because of this, all exported certificate data will be in this format.

---

If the certificate was compressed when it was added to the store, it cannot be exported.

If the export buffer is not large enough, an error is returned along with the expected size.

**See Also** `CertMgrImportCert()`

## CertMgrFindCert Function

**Purpose** Search the certificate store for a certificate matching the specified criteria.

**Declared In** `CertificateMgr.h`

**Prototype**

```
status_t CertMgrFindCert (uint32_t *index,
    CertMgrCertSearchEnum searchFlag,
    uint8_t *reference, uint32_t referenceLen,
    CertMgrCertInfoType *certInfoP)
```

**Parameters**

↔ *`index`*
: Set to 0 to start new search. As a certificate matching the specified criteria is found, `*`*`index`* is set to the index of the certificate within the certificate store.

→ *`searchFlag`*
: Value that specifies how the search is to be performed. Supply one of the values listed under `CertMgrCertSearchEnum`.

→ *`reference`*
: The data being searched for. If *`searchFlag`* is `apCertMgrSearchCertID`, this should be the 20-byte certID being searched for. If *`searchFlag`* is `apCertMgrSearchSubjectRDN` search, this should be the SubjectRDN being searched for.

→ *`referenceLen`*
: The size, in bytes, of the data pointed to by *`reference`*.

← *certInfoP*
Pointer to a `CertMgrCertInfoType` structure that is filled in as appropriate to identify the certificate that was found.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`certMgrErrInvalidParam`
One of the function parameters is invalid.

`certMgrErrServiceNotStarted`
The Certificate Manager process has not started.

`certMgrErrCertNotFound`
A certificate matching the specified criteria was not found.

`certMgrErrBackupInProgress`
The certificate vault could not be accessed because it is in the process of being backed up.

**Example** To find a certificate with the certificate ID in `certificateID`, do the following:

```
err = CertMgrFindCert(0, apCertMgrSearchCertID,
certificateID, 20,
   &certInfo);
```

**See Also** `CertMgrReleaseCertInfo()`

## CertMgrGetField Function

**Purpose** Get the value of a certificate field.

**Declared In** `CertificateMgr.h`

**Prototype**
```
status_t CertMgrGetField
    (CertMgrCertInfoType *certInfoP,
    CertMgrCertElementEnum elementType,
    CertMgrCertFieldEnum fieldType,
    CertMgrElementListType *result,
    uint32_t *resultLengthP)
```

**Parameters** → *certInfoP*
Reference to the certificate from which the field is to be retrieved.

→ *elementType*
: The certificate element type. This should be one of the values listed under `CertMgrCertElementEnum`.

→ *fieldType*
: The type of field to be retrieved. See `CertMgrCertFieldEnum`. Note that the set of values that can be supplied to this parameter varies depending on the value of the *elementType* parameter.

← *result*
: Pointer to a buffer into which the field data is written, or `NULL` to obtain the size of the needed buffer. The size of the field data (actual or needed) is written into **resultLengthP*. Note that the contents of the buffer upon return are structured according to `CertMgrElementListType`.

↔ *resultLengthP*
: When calling this function, **resultLengthP* should be set to the size of the *result* buffer, or 0 to determine how large the *result* buffer should be. Upon return **resultLengthP* is set to the size of the field data.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`certMgrErrInvalidParam`
: *certInfoP* is `NULL` or it is not a valid certificate reference, or *resultLengthP* is `NULL`.

`certMgrErrBufTooSmall`
: The export buffer is too small. The required size is written into **resultLengthP*.

`certMgrErrFieldNotFound`
: The specified field could not be found.

**Example** The following code excerpt shows you you might use this function.

```
#define CertMgrOrganicationNameOIDLen   5
static uint8_t
CertMgrOrganizationNameOID[CertMgrOrganicationNameOIDLen] =
   {0x06, 0x03, 0x55, 0x04, 0x0a};

/* Getting the issuer RDN from the cert */
CertMgrImplGetField(certInfoP, apCertMgrElementTypeRDN,  0,
   field, &fieldlen);
```

```
/* This gets the whole issuer RDN, you must then go through
each field and find the one that you want, an OID field leads
a data field */

/* The following code finds the code for the issuer name and
sets a label to it */
/* Fields in list */
for (index = 0; index < field->count; index++) {

   if (field->element[index].dataType ==
      apCertMgrElementDataTypeASN1OID) {
      if (field->element[index].length ==
         CertMgrOrganicationNameOIDLen) {
         if (MemCmp(CertMgrOrganizationNameOID,
            ((uint8_t *)field)+field->element[index].offset,
            CertMgrOrganizationNameOIDLen) == 0) {

            uint16_t buflen = field->element[index +
               1].length;
            uint8_t *buffer = ((uint8_t *)field) +
               field->element[index + 1].offset;
            uint16_t count  = 0;

            Char label[40];
            uint16_t pre = 0;

            MemSet(label, 40, 0);

            if (buflen < 40) {
               pre = (40 - buflen) / 2;

               MemSet(label, pre, ' ');
            }

            MemMove(label+pre,buffer,buflen>39 ? 39:buflen);

            FrmCopyLabel(frmP, selfsignedaddCertnameLabel,
               label);
            break;
         }
      }
   }
}
```

## CertMgrImportCert Function

**Purpose** Imports a certificate from a buffer into the certificate store.

**Declared In** `CertificateMgr.h`

**Prototype**
```
status_t CertMgrImportCert (uint8_t *certData,
    uint32_t certDataLen,
    CertMgrCertInfoType *certInfoP)
```

**Parameters**

→ *`certData`*
Pointer to a buffer containing the certificate data being imported.

→ *`certDataLen`*
The size, in bytes, of the data in *`certData`*.

↔ *`certInfoP`*
When calling this function, you can optionally specify the format of the certificate data by setting the `format` field of this structure. Upon return, this structure's fields are filled in appropriately to identify the certificate that was imported.

**NOTE:** In Palm OS Cobalt only DER encoded X509 format certificates are supported.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`certMgrErrInvalidEncoding`
The specified format encoding is invalid.

`certMgrErrOutOfMemory`
There was insufficient memory to complete the operation.

`certMgrErrInvalidParam`
One of the function parameters is invalid.

`certMgrErrServiceNotStarted`
The Certificate Manager process has not started.

`certMgrErrBackupInProgress`
The certificate vault could not be accessed because it is in the process of being backed up.

`certMgrErrNotImplemented`
The requested certificate format is not supported.

**Example** A likely scenario would be an application that imports a certificate that was stored in the its PRC. The following code excerpt shows how to do this:

```
/* Get certificate from PRC */
err = SignGetCertificateByIndex(dbP, certIndex, &certBlock,
   &certDataLength, certData);

/* Load certificate onto cert mgr */
err = CertMgrImportCert(certData, certDataLength, &certInfo);
```

**See Also** `CertMgrExportCert()`

## CertMgrReleaseCertInfo Function

**Purpose** Release resources that allocated by the Certificate Manager during a successful call to `CertMgrFindCert()` or `CertMgrImportCert()`.

**Declared In** `CertificateMgr.h`

**Prototype**
```
status_t CertMgrReleaseCertInfo
    (CertMgrCertInfoType *certInfoP)
```

**Parameters** → *`certInfoP`*
Reference to the certificate for which resources are to be released..

**Returns** Returns `errNone` if the operation completed successfully, or `certMgrErrInvalidParam` if *`certInfoP`* is invalid.

**Comments** Failure to call this function after a successful call to `CertMgrFindCert()` or `CertMgrImportCert()` will result in a memory leak.

## CertMgrRemoveCert Function

**Purpose** Remove a certificate from the certificate store.

**Declared In** `CertificateMgr.h`

**Prototype**
```
status_t CertMgrRemoveCert
    (CertMgrCertInfoType *certInfoP)
```

**Parameters** → *certInfoP*
Pointer to `CertMgrCertInfoType` structure for the certificate to be added.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`certMgrErrInvalidParam`
*certInfoP* is invalid.

`certMgrErrBackupInProgress`
The certificate vault could not be accessed because it is in the process of being backed up.

`certMgrErrNotRemovable`
The certificate is not removable.

**Comments** This function does not invalidate any other certificates already in the store. However, it may cause further verifications of new certificates to fail (for instance, if a root certificate is removed).

**See Also** `CertMgrAddCert()`

## CertMgrVerifyCert Function

**Purpose** Authenticate the validity of a certificate.

**Declared In** `CertificateMgr.h`

**Prototype**
```
status_t CertMgrVerifyCert
    (CertMgrCertInfoType *certInfoP,
    CertMgrCertChainType *certChainP,
    CertMgrVerifyResultType *verifyResult)
```

**Parameters** → *certInfoP*
Pointer to `CertMgrCertInfoType` structure for the certificate to be validated.

→ *certChainP*
A chain of certificates that make up the authentication tree for this certificate.

← *verifyResult*
Supply a pointer to a `CertMgrVerifyResultType` structure that will be filled in if the certificate could not be verified.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`certMgrErrInvalidParam`
*certInfoP* or *verifyResult* is `NULL`.

`certMgrErrOutOfMemory`
There was insufficient memory to complete the operation.

`certMgrErrOutOfResources`
The Certificate Manager ran out of resources.

`certMgrErrServiceNotStarted`
The Certificate Manager process has not started.

This function authenticates the validity of the certificate. Many different error conditions may occur, and they are returned through the *verifyResult* parameter.

Specify a certificate chain when the chain of certificates that authenticate the specified certificate is not contained in the Certificate Manager's certificate store. (For instance, when all or some of the certificatess in the chain have just been imported but not yet added.) When necessary, the Certificate Manager will also authenticate a certificate in the chain. The chain need not be in any order, and the certificates in the chain need not all be part of the chain.

**See Also** `CertMgrAddCert()`

## CertMgrVerifyFailure Macro

**Purpose** Determine if a given error code is a Certificate Manager verification error.

**Declared In** `CertificateMgr.h`

**Prototype** `#define CertMgrVerifyFailure (err)`

**Parameters** → *err*
The error code being checked.

**Returns** Evaluates to `true` if the supplied error code is within the range of verification failure errors, `false` otherwise. See "Certificate Verification Failure Codes" on page 152 for those error codes that are classified as verification errors.

# 9

# CPM Library ARM Interface

The functions documented in this chapter constitute the interface implementation for the Cryptographic Provider Manager library in Palm OS Cobalt. This chapter consists of a single section:



The header file `CPMLibARMInterface.h` declares the API that this chapter describes.

## CPM Library ARM Interface Functions and Macros

### CPMLibAddRandomSeed Function

**Purpose** Puts a number of seed bytes into the pseudo-random number generator maintained by the CPM.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibAddRandomSeed (uint8_t *seedDataP,
    uint32_t dataLen)
```

**Parameters**

→ *seedDataP*
A buffer of seed bytes.

→ *dataLen*
The number of bytes in *seedDataP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibGenerateRandomBytes()`

## CPMLibClose Function

**Purpose** Handles the closing of the CPM library.

**Declared In** `CPMLibARMInterface.h`

**Prototype** `status_t CPMLibClose (void)`

**Parameters** None.

**Returns** `errNone` if the operation completed successfully, or one of the following otherwise:

`cpmErrNotOpen`
: The CPM library is not open.

**Comments** Decrements the reference count. When the reference count reaches zero, memory is cleared out and freed, all resources are returned to the system, and the library is taken out of the system.

---

**WARNING!** If you completely close the CPM library (to the point where the reference count is zero), you can prevent other operating system functionality from working (SSL, Authorization Manager, Certification Manager, some areas of the Data Manager and the System library, plus possibly others). Never call `CPMLibClose()` more times than you have called `CPMLibOpen()`.

---

**See Also** `CPMLibOpen()`, `CPMLibSleep()`, `CPMLibWake()`

## CPMLibDecrypt Function

**Purpose** Performs the decryption operation in one pass.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibDecrypt (APKeyInfoType *keyInfoP,
    APCipherInfoType *cipherInfoP, uint8_t *bufIn,
    uint32_t bufInLen, uint8_t *bufOut,
    uint32_t *bufOutLenP)
```

**Parameters** → *keyInfoP*
: An `APKeyInfoType` structure, allocated and optionally initialized by the application, that holds the key to be used for the operation. Note that for this single-part operation, this

structure is not required unless the application wants to pass setting information to or receive setting information from the CPM or provider.

↔ *cipherInfoP*
An `APCipherInfoType` structure, allocated and optionally initialized by the application, that holds the context information to be used for this operation. Note that for this single-part operation, this structure is not required unless the application wants to pass setting information to or receive setting information from the CPM or provider.

→ *bufIn*
Pointer to a buffer containing the data for the operation. This parameter cannot be `NULL`.

→ *bufInLen*
Size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
Pointer to a buffer, allocated by the application, that receives the output of the operation.

↔ *bufOutLenP*
The length, in bytes, of the buffer specified by *bufOut*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` and `APCipherInfoType` structures. The application must also call `CPMLibReleaseKeyInfo()` or `CPMLibReleaseCipherInfo()`, as appropriate, before freeing the structure to allow the CPM and the provider(s) to clean up.

**See Also** `CPMLibDecryptFinal()`, `CPMLibDecryptInit()`, `CPMLibDecryptUpdate()`, `CPMLibEncrypt()`, `CPMLibReleaseCipherInfo()`, `CPMLibReleaseKeyInfo()`

## CPMLibDecryptFinal Function

**Purpose** Finalizes a multi-part decryption operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibDecryptFinal
    (APKeyInfoType *keyInfoP,
    APCipherInfoType *cipherInfoP, uint8_t *bufIn,
    uint32_t bufInLen, uint8_t *bufOut,
    uint32_t *bufOutLenP)
```

**Parameters**

→ *keyInfoP*
The key to be used for the operation.

↔ *cipherInfoP*
The context returned from `CPMLibDecryptInit()`.

→ *bufIn*
Pointer to a buffer containing the final data for the operation, or `NULL` if there is no additional data.

→ *bufInLen*
Size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
Pointer to a buffer, allocated by the application, that receives the output of the operation.

↔ *bufOutLenP*
The length, in bytes, of the buffer specified by *bufOut*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibDecryptInit()`, `CPMLibDecryptUpdate()`, `CPMLibEncryptFinal()`, `CPMLibReleaseCipherInfo()`

## CPMLibDecryptInit Function

**Purpose** Begins a multi-part decryption operation with the specified key and returns the context of the operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibDecryptInit
    (APKeyInfoType *keyInfoP,
    APCipherInfoType *cipherInfoP)
```

**Parameters**

→ *keyInfoP*
: Pointer to an `APKeyInfoType` structure, allocated and optionally initialized by the application, containing the key to be used for the subsequent calls to `CPMLibDecryptUpdate()` and `CPMLibDecryptFinal()`.

↔ *cipherInfoP*
: An `APCipherInfoType` structure, allocated and optionally initialized by the application, that holds the context information for use in subsequent calls to the same class of operations.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` and `APCipherInfoType` structures. The application must also call `CPMLibReleaseKeyInfo()` or `CPMLibReleaseCipherInfo()`, as appropriate, before freeing the structure to allow the CPM and the provider(s) to clean up.

You must call `CPMLibDecryptFinal()` to finalize the operation.

**See Also** `CPMLibDecrypt()`, `CPMLibDecryptFinal()`, `CPMLibDecryptUpdate()`, `CPMLibEncryptInit()`, `CPMLibReleaseCipherInfo()`

## CPMLibDecryptUpdate Function

**Purpose** Updates a multi-part decryption operation with more data.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibDecryptUpdate
   (APKeyInfoType *keyInfoP,
   APCipherInfoType *cipherInfoP, uint8_t *bufIn,
   uint32_t bufInLen, uint8_t *bufOut,
   uint32_t *bufOutLenP)
```

**Parameters**

→ *keyInfoP*
The key to be used for the operation.

↔ *cipherInfoP*
The context returned from `CPMLibDecryptInit()`.

→ *bufIn*
Pointer to a buffer containing the data for the operation. This parameter cannot be `NULL`.

→ *bufInLen*
Size, in bytes, of the buffer specified by *bufIn*. This value must be greater than zero.

↔ *bufOut*
Pointer to a buffer, allocated by the application, that receives the output of the operation.

↔ *bufOutLenP*
The length, in bytes, of the buffer specified by *bufOut*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibDecryptFinal()`, `CPMLibDecryptInit()`, `CPMLibEncryptUpdate()`

## CPMLibDeriveKeyData Function

**Purpose** Derives a key from the supplied input data.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibDeriveKeyData
   (APDerivedKeyInfoType *derivedKeyInfoP,
   uint8_t *keyDataP, uint32_t *dataLen)
```

**Parameters**

↔ *derivedKeyInfoP*
Pointer to an `APDerivedKeyInfoType` structure.

→ *keyDataP*
Pointer to a buffer into which the derived key data is written. Pass `NULL` to determine how large this buffer should be.

↔ *dataLen*
When calling this function, set the variable to which this parameter points to the size of the *keyDataP* buffer. Upon return, the variable will be set to the number of bytes written to *keyDataP*. If you set *keyDataP* to `NULL`, set this variable to 0.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** Given the same input data, the same key is always derived from that data. Key derivation is useful for operations such as Password-Based Encryption (PBE) where the password is used to derive a key for a particular cryptographic operation (usually encryption or decryption).

Unlike with generated keys, applications typically do not export and save derived keys since they can be re-derived from the same input data.

**Example** To determine how large a buffer you need to allocate for the derived key data, set *keyDataP* to `NULL` and `*`*dataLen* to `0`. `CPMLibDeriveKeyData()` will return an error code of `cpmErrBuffTooSmall` and will set the variable pointed to by *dataLen* to the needed buffer size. Your code can then allocate the needed buffer and again call `CPMLibDeriveKeyData()` with a pointer to the buffer, as shown in the following code excerpt:

```
uint32_t size;
uint32_t *key_data;

// The APDerivedKeyInfoType structure is initialized
// prior to this point
size = 0;
err = CPMLibDeriveKeyData(&dki, NULL, &size);
if (err == cpmErrBufTooSmall) {
   key_data = MemPtrNew(size);
   if (key_data != NULL) {
      err = CPMLibDeriveKeyData(&dki, key_data, &size);
      if (err) {
         // handle errors here
      } else {
         // The key data was successfully derived; use
         // key_data as import data to get a key
         MemSet(&keyInfo, sizeof(APKeyInfoType), 0);
         err = CPMLibImportKeyInfo(IMPORT_EXPORT_TYPE_RAW,
            key_data, size, &keyInfo);
         if (err) {
            // handle errors here
         } else {
            // At this point, we have an APKeyInfoType struct
         }
      }
   }
}
```

**See Also** `CPMLibGenerateKey()`, `CPMLibGenerateKeyPair()`

## CPMLibEncrypt Function

**Purpose** Performs an encryption operation in one pass.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibEncrypt (APKeyInfoType *keyInfoP,
   APCipherInfoType *cipherInfoP, uint8_t *bufIn,
   uint32_t bufInLen, uint8_t *bufOut,
   uint32_t *bufOutLenP)
```

**Parameters**

→ *keyInfoP*
: An `APKeyInfoType` structure, allocated and optionally initialized by the application, that holds the key to be used for the operation. Note that for this single-part operation, this structure is not required unless the application wants to pass setting information to or receive setting information from the CPM or provider.

↔ *cipherInfoP*
: An `APCipherInfoType` structure, allocated and optionally initialized by the application, that holds the context information to be used for this operation. Note that for this single-part operation, this structure is not required unless the application wants to pass setting information to or receive setting information from the CPM or provider.

→ *bufIn*
: Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

→ *bufInLen*
: Size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
: Pointer to a buffer, allocated by the application, that receives the output of the operation.

↔ *bufOutLenP*
: The length, in bytes, of the buffer specified by *bufOut*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` and `APCipherInfoType` structures. The application must also call `CPMLibReleaseKeyInfo()` or

CPMLibReleaseCipherInfo(), as appropriate, before freeing the structure to allow the CPM and the provider(s) to clean up.

**See Also** CPMLibDecrypt(), CPMLibEncryptFinal(), CPMLibEncryptInit(), CPMLibEncryptUpdate(), CPMLibReleaseCipherInfo()

## CPMLibEncryptFinal Function

**Purpose** Finalizes a multi-part encryption operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibEncryptFinal
   (APKeyInfoType *keyInfoP,
   APCipherInfoType *cipherInfoP, uint8_t *bufIn,
   uint32_t bufInLen, uint8_t *bufOut,
   uint32_t *bufOutLenP)
```

**Parameters**

→ *keyInfoP*
The key to be used for the operation.

→ *cipherInfoP*
The context information returned from the CPMLibEncryptInit() call.

→ *bufIn*
Pointer to a buffer containing the final data for the operation, or `NULL` if there is no additional data.

→ *bufInLen*
Size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
Pointer to a buffer, allocated by the application, that receives the output of the operation.

↔ *bufOutLenP*
The length, in bytes, of the buffer specified by *bufOut*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under “CPM Library Error Codes” on page 239 otherwise.

**See Also** CPMLibDecryptFinal(), CPMLibEncryptInit(), CPMLibEncryptUpdate(), CPMLibReleaseCipherInfo()

## CPMLibEncryptInit Function

**Purpose** Begins multi-part encryption operation with the specified key and returns the context of the operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibEncryptInit
    (APKeyInfoType *keyInfoP,
    APCipherInfoType *cipherInfoP)
```

**Parameters**

→ *keyInfoP*
: Pointer to an `APKeyInfoType` structure, allocated and optionally initialized by the application, containing the key to be used for the subsequent calls to `CPMLibEncryptUpdate()` and `CPMLibEncryptFinal()`.

↔ *cipherInfoP*
: An `APCipherInfoType` structure, allocated and optionally initialized by the application, that holds the context information for use in subsequent calls to the same class of operations.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` and `APCipherInfoType` structures. The application must also call `CPMLibReleaseKeyInfo()` or `CPMLibReleaseCipherInfo()`, as appropriate, before freeing the structure to allow the CPM and the provider(s) to clean up.

You must call `CPMLibEncryptFinal()` to finalize the operation.

**See Also** `CPMLibDecryptInit()`, `CPMLibEncrypt()`, `CPMLibEncryptFinal()`, `CPMLibEncryptUpdate()`, `CPMLibReleaseCipherInfo()`

## CPMLibEncryptUpdate Function

**Purpose** Updates a multi-part encryption operation with more data.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibEncryptUpdate
   (APKeyInfoType *keyInfoP,
   APCipherInfoType *cipherInfoP, uint8_t *bufIn,
   uint32_t bufInLen, uint8_t *bufOut,
   uint32_t *bufOutLenP)
```

**Parameters**

→ *keyInfoP*
The key to be used for the operation.

→ *cipherInfoP*
The context information returned from the `CPMLibEncryptInit()` call.

→ *bufIn*
Pointer to a buffer containing the data for the operation.

→ *bufInLen*
Size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
Pointer to a buffer, allocated by the application, that receives the output of the operation.

↔ *bufOutLenP*
The length, in bytes, of the buffer specified by *bufOut*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The *bufIn* parameter may not be `NULL` and *bufInLen* may not be zero.

**See Also** `CPMLibDecryptUpdate()`, `CPMLibEncryptFinal()`, `CPMLibEncryptInit()`

## CPMLibEnumerateProviders Function

**Purpose** Enumerates the providers that the CPM library currently knows about.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibEnumerateProviders
    (uint32_t providerIDs[],
    uint16_t *numProviders)
```

**Parameters**

→ *`providerIDs`*
An array of provider IDs for the providers about which the CPM currently knows.

← *`numProviders`*
The number of providers currently installed under the CPM library. Also the number of IDs in the *`providerIDs`* parameter.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The CPM library returns the IDs of the providers. The IDs can be used to specifically reference a particular provider.

**See Also** `CPMLibGetInfo()`, `CPMLibGetProviderInfo()`, `CPMLibOpen`

## CPMLibExportCipherInfo Function

**Purpose** Creates a storable instance of an `APCipherInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibExportCipherInfo
    (APCipherInfoType *cipherInfoP,
    uint8_t encoding, uint8_t *exportDataP,
    uint32_t *dataLenP)
```

**Parameters**

↔ *`cipherInfoP`*
An `APCipherInfoType` structure, allocated and optionally initialized by the application, that holds the context information to be used for this operation.

→ *encoding*
One of the encodings documented under "Import/Export Types" on page 237.

↔ *exportDataP*
Pointer to a buffer to receive the raw exported data.

↔ *dataLenP*
Size, in bytes, of the buffer specified by *exportDataP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APCipherInfoType` structure. The application must also call `CPMLibReleaseCipherInfo()` before freeing the `APICipherInfoType` structure to allow the CPM and the provider(s) to clean up.

**See Also** `CPMLibImportCipherInfo()`, `CPMLibReleaseCipherInfo()`

## CPMLibExportHashInfo Function

**Purpose** Creates a storable instance of an `APHashInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibExportHashInfo
    (APHashInfoType *hashInfoP, uint8_t encoding,
    uint8_t *exportDataP, uint32_t *dataLenP)
```

**Parameters**

↔ *hashInfoP*
An `APHashInfoType` structure, allocated by the application, that holds information about the hashing operation.

→ *encoding*
One of the encodings documented under "Import/Export Types" on page 237.

↔ *exportDataP*
Pointer to a buffer, allocated by the application, into which the raw exported data will be placed.

↔ *dataLenP*
The size, in bytes, of the buffer specified by *exportDataP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibImportHashInfo()`

## CPMLibExportKeyInfo Function

**Purpose** Creates a storable instance of a key that is already familiar to the CPM framework.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibExportKeyInfo
    (APKeyInfoType *keyInfoP, uint8_t encoding,
    uint8_t *exportDataP, uint32_t *dataLenP)
```

**Parameters**

↔ *keyInfoP*
: An `APKeyInfoType` structure, allocated and optionally initialized by the application, that holds information about the key to be used for this operation.

→ *encoding*
: One of the values documented under "Import/Export Types" on page 237.

↔ *exportDataP*
: Pointer to a buffer, allocated by the application, into which the raw exported data is to be placed.

↔ *dataLenP*
: The size of the buffer specified by *exportDataP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** If a generated key is used for any cryptographic operations, it must be exported and saved in order to be used again. It is statistically improbable that a generated key could be regenerated. Derived keys, on the other hand, are generally not exported since given the same input data (often a password or something similar), the same key is always derived from that data.

The application is always responsible for allocating and freeing the `APKeyInfoType` structure. The application must call

CPMLibReleaseKeyInfo() before freeing the `APKeyInfoType` structure to allow the CPM and provider to clean up.

**See Also** CPMLibImportKeyInfo(), CPMLibReleaseKeyInfo()

## CPMLibExportKeyPairInfo Function

**Purpose** Creates a storable instance of a set of APKeyInfoType structures representing a private key and a public key.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibExportKeyPairInfo
   (APKeyInfoType *privateKeyInfoP,
   APKeyInfoType *publicKeyInfoP,
   uint8_t encoding, uint8_t *exportDataP,
   uint32_t *dataLenP)
```

**Parameters**

↔ *privateKeyInfoP*
: Pointer to an APKeyInfoType structure for the private key.

↔ *publicKeyInfoP*
: Pointer to an APKeyInfoType structure for the public key.

→ *encoding*
: One of the values documented under "Import/Export Types" on page 237.

↔ *exportDataP*
: Pointer to a buffer, allocated by the application, into which the raw exported data is placed.

↔ *dataLenP*
: The size, in bytes, of the buffer indicated by *exportDataP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` structures. The application must call CPMLibReleaseKeyInfo() before freeing an `APKeyInfoType` structure to allow the CPM and provider to clean up.

**See Also** CPMLibImportKeyPairInfo(), CPMLibReleaseKeyInfo()

## CPMLibExportMACInfo Function

**Purpose** Creates a storable instance of an `APMACInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibExportMACInfo
    (APMACInfoType *macInfoP, uint8_t encoding,
    uint8_t *exportDataP, uint32_t *dataLenP)
```

**Parameters**

↔ *macInfoP*
An `APMACInfoType` structure, allocated and optionally initialized by the application, that holds the message authentication context information to be used for this operation.

→ *encoding*
One of the encodings documented under "Import/Export Types" on page 237.

↔ *exportDataP*
Pointer to a buffer to receive the raw exported data.

↔ *dataLenP*
Size, in bytes, of the buffer specified by *exportDataP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APMACInfoType` structures. The application must call `CPMLibReleaseMACInfo()` before freeing an `APMACInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibImportMACInfo()`, `CPMLibReleaseMACInfo()`

## CPMLibExportSignInfo Function

**Purpose** Creates a storable instance of an `APSignInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibExportSignInfo
   (APSignInfoType *signInfoP, uint8_t encoding,
   uint8_t *exportDataP, uint32_t *dataLenP)
```

**Parameters**

↔ *signInfoP*
An `APSignInfoType` structure, allocated and optionally initialized by the application, that holds the context information to be used for this operation.

→ *encoding*
One of the encodings documented under "Import/Export Types" on page 237.

↔ *exportDataP*
Pointer to a buffer to receive the raw exported data.

↔ *dataLenP*
Size, in bytes, of the buffer specified by *exportDataP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APSignInfoType` structures. The application must call `CPMLibReleaseSignInfo()` before freeing an `APSignInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibImportSignInfo()`, `CPMLibReleaseSignInfo()`

## CPMLibExportVerifyInfo Function

**Purpose** Creates a storable instance of an `APVerifyInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibExportVerifyInfo
   (APVerifyInfoType *verifyInfoP,
   uint8_t encoding, uint8_t *exportDataP,
   uint32_t *dataLenP)
```

**Parameters**

↔ *verifyInfoP*
An `APVerifyInfoType` structure, allocated and optionally initialized by the application, that holds the context information to be used for this operation.

→ *encoding*
One of the encodings documented under "Import/Export Types" on page 237.

↔ *exportDataP*
Pointer to a buffer to receive the raw exported data.

↔ *dataLenP*
Size, in bytes, of the buffer specified by *exportDataP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APVerifyInfoType` structures. The application must call `CPMLibReleaseVerifyInfo()` before freeing an `APSVerifyInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibImportVerifyInfo()`, `CPMLibReleaseVerifyInfo()`

## CPMLibGenerateKey Function

**Purpose** Generates a new key.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibGenerateKey (uint8_t *keyDataP,
   uint32_t dataLen, APKeyInfoType *keyInfoP)
```

**Parameters**

→ *keyDataP*
: Pointer to a buffer of seed bytes to be used by the pseudo-random number generator, or `NULL` to have the pseudo-random number generator use the seed data it already has.

→ *dataLen*
: The length, in bytes, of the buffer pointed to by *keyDataP*.

↔ *keyInfoP*
: An `APKeyInfoType` structure, allocated and optionally initialized by the application, into which the generated key is written.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** If the newly-generated key is utilized for any cryptographic operations, it must be exported and saved in order to be used again. It is statistically improbably that a generated key could be regenerated.

The application is always responsible for allocating and freeing the `APKeyInfoType` structure. The application must call `CPMLibReleaseKeyInfo()` before freeing the `APKeyInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibExportKeyInfo()`, `CPMLibImportKeyInfo()`

## CPMLibGenerateKeyPair Function

**Purpose** Generates a new public/private key pair.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibGenerateKeyPair
   (uint8_t *keyDataP, uint32_t dataLen,
   APKeyInfoType *privateKeyInfoP,
   APKeyInfoType *publicKeyInfoP)
```

**Parameters**

→ *keyDataP*
Pointer to a buffer of seed bytes to be used by the pseudo-random number generator, or `NULL` to have the pseudo-random number generator use the seed data it already has.

→ *dataLen*
The length, in bytes, of the buffer pointed to by *keyDataP*.

↔ *privateKeyInfoP*
Pointer to the `APKeyInfoType` structure for the private key.

↔ *publicKeyInfoP*
Pointer to the `APKeyInfoType` structure for the public key.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** If the newly-generated key pair is utilized for any cryptographic operations, the pair must be exported and saved in order to be used again. It is statistically improbably that a generated key pair could be regenerated.

The application is always responsible for allocating and freeing the `APKeyInfoType` structures. The application must call `CPMLibReleaseKeyInfo()` before freeing an `APKeyInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibExportKeyPairInfo()`, `CPMLibImportKeyPairInfo()`

## CPMLibGenerateRandomBytes Function

**Purpose** Returns a requested number of random bytes.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibGenerateRandomBytes
    (uint8_t *bufferP, uint32_t *bufLenP)
```

**Parameters**

← *bufferP*
Pointer to a buffer, allocated by the application, into which the random bytes are written.

↔ *bufLenP*
When calling this function, set the variable pointed to by this parameter to the size of *bufferP*. Upon return, the variable contains the number of random bytes written to *bufferP*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** If there are not *bufLenP* random bytes available, this function returns the available random bytes and returns the number of available bytes in *bufLenP*.

**See Also** `CPMLibAddRandomSeed()`

## CPMLibGetInfo Function

**Purpose** Returns information about the CPM library as its currently known to the system. This includes the number of instances of the CPM library, the number of providers the CPM library is aware of, and whether or not the default provider is known.

**Declared In** `CPMLibARMInterface.h`

**Prototype** `status_t CPMLibGetInfo (CPMInfoType *infoP)`

**Parameters**

← *infoP*
Information about the CPM library. See `CPMInfoType`.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibEnumerateProviders()`, `CPMLibGetProviderInfo()`

## CPMLibGetProviderInfo Function

**Purpose** Gets information about the requested provider. Information returned includes the name of the provider, some additional text about the provider, the "algorithms" supported, and so on.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibGetProviderInfo
    (uint32_t providerID,
    APProviderInfoType *providerInfoP)
```

**Parameters**

→ *providerID*
A provider ID referencing the provider for which info is being requested.

← *providerInfoP*
A `APProviderInfoType` structure, allocated by the application, into which information about the provider is written.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibEnumerateProviders()`, `CPMLibGetInfo()`

## CPMLibHash Function

**Purpose** Performs the hashing operation in one pass.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibHash (APHashEnum type,
    APHashInfoType *hashinfo, uint8_t *bufIn,
    uint32_t bufInLen, uint8_t *bufOut,
    uint32_t *bufOutLenP)
```

**Parameters**

→ *type*
The algorithm provider hash type. One of the values defined by the `APHashEnum`.

↔ *hashinfo*
Pointer to an `APHashInfoType` structure, allocated by the application, into which the context is stored.

→ *bufIn*
Pointer to a buffer containing the data for the operation. This parameter cannot be `NULL`.

→ *bufInLen*
The size, in bytes, of the buffer specified by the *bufIn* parameter.

↔ *bufOut*
Pointer to a buffer, allocated by the application, that receives the output of the operation.

↔ *bufOutLenP*
The size, in bytes, of the buffer specified by *bufOut*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibHashFinal()`, `CPMLibHashInit()`, `CPMLibHashUpdate()`

## CPMLibHashFinal Function

**Purpose** Finalizes a multi-part hash operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibHashFinal
    (APHashInfoType *hashinfo, uint8_t *bufIn,
    uint32_t bufInLen, uint8_t *bufOut,
    uint32_t *bufOutLenP)
```

**Parameters**

→ *hashinfo*
The context returned from the call to `CPMLibHashInit()`. This is an `APHashInfoType` structure.

→ *bufIn*
Pointer a buffer containing the final data for the operation, or `NULL` if there is no additional data.

→ *bufInLen*
The size of the buffer specified by the *bufIn* parameter.

↔ *bufOut*
Pointer to a buffer, allocated by the application, that receives the output of the operation.

↔ *bufOutLenP*
The size, in bytes, of the buffer specified by *bufOut*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The results of the hash operation are placed in *bufOut*.

> **NOTE:** When this function returns, the context in `*`*hashinfo* is no longer valid.

**See Also** `CPMLibHashInit()`, `CPMLibHashUpdate()`, `CPMLibReleaseHashInfo()`

## CPMLibHashInit Function

**Purpose** Begins a multi-part hash operation of a specified type and returns the context of the hash operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibHashInit
    (APHashInfoType *hashinfo)
```

**Parameters** ↔ *hashinfo*
Pointer to an `APHashInfoType` structure, allocated by the application, into which the context is stored. This context is needed by the `CPMLibHashUpdate()` and `CPMLibHashFinal()` functions.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** Requires that `CPMLibHashFinal()` must be called to free the context.

**See Also** `CPMLibHash()`, `CPMLibHashFinal()`, `CPMLibHashUpdate()`

## CPMLibHashUpdate Function

**Purpose** Updates a multi-part hash operation with more data.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibHashUpdate
   (APHashInfoType *hashinfo, uint8_t *bufIn,
   uint32_t bufInLen)
```

**Parameters**

→ *hashinfo*
The context returned from the call to `CPMLibHashInit()`. This is an `APHashInfoType` structure.

→ *bufIn*
Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

→ *bufInLen*
The size of the buffer specified by the *bufIn* parameter. This value must be greater than zero.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibHashFinal()`, `CPMLibHashInit()`

## CPMLibImportCipherInfo Function

**Purpose** Initialize the contents of an `APCipherInfoType` structure based upon a storable instance of that structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibImportCipherInfo
   (uint8_t encoding, uint8_t *importDataP,
   uint32_t dataLen,
   APCipherInfoType *cipherInfoP)
```

**Parameters**

→ *encoding*
One of the encodings documented under "Import/Export Types" on page 237.

→ *importDataP*
Pointer to a buffer containing the raw data to be imported.

→ *dataLen*

Length, in bytes, of the buffer specified by *importDataP*.

↔ *cipherInfoP*

An APCipherInfoType structure, allocated and optionally initialized by the application, that holds the context information to be used for this operation.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APCipherInfoType` structure. The application must also call `CPMLibReleaseCipherInfo()` before freeing the `APICipherInfoType` structure to allow the CPM and the provider(s) to clean up.

**See Also** `CPMLibExportCipherInfo()`, `CPMLibReleaseCipherInfo()`

## CPMLibImportHashInfo Function

**Purpose** Initialize the contents of an `APHashInfoType` structure based upon a storable instance of that structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibImportHashInfo (uint8_t encoding,
    uint8_t *importDataP, uint32_t dataLen,
    APHashInfoType *hashInfoP)
```

**Parameters**

→ *encoding*

One of the encodings documented under "Import/Export Types" on page 237.

→ *importDataP*

Pointer to a buffer containing the raw data to be imported.

→ *dataLen*

The size, in bytes, of the buffer indicated by *importDataP*.

↔ *hashInfoP*

An `APHashInfoType` structure, allocated by the application, that holds information about the hashing operation.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibExportHashInfo()`

## CPMLibImportKeyInfo Function

**Purpose** Introduces an existing key to the CPM framework.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibImportKeyInfo (uint8_t encoding,
    uint8_t *importDataP, uint32_t dataLen,
    APKeyInfoType *keyInfoP)
```

**Parameters**

→ *encoding*
One of the values documented under "Import/Export Types" on page 237.

→ *importDataP*
Pointer to a buffer containing the raw data to be imported.

→ *dataLen*
The size of the buffer specified by *importDataP*.

↔ *keyInfoP*
An `APKeyInfoType` structure, allocated and optionally initialized by the application, that holds information about the key to be used for this operation.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** If no previous key exists, use `CPMLibGenerateKey()` to generate a new key.

The application is always responsible for allocating and freeing the `APKeyInfoType` structure. The application must call `CPMLibReleaseKeyInfo()` before freeing the `APKeyInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibExportKeyInfo()`, `CPMLibGenerateKey()`, `CPMLibReleaseKeyInfo()`

## CPMLibImportKeyPairInfo Function

**Purpose** Introduces an existing public/private key pair to the CPM framework.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibImportKeyPairInfo
    (uint8_t encoding, uint8_t *importDataP,
    uint32_t dataLen,
    APKeyInfoType *privateKeyInfoP,
    APKeyInfoType *publicKeyInfoP)
```

**Parameters**

→ *encoding*
One of the values documented under "Import/Export Types" on page 237.

→ *importDataP*
Pointer to a buffer containing the raw data to be imported.

→ *dataLen*
The size, in bytes, of the buffer indicated by *importDataP*.

↔ *privateKeyInfoP*
Pointer to an `APKeyInfoType` structure for the private key.

↔ *publicKeyInfoP*
Pointer to an `APKeyInfoType` structure for the public key.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` structures. The application must call `CPMLibReleaseKeyInfo()` before freeing an `APKeyInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibExportKeyPairInfo()`, `CPMLibReleaseKeyInfo()`

## CPMLibImportMACInfo Function

**Purpose** Initialize the contents of an `APMACInfoType` structure based upon a storable instance of that structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibImportMACInfo (uint8_t encoding,
    uint8_t *importDataP, uint32_t dataLen,
    APMACInfoType *macInfoP)
```

**Parameters**

→ *encoding*
One of the values documented under "Import/Export Types" on page 237.

→ *importDataP*
Pointer to a buffer containing the raw data to be imported.

→ *dataLen*
The size, in bytes, of the buffer indicated by *importDataP*.

↔ *macInfoP*
Pointer to an `APMACInfoType` structure, allocated by the application, to contain information about the message authentication context.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APMACInfoType` structure. The application must call `CPMLibReleaseMACInfo()` before freeing the `APMACInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibExportMACInfo()`, `CPMLibReleaseMACInfo()`

## CPMLibImportSignInfo Function

**Purpose** Initialize the contents of an `APSignInfoType` structure based upon a storable instance of that structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibImportSignInfo (uint8_t encoding,
    uint8_t *importDataP, uint32_t dataLen,
    APSignInfoType *signInfoP)
```

**Parameters**

→ *encoding*
One of the values documented under "Import/Export Types" on page 237.

→ *importDataP*
Pointer to a buffer containing the raw data to be imported.

→ *dataLen*
The size, in bytes, of the buffer indicated by *importDataP*.

↔ *signInfoP*
Pointer to an `APSignInfoType` structure, allocated by the application, to contain information about the signature context.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APSignInfoType` structure. The application must call `CPMLibReleaseSignInfo()` before freeing the `APSignInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibExportSignInfo()`, `CPMLibReleaseSignInfo()`

## CPMLibImportVerifyInfo Function

**Purpose** Initialize the contents of an `APVerifyInfoType` structure based upon a storable instance of that structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibImportVerifyInfo
    (uint8_t encoding, uint8_t *importDataP,
    uint32_t dataLen,
    APVerifyInfoType *verifyInfoP)
```

**Parameters**

→ *encoding*
One of the values documented under "Import/Export Types" on page 237.

→ *importDataP*
Pointer to a buffer containing the raw data to be imported.

→ *dataLen*
The size, in bytes, of the buffer indicated by *importDataP*.

↔ *verifyInfoP*
Pointer to an `APVerifyInfoType` structure, allocated by the application, to contain information about the verification context.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APVerifyInfoType` structure. The application must call `CPMLibReleaseVerifyInfo()` before freeing the `APVerifyInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibExportVerifyInfo()`, `CPMLibReleaseVerifyInfo()`

## CPMLibMAC Function

**Purpose** Performs the message authentication operation in one pass.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibMAC (APKeyInfoType *keyInfoP,
    APHashInfoType *hashInfoP, APMACEnum type,
    APMACInfoType *macInfoP, uint8_t *bufIn,
    uint32_t bufInLen, uint8_t *bufOut,
    uint32_t *bufOutLenP)
```

**Parameters**

↔ *keyInfoP*
: Pointer to an `APKeyInfoType` structure, allocated by the application, containing the key to be used for the operation.

↔ *hashInfoP*
: Pointer to an `APHashInfoType` structure, allocated by the application, that holds information about the hashing operation to be used for the operation.

→ *type*
: One of the values declared by the `APMACEnum` enum.

↔ *macInfoP*
: Pointer to an `APMACInfoType` structure, allocated by the application, to be used in subsequent calls to the same class of operations.

→ *bufIn*
: Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

→ *bufInLen*
: The size, in bytes, of the buffer specified by *bufIn*. This value must be greater than zero.

↔ *bufOut*
: Pointer to a buffer, allocated by the application, to receive the output of the operation.

↔ *bufOutLenP*
: The size, in bytes, of the buffer specified by the *bufOut* parameter.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType`, `APHashInfoType`, and `APMACInfoType` structures. The application must call the appropriate release function before freeing the structure to allow the CPM and provider to clean up.

**See Also** `CPMLibMACFinal()CPMLibMACInit()`, `CPMLibMACUpdate()`, `CPMLibReleaseKeyInfo()`, `CPMLibReleaseMACInfo()`, `CPMLibReleaseSignInfo()`

## CPMLibMACFinal Function

**Purpose** Finalizes a multi-part message authentication operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibMACFinal (APMACInfoType *macInfoP,
   uint8_t *bufIn, uint32_t bufInLen,
   uint8_t *bufOut, uint32_t *bufOutLenP)
```

**Parameters**

→ *macInfoP*
Pointer to the `APMACInfoType` structure that was initialized during the call to `CPMLibMACInit()`.

→ *bufIn*
Pointer to a buffer containing the data for the operation, or `NULL` if there is no additional data.

→ *bufInLen*
The size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
Pointer to a buffer, allocated by the application, to receive the output of the operation.

↔ *bufOutLenP*
The size, in bytes, of the buffer specified by the *bufOut* parameter.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under “CPM Library Error Codes” on page 239 otherwise.

**See Also** `CPMLibMACInit()`, `CPMLibMACUpdate()`, `CPMLibReleaseMACInfo()`

## CPMLibMACInit Function

**Purpose** Begins a multi-part message authentication operation with the specified key and hash info and returns the context of the operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibMACInit (APKeyInfoType *keyInfoP,
    APHashInfoType *hashInfoP,
    APMACInfoType *macInfoP)
```

**Parameters**

↔ *keyInfoP*
Pointer to an `APKeyInfoType` structure, allocated by the application, containing the key to be used in the subsequent calls to `CPMLibMACUpdate()` and `CPMLibMACFinal()`.

↔ *hashInfoP*
Pointer to an `APHashInfoType` structure, allocated by the application, that holds information about the hashing operation for use in the subsequent calls to `CPMLibMACUpdate()` and `CPMLibMACFinal()`.

↔ *macInfoP*
Pointer to an `APMACInfoType` structure, allocated by the application, to be used in subsequent calls to the same class of operations.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType`, `APHashInfoType`, and `APMACInfoType` structures. The application must call the appropriate release function before freeing the `APKeyInfoType` or `APMACInfoType` structure to allow the CPM and provider to clean up.

**See Also** `CPMLibMAC()`, `CPMLibMACFinal()`, `CPMLibMACUpdate()`, `CPMLibReleaseHashInfo()`, `CPMLibReleaseKeyInfo()`, `CPMLibReleaseMACInfo()`

## CPMLibMACUpdate Function

**Purpose** Updates a multi-part message authentication operation with more data.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibMACUpdate
   (APMACInfoType *macInfoP, uint8_t *bufIn,
   uint32_t bufInLen, uint8_t *bufOut,
   uint32_t *bufOutLenP)
```

**Parameters**

→ *macInfoP*
: Pointer to the `APMACInfoType` structure that was initialized during the call to `CPMLibMACInit()`.

→ *bufIn*
: Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

→ *bufInLen*
: The size, in bytes, of the buffer specified by *bufIn*. This value must be greater than zero.

↔ *bufOut*
: Pointer to a buffer, allocated by the application, to receive the output of the operation.

↔ *bufOutLenP*
: The size, in bytes, of the buffer specified by the *bufOut* parameter.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibSignFinal()`, `CPMLibSignInit()`

## CPMLibOpen Function

**Purpose** Handles the open of the CPM library.

**Declared In** `CPMLibARMInterface.h`

**Prototype** `status_t CPMLibOpen (uint16_t *numProviders)`

**Parameters**

← *numProviders*
: The number of providers the CPM currently knows about.

**Returns** `errNone` if the operation completed successfully, one of the errors listed under "Data Manager Error Codes" on page 112 of *Exploring Palm OS: Memory, Databases, Files* if the Data Manager couldn't open the library, or one of the following otherwise:

`cpmErrAlreadyOpen`
The library is already open.

`cpmErrOutOfMemory`
There wasn't enough memory to open the library.

`cpmErrNoProviders`
The CPM library is not aware of any providers. With no providers the CPM library has no functionality.

`cpmErrNoBaseProvider`
The CPM library cannot load the base provider.

**Comments** This function establishes the CPM application context to be used in future CPM calls. It also returns the number of providers the CPM currently knows about. This number should be 1 for the base provider plus any additional providers that may be installed. To enumerate those providers, use `CPMLibEnumerateProviders()`.

**See Also** `CPMLibClose()`, `CPMLibSleep()`, `CPMLibWake()`

## CPMLibReleaseCipherInfo Function

**Purpose** Allows the CPM and the provider(s) to clean up before the application frees the `APICipherInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibReleaseCipherInfo
    (APCipherInfoType *cipherInfoP)
```

**Parameters** ↔ *`cipherInfoP`*
Pointer to the `APCipherInfoType` structure.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## CPMLibReleaseHashInfo Function

**Purpose** Allows the CPM and the provider(s) to clean up before the application frees the `APIHashInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibReleaseHashInfo
    (APHashInfoType *hashInfoP)
```

**Parameters** ↔ *`hashInfoP`*
Pointer to the `APHashInfoType` structure.

**Comments** The application is not required to call this function; `APHashInfoType` structures need not be released.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## CPMLibReleaseKeyInfo Function

**Purpose** Allows the CPM and the provider(s) to clean up before the application frees the `APKeyInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibReleaseKeyInfo
    (APKeyInfoType *keyInfoP)
```

**Parameters** ↔ *`keyInfoP`*
Pointer to an `APKeyInfoType` structure.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## CPMLibReleaseMACInfo Function

**Purpose** Allows the CPM and the provider(s) to clean up before the application frees the `APMACInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibReleaseMACInfo
    (APMACInfoType *macInfoP)
```

**Parameters** ↔ *`macInfoP`*
Pointer to an `APMACInfoType` structure.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## CPMLibReleaseSignInfo Function

**Purpose** Allows the CPM and the provider(s) to clean up before the application frees the `APSignInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibReleaseSignInfo
    (APSignInfoType *signInfoP)
```

**Parameters** ↔ *`signInfoP`*
Pointer to an `APSignInfoType` structure.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## CPMLibReleaseVerifyInfo Function

**Purpose** Allows the CPM and the provider(s) to clean up before the application frees the `APVerifyInfoType` structure.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibReleaseVerifyInfo
    (APVerifyInfoType *verifyInfoP)
```

**Parameters** ↔ *`verifyInfoP`*
Pointer to an `APVerifyInfoType` structure.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## CPMLibSetDebugLevel Function

**Purpose** Specify the level of debug output to be sent from the library using `DbgMessage()`.

**Declared In** `CPMLibARMInterface.h`

**Prototype** `status_t CPMLibSetDebugLevel (uint8_t debugLevel)`

**Parameters** → *`debugLevel`*
The level of debug output to be sent. One of the values listed under "Debug Output Levels" on page 239.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## CPMLibSetDefaultProvider Function

**Purpose** Sets the default provider.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibSetDefaultProvider
    (uint32_t providerID)
```

**Parameters** → *`providerID`*
A provider ID referencing the provider that is to be the default provider.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The default provider is checked first for supporting operations rather than performing a linear search through the known providers. If an operation is not supported by the default provider, the CPM then checks the other providers.

**See Also** `CPMLibEnumerateProviders()`, `CPMLibGetProviderInfo()`

## CPMLibSign Function

**Purpose** Performs the signing operation in one pass.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibSign (APKeyInfoType *keyInfoP,
    APSignInfoType *signInfoP, uint8_t *bufIn,
    uint32_t bufInLen, uint8_t *bufOut,
    uint32_t *bufOutLenP, uint8_t *signature,
    uint32_t *signatureLenP)
```

**Parameters**

↔ *keyInfoP*
: Pointer to an `APKeyInfoType` structure, allocated by the application, containing the key to be used for this signing operation. Note that for this single-part operation, this structure is not required unless the application wants to pass setting information to or receive setting information from the CPM or provider.

↔ *signInfoP*
: Pointer to an `APSignInfoType` structure, allocated by the application, to be used for this signing operation. Note that for this single-part operation, this structure is not required unless the application wants to pass setting information to or receive setting information from the CPM or provider.

→ *bufIn*
: Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

→ *bufInLen*
: The size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
: Pointer to a buffer, allocated by the application, to receive the output of the operation.

↔ *bufOutLenP*
: The size, in bytes, of the buffer specified by the *bufOut* parameter.

↔ *signature*
: Pointer to a buffer, allocated by the application, to receive the calculated signature.

↔ *signatureLenP*
The size, in bytes, of the buffer specified by the *signature* parameter.

**Returns** errNone if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the APKeyInfoType and APSignInfoType structures. The application must call CPMLibReleaseKeyInfo() or CPMLibReleaseSignInfo(), as appropriate, before freeing the structure to allow the CPM and provider to clean up.

**See Also** CPMLibSignFinal()CPMLibSignInit(), CPMLibSignUpdate(), CPMLibReleaseKeyInfo(), CPMLibReleaseSignInfo()

## CPMLibSignFinal Function

**Purpose** Finalizes a multi-part signing operation.

**Declared In** CPMLibARMInterface.h

**Prototype**
```
status_t CPMLibSignFinal
   (APKeyInfoType *keyInfoP,
   APSignInfoType *signInfoP, uint8_t *bufIn,
   uint32_t bufInLen, uint8_t *bufOut,
   uint32_t *bufOutLenP, uint8_t *signature,
   uint32_t *signatureLenP)
```

**Parameters**

→ *keyInfoP*
Pointer to an APKeyInfoType structure containing the key to be used for the operation.

→ *signInfoP*
Pointer to the APSignInfoType structure that was initialized during the call to CPMLibSignInit().

→ *bufIn*
Pointer to a buffer containing the final data for the operation, or NULL if there is no additional data.

→ *bufInLen*
The size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
Pointer to a buffer, allocated by the application, to receive the output of the operation.

↔ *bufOutLenP*
The size, in bytes, of the buffer specified by the *bufOut* parameter.

↔ *signature*
Pointer to a buffer, allocated by the application, to receive the final calculated signature.

↔ *signatureLenP*
The size, in bytes, of the buffer specified by the *signature* parameter.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibSignInit()`, `CPMLibSignUpdate()`, `CPMLibReleaseKeyInfo()`, `CPMLibReleaseSignInfo()`

## CPMLibSignInit Function

**Purpose** Begins a multi-part signing operation with the specified key and returns the context of the signing operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibSignInit (APKeyInfoType *keyInfoP,
    APSignInfoType *signInfoP)
```

**Parameters**

→ *keyInfoP*
Pointer to an `APKeyInfoType` structure, allocated by the application, containing the key to be used for the subsequent calls to `CPMLibSignUpdate()` and `CPMLibSignFinal()`.

← *signInfoP*
Pointer to an `APSignInfoType` structure, allocated by the application, to be used in subsequent calls to the same class of operations.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` and `APSignInfoType` structures. The application must call `CPMLibReleaseKeyInfo()` or `CPMLibReleaseSignInfo()`, as appropriate, before freeing the structure to allow the CPM and provider to clean up.

The application must call `CPMLibSignFinal()` to finalize the operation.

**See Also** `CPMLibSign()`, `CPMLibSignFinal()`, `CPMLibSignUpdate()`, `CPMLibReleaseKeyInfo()`, `CPMLibReleaseSignInfo()`

## CPMLibSignUpdate Function

**Purpose** Updates a multi-part signing operation with more data.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibSignUpdate
    (APKeyInfoType *keyInfoP,
    APSignInfoType *signInfoP, uint8_t *bufIn,
    uint32_t bufInLen)
```

**Parameters**

→ *keyInfoP*
: Pointer to an `APKeyInfoType` structure containing the key to be used for the operation.

→ *signInfoP*
: Pointer to the `APSignInfoType` structure that was initialized during the call to `CPMLibSignInit()`.

→ *bufIn*
: Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

→ *bufInLen*
: The size, in bytes, of the buffer specified by *bufIn*. This value must be greater than zero.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibSignFinal()`, `CPMLibSignInit()`

## CPMLibSleep Function

**Purpose** Allows the library to handle the device going to sleep.

**Declared In** `CPMLibARMInterface.h`

**Prototype** `status_t CPMLibSleep (void)`

**Parameters** None.

**Returns** `errNone`.

**See Also** `CPMLibClose()`, `CPMLibOpen()`, `CPMLibWake()`

## CPMLibVerify Function

**Purpose** Performs the verify operation in one pass.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibVerify (APKeyInfoType *keyInfoP,
    APVerifyInfoType *verifyInfoP, uint8_t *bufIn,
    uint32_t bufInLen, uint8_t *bufOut,
    uint32_t *bufOutLenP, uint8_t *signature,
    uint32_t signatureLen,
    VerifyResultType *verifyResultP)
```

**Parameters**

↔ *keyInfoP*

Pointer to an `APKeyInfoType` structure, allocated by the application, containing the key to be used for this signing operation. Note that for this single-part operation, this structure is not required unless the application wants to pass setting information to or receive setting information from the CPM or provider.

↔ *verifyInfoP*

Pointer to an `APVerifyInfoType` structure, allocated by the application, to be used for this verify operation. Note that for this single-part operation, this structure is not required unless the application wants to pass setting information to or receive setting information from the CPM or provider.

→ *bufIn*

Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

→ *bufInLen*

The size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
Pointer to a buffer, allocated by the application, to receive the output of the operation.

↔ *bufOutLenP*
The size, in bytes, of the buffer specified by the *bufOut* parameter.

→ *signature*
Pointer to a buffer containing the previously calculated signature that is being verified.

→ *signatureLen*
The length, in bytes, of the buffer specified by *signature*.

← *verifyResultP*
Supply a pointer to a `VerifyResultType`. If the function call completed without error, upon return the `VerifyResultType` variable will be set to zero if the signature verifies or 1 if the signature did not verify.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` and `APVerifyInfoType` structures. The application must call `CPMLibReleaseKeyInfo()` or `CPMLibReleaseVerifyInfo()`, as appropriate, before freeing the structure to allow the CPM and provider to clean up.

**See Also** `CPMLibVerifyFinal()CPMLibVerifyInit()`, `CPMLibVerifyUpdate()`, `CPMLibReleaseKeyInfo()`, `CPMLibReleaseVerifyInfo()`

## CPMLibVerifyFinal Function

**Purpose** Finalizes a multi-part verification operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**

```
status_t CPMLibVerifyFinal
    (APKeyInfoType *keyInfoP,
    APVerifyInfoType *verifyInfoP, uint8_t *bufIn,
    uint32_t bufInLen, uint8_t *bufOut,
    uint32_t *bufOutLenP, uint8_t *signature,
    uint32_t signatureLen,
    VerifyResultType *verifyResultP)
```

**Parameters**

→ *keyInfoP*
Pointer to an `APKeyInfoType` structure containing the key to be used for the operation.

→ *verifyInfoP*
Pointer to the `APVerifyInfoType` structure that was initialized during the call to `CPMLibVerifyInit()`.

→ *bufIn*
Pointer to a buffer containing the final data for the operation, or `NULL` if there is no additional data.

→ *bufInLen*
The size, in bytes, of the buffer specified by *bufIn*.

↔ *bufOut*
Pointer to a buffer, allocated by the application, to receive the output of the operation.

↔ *bufOutLenP*
The size, in bytes, of the buffer specified by the *bufOut* parameter.

→ *signature*
Pointer to a buffer containing the previously calculated signature that is being verified.

→ *signatureLen*
The length, in bytes, of the buffer specified by *signature*.

← *verifyResultP*
Supply a pointer to a `VerifyResultType`. If the function call completed without error, upon return the `VerifyResultType` variable will be set to zero if the signature verifies or 1 if the signature did not verify.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibVerifyInit()`, `CPMLibVerifyUpdate()`, `CPMLibReleaseKeyInfo()`, `CPMLibReleaseVerifyInfo()`

## CPMLibVerifyInit Function

**Purpose** Begins a multi-part verification operation with the specified key and returns the context of the verification operation.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibVerifyInit
    (APKeyInfoType *keyInfoP,
    APVerifyInfoType *verifyInfoP)
```

**Parameters**

→ *keyInfoP*
: Pointer to an `APKeyInfoType` structure, allocated and optionally initialized by the application, containing the key to be used for the subsequent calls to `CPMLibVerifyUpdate()` and `CPMLibVerifyFinal()`.

← *verifyInfoP*
: Pointer to an `APVerifyInfoType` structure, allocated by the application, to be used in subsequent calls to `CPMLibVerifyUpdate()` and `CPMLibVerifyFinal()`.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` and `APVerifyInfoType` structures. The application must call `CPMLibReleaseKeyInfo()` or `CPMLibReleaseVerifyInfo()`, as appropriate, before freeing the structure to allow the CPM and provider to clean up.

The application must call `CPMLibVerifyFinal()` to finalize the operation.

**See Also** `CPMLibVerify()`, `CPMLibVerifyFinal()`, `CPMLibVerifyUpdate()`, `CPMLibReleaseKeyInfo()`, `CPMLibReleaseVerifyInfo()`

## CPMLibVerifyUpdate Function

**Purpose** Updates a multi-part verification operation with more data.

**Declared In** `CPMLibARMInterface.h`

**Prototype**
```
status_t CPMLibVerifyUpdate
    (APKeyInfoType *keyInfoP,
    APVerifyInfoType *verifyInfoP, uint8_t *bufIn,
    uint32_t bufInLen)
```

**Parameters**

→ *keyInfoP*
Pointer to an `APKeyInfoType` structure containing the key to be used for the operation.

→ *verifyInfoP*
Pointer to the `APVerifyInfoType` structure that was initialized during the call to `CPMLibVerifyInit()`.

→ *bufIn*
Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

→ *bufInLen*
The size, in bytes, of the buffer specified by *bufIn*. This value must be greater than zero.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMLibVerifyFinal()`, `CPMLibVerifyInit()`

## CPMLibWake Function

**Purpose** Allows the library to handle the device waking up.

**Declared In** `CPMLibARMInterface.h`

**Prototype** `status_t CPMLibWake (void)`

**Parameters** None.

**Returns** `errNone`.

**See Also** `CPMLibClose()`, `CPMLibOpen()`, `CPMLibSleep()`

# 10

# CPM Library Common Definitions

The `CPMLibCommon.h` header file declares the various structures, types, and constants used by the CPM library. These structures, types, and constants are documented in this chapter; this documentation is organized into the following sections:

# CPM Library Structures and Types

## APCipherInfoType Struct

**Purpose** Structure to hold information about a particular operation's context, including generic information about contexts and specific information about the provider's concept of the context.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APCipherInfoStruct {
   APProviderContextType providerContext;
   APAlgorithmEnum type;
   APPaddingEnum padding;
   APModeEnum mode;
   uint8_t *iv;
   uint32_t ivLength;
   void *algorithmParams;
}
typedef struct APCipherInfoStruct
APCipherInfoType, *APCipherInfoPtr
```

**Fields**

`providerContext`
: The provider context of this info type.

`type`
: Cipher type. One of the `APAlgorithmEnum` values.

`padding`
: The type of padding for this cipher. One of the `APPaddingEnum` values.

`mode`
: Modes of operation for symmetric encryption/decryption. One of the `APModeEnum` values.

`iv`
: Initialization vector as specified by the caller.

`ivLength`
: The size, in bytes, of the initialization vector.

`algorithmParams`
: Provider-specific algorithm parameters.

**Comments** The application is always responsible for allocating and freeing the `APCipherInfoType` structure. The application must also call `CPMLibReleaseCipherInfo()` before freeing the `APICipherInfoType` structure to allow the CPM and the provider(s) to clean up.

If the application does not care or wants default settings for the `APCipherInfoType`, allocate the `APCipherInfoType` structure and zero its contents. Upon return from either `CPMLibEncryptInit()`, `CPMLibEncrypt()`, `CPMLibDecryptInit()`, `CPMLibDecrypt()`, or `CPMLibImportCipherInfo()` the `APICipherInfoType` structure will have been filled in by the CPM and provider with the appropriate information.

if the application does care about the settings for the `APCipherInfoType`, the application should allocate the `APICipherInfoType` structure and set the fields as appropriate before passing it in to one of the previously mentioned functions.

## APDerivedKeyInfoType Struct

**Purpose** Structure to hold the various pieces of information about the derivation functions and parameters.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APDerivedKeyInfoStruct {
   APProviderContextType providerContext;
   APKeyDerivationEnum kdType;
   APKeyDerivationUsageEnum kdUsage;
   uint8_t *salt;
   uint32_t saltLen;
   uint32_t iterationCount;
   void *kdInfo;
}
typedef struct APDerivedKeyInfoStruct
APDerivedKeyInfoType, *APKeyDerivedKeyInfoPtr
```

**Fields** `providerContext`
The provider context of this info type. See `APProviderContextType`.

`kdType`
: The type of key derivation to be performed. One of the values specified by `APKeyDerivationEnum`.

`kdUsage`
: The allowed usage of the subsequently derived key. One of the values specified by `APKeyDerivationUsageEnum`.

`salt`
: Cryptographic salt to be used for the derivation function.

`saltLen`
: The length, in bytes, of the salt.

`iterationCount`
: The number of iterations for this particular derivation operation.

`kdInfo`
: Other derivation-function-specific parameters as defined by both the provider handling the derivation and the derivation function itself.

## APHashInfoType Struct

**Purpose** Structure to hold information about a particular hashing operation, including generic information about hashing operations and specific information about the provider's concept of the hashing operation.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APHashInfoStruct {
   APProviderContextType providerContext;
   APHashEnum type;
   uint32_t length;
}
typedef struct APHashInfoStruct APHashInfoType,
*APHashInfoPtr
```

**Fields**

`providerContext`
: The provider context of this info type. See `APProviderContextType`.

`type`
: The type of the hash. One of the values specified by `APHashEnum`.

`length`
: The length of the hash, in bytes.

**Comments** The application is responsible for allocating memory and deallocating memory for this structure.

## APKeyInfoType Struct

**Purpose** Structure to hold information about a particular key, including generic information about all keys and specific information about the provider's concept of the key.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APKeyInfoStruct {
   APProviderContextType providerContext;
   APAlgorithmEnum type;
   APKeyUsageEnum usage;
   APKeyClassEnum keyclass;
   uint32_t length;
   uint32_t actualLength;
   uint16_t exportable;
   uint16_t ephemeral;
}
typedef struct APKeyInfoStruct APKeyInfoType,
*APKeyInfoPtr
```

**Fields**

`providerContext`
: The provider context of this info type. See `APProviderContextType`.

`type`
: The type of this key. One of the values specified by `APAlgorithmEnum`.

`usage`
: The key usage. One of the values specified by `APKeyUsageEnum`.

`keyclass`
The key class. One of the values specified by `APKeyClassEnum`.

`length`

`actualLength`

`exportable`
Whether or not this key is exportable. This field has a value of 1 if it is, or 0 if it is not.

`ephemeral`
Whether or not this key is permanent. This field has a value of 1 if it is, or 0 if it is not.

**Comments** The application is always responsible for allocating and freeing the `APKeyInfoType` structure. The application must call `CPMLibReleaseKeyInfo()` before freeing the `APKeyInfoType` structure to allow the CPM and provider to clean up.

If the application does not care or wants default settings for the `APKeyInfoType`, the application should allocate the `APKeyInfoType` structure and set its contents to zero. Upon return from either `CPMLibImportKeyInfo()` or `CPMLibGenerateKey()` the `APKeyInfoType` structure would have been filled in by the CPM and the provider with the appropriate information.

If the application does care about the settings for the `APKeyInfoType`, the application should allocate the `APKeyInfoType` structure and set the fields appropriately before passing it in to `CPMLibGenerateKey()`.

## APMACInfoType Struct

**Purpose**

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APMACInfoStruct {
   APProviderContextType providerContext;
   APMACEnum type;
}
typedef struct APMACInfoStruct APMACInfoType,
*APMACInfoPtr
```

**Fields**

`providerContext`
The provider context of this info type. See `APProviderContextType`.

`type`
The MAC type. One of the `APMACEnum` values.

## APProviderContextType Struct

**Purpose** Contains provider-specific information.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APProviderContextStruct {
   uint32_t providerID;
   void *localContext;
}
typedef struct APProviderContextStruct
APProviderContextType, *APProviderContextPtr
```

**Fields**

`providerID`
The provider handling this operation.

`localContext`
Provider-specific infomation about this operation.

## APProviderInfoType Struct

**Purpose** Structure to hold information about a particular provider as it is known by the current instantiation of the CPM library.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APProviderInfoStruct {
   char name[32];
   char other[64];
   uint32_t flags;
   uint8_t numAlgorithms;
   Boolean bHardware;
}
typedef struct APProviderInfoStruct
APProviderInfoType, *APProviderInfoPtr
```

**Fields**

`name`
Name of the provider.

`other`
Other textual information.

`flags`
Flags to indicate functionality supported by the cryptographic provider. This is a combination of the values documented under "Cryptographic Provider Functionality Flags" on page 238.

`numAlgorithms`
Number of algorithms supported.

`bHardware`
Whether or not this is a hardware provider.

## APSignInfoType Struct

**Purpose** Structure to hold information about a particular operation's signature context, including generic information about signature contexts and specific information about the provider's concept of the signature context. This structure is needed to maintain state across the various multi-part operations.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APSignInfoStruct {
   APProviderContextType providerContext;
   APHashInfoType *hashInfoP;
   APCipherInfoType *cipherInfoP;
}
typedef struct APSignInfoStruct APSignInfoType,
*APSignInfoPtr
```

**Fields**

`providerContext`
The provider context of this info type. See `APProviderContextType`.

`hashInfoP`
The hash to use. See `APHashInfoType`.

`cipherInfoP`
The cipher to use. See `APCipherInfoType`.

**Comments** The application is always responsible for allocating and freeing the `APSignInfoType` structure and the associated `APCipherInfoType` and `APHashInfoType` structures. The application must call `CPMLibReleaseCipherInfo()` with the `APCipherInfoType` structure to allow the CPM and the provider(s) to clean up the `APCipherInfoType` structure. The application must call `CPMLibReleaseHashInfo()` with the `APHashInfoType` structure to allow the CPM and the provider(s) to clean up the `APHashInfoType` structure.

If the application does not care or wants default settings for either the `APCipherInfoType` or the `APHashInfoType`, allocate and zero the structures. Upon return from `CPMLibSign()` or `CPMLibSignInit()` the `APCipherInfoType` structure and the `APHashInfoType` structure would have been filled in by the CPM and the provider(s) with the appropriate information.

If the application does care about the settings for either the `APCipherInfoType` or the `APHashInfoType`, allocate the structures and set the fields appropriately before calling `CPMLibSign()` or `CPMLibSignInit()`. The application may set the fields in one or the other or both.

## APVerifyInfoType Struct

**Purpose** Structure to hold information about a particular operation's verification context, including generic information about verification contexts and specific information about the provider's concept of the verification context.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct APVerifyInfoStruct {
   APProviderContextType providerContext;
   APHashInfoType *hashInfoP;
   APCipherInfoType *cipherInfoP;
}
typedef struct APVerifyInfoStruct
APVerifyInfoType, *APVerifyInfoPtr
```

**Fields**

`providerContext`
: The provider context of this info type. See `APProviderContextType`.

`hashInfoP`
: The hash to use. See `APHashInfoType`.

`cipherInfoP`
: The cipher to use. See `APCipherInfoType`.

**Comments** The application is always responsible for allocating and freeing the `APSignInfoType` structure and the associated `APCipherInfoType` and `APHashInfoType` structures. The application must call `CPMLibReleaseCipherInfo()` with the `APCipherInfoType` structure to allow the CPM and the provider(s) to clean up the `APCipherInfoType` structure. The application must call `CPMLibReleaseHashInfo()` with the `APHashInfoType` structure to allow the CPM and the provider(s) to clean up the `APHashInfoType` structure.

If the application does not care or wants default settings for either the `APCipherInfoType` or the `APHashInfoType`, allocate and zero the structures. Upon return from `CPMLibVerify()` or `CPMLibVerifyInit()` the `APCipherInfoType` structure and the `APHashInfoType` structure would have been filled in by the CPM and the provider(s) with the appropriate information.

If the application does care about the settings for either the `APCipherInfoType` or the `APHashInfoType`, allocate the structures and set the fields appropriately before calling `CPMLibVerify()` or `CPMLibVerifyInit()`. The application may set the fields in one or the other or both.

## CPMInfoType Struct

**Purpose** Structure to hold information about the CPM library as it is known by the currently running system.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
struct CPMInfoStruct {
   uint8_t numInstances;
   uint8_t numProviders;
   Boolean defaultProviderPresent;
}
typedef struct CPMInfoStruct CPMInfoType,
*CPMInfoPtr
```

**Fields**

`numInstances`
Number of instances of this library.

`numProviders`
Number of providers this library knows about.

`defaultProviderPresent`
Whether or not the default provider is known.

## VerifyResultType Typedef

**Purpose** Type that holds the result of a verification operation.

**Declared In** `CPMLibCommon.h`

**Prototype**

```
typedef uint32_t VerifyResultType,
*VerifyResultPtr
```

**Comments** This type is used by `CPMLibVerify()` and `CPMLibVerifyFinal()`. These functions set a variable of this type to zero if the signature verifies, or to 1 if the signature did not verify.

# CPM Library Constants

## APAlgorithmEnum Typedef

**Purpose** Cipher types. Used by the `APCipherInfoType` and `APKeyInfoType` structures.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APAlgorithmEnum`

**Constants** `#define apAlgorithmTypeUnspecified 0x00L`
The algorithm type is unspecified.

### Block Ciphers

```
#define apSymmetricTypeDES 0x01L
#define apSymmetricTypeRC2 0x02L
#define apSymmetricTypeRC4 0x03L
#define apSymmetricTypeRC5 0x04L
#define apSymmetricTypeRC6 0x05L
#define apSymmetricTypeDESX_XDX3 0x06L
#define apSymmetricType3DES_EDE2 0x07L
#define apSymmetricType3DES_EDE3 0x08L
#define apSymmetricTypeIDEA 0x09L
#define apSymmetricTypeDiamond2 0x0aL
#define apSymmetricTypeBlowfish 0x0bL
#define apSymmetricTypeTEA 0x0cL
#define apSymmetricTypeSAFER 0x0dL
#define apSymmetricType3WAY 0x0eL
#define apSymmetricTypeGOST 0x0fL
#define apSymmetricTypeSHARK 0x10L
#define apSymmetricTypeCAST128 0x11L
#define apSymmetricTypeSquare 0x12L
```

```
#define apSymmetricTypeSkipjack 0x13L
```

#### Stream Ciphers

```
#define apSymmetricTypePanama 0x14L
#define apSymmetricTypeARC4 0x15L
#define apSymmetricTypeSEAL 0x16L
#define apSymmetricTypeWAKE 0x17L
#define apSymmetricTypeSapphire 0x18L
#define apSymmetricTypeBBS 0x19L
```

#### AES Block Ciphers

```
#define apSymmetricTypeRijndael 0x2aL
#define apSymmetricTypeCAST256 0x2bL
#define apSymmetricTypeTwofish 0x2cL
#define apSymmetricTypeMARS 0x2dL
#define apSymmetricTypeSerpent 0x2eL
```

#### Asymmetric Key Types

```
#define apAsymmetricTypeRSA 0x2fL
#define apAsymmetricTypeDSA 0x30L
#define apAsymmetricTypeElgamal 0x31L
#define apAsymmetricTypeNR 0x32L
```

Nyberg-Rueppel

```
#define apAsymmetricTypeBlumGoldwasser 0x33L
#define apAsymmetricTypeRabin 0x34L
#define apAsymmetricTypeRW 0x35L
```

Rabin-Williams

```
#define apAsymmetricTypeLUC 0x36L
#define apAsymmetricTypeLUCELG 0x37L
```

#### Elliptic Curve

```
#define apAsymmetricTypeECDSA 0x38L
```

`#define apAsymmetricTypeECNR 0x39L`

`#define apAsymmetricTypeECIES 0x3aL`

`#define apAsymmetricTypeECDHC 0x3bL`

`#define apAsymmetricTypeECMQVC 0x3cL`

#### Key Agreement

`#define apKeyAgreementTypeDH 0x3dL`

`#define apKeyAgreementTypeDH2 0x3eL`
Unified Diffie-Hellman

`#define apKeyAgreementTypeMQV 0x3fL`
Menezes-Qu-Vanstone

`#define apKeyAgreementTypeLUCDIF 0x40L`

`#define apKeyAgreementTypeXTRDH 0x41L`

## APHashEnum Typedef

**Purpose** Algorithm provider hash types. Used by the `APHashInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APHashEnum`

**Constants** `#define apHashTypeHAVAL 0x05L`

`#define apHashTypeMD2 0x02L`

`#define apHashTypeMD5 0x03L`

`#define apHashTypeNone 0x01L`

`#define apHashTypePanama 0x08L`

`#define apHashTypeRIPEMD160 0x06L`

```
#define apHashTypeSHA1 0x04L

#define apHashTypeSHA256 0x09L

#define apHashTypeSHA384 0x0aL

#define apHashTypeSHA512 0x0bL

#define apHashTypeTiger 0x07L

#define apHashTypeUnspecified 0x00L
```

## APKeyClassEnum Typedef

**Purpose** The key class. Used by the `APKeyInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APKeyClassEnum`

**Constants**

```
#define apKeyClassPrivate 0x03L

#define apKeyClassPublic 0x02L

#define apKeyClassSymmetric 0x01L

#define apKeyClassUnspecified 0x00L
```

## APKeyDerivationEnum Typedef

**Purpose** The type of key derivation to be performed. Used by the `APDerivedKeyInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APKeyDerivationEnum`

**Constants** `#define apKeyDerivationTypePKCS12 0x03L`

`#define apKeyDerivationTypePKCS5v1 0x01L`

`#define apKeyDerivationTypePKCS5v2 0x02L`

`#define apKeyDerivationTypePKIX 0x04L`

`#define apKeyDerivationTypeTLS 0x05L`

`#define apKeyDerivationUnspecified 0x00L`

## APKeyDerivationUsageEnum Typedef

**Purpose** The allowed usage of the subsequently derived key. Used by the `APDerivedKeyInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APKeyDerivationUsageEnum`

**Constants** `#define apKeyDerivationUsageEncryption 0x01L`

`#define apKeyDerivationUsageIV 0x03L`

`#define apKeyDerivationUsageMAC 0x02L`

`#define apKeyDerivationUsageUnspecified 0x00L`

## APKeyUsageEnum Typedef

**Purpose** The key usage. Used by the `APKeyInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APKeyUsageEnum`

**Constants**

```
#define apKeyUsageAll 0x01L

#define apKeyUsageCertificateSigning 0x04L

#define apKeyUsageDataEncrypting 0x06L

#define apKeyUsageEncryption 0x03L

#define apKeyUsageKeyEncrypting 0x05L

#define apKeyUsageMessageIntegrity 0x07L

#define apKeyUsageSigning 0x02L

#define apKeyUsageUnspecified 0x00L
```

## APMACEnum Typedef

**Purpose** The MAC type. Used by the `APMACInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APMACEnum`

**Constants**

```
#define apMACHMAC 0x01L

#define apMACUnspecified 0x00L
```

## APModeEnum Typedef

**Purpose** Modes of operation for symmetric encryption/decryption. Used by the `APCipherInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APModeEnum`

**Constants**

```
#define apModeCounter 0x07L

#define apModeTypeCBC 0x03L

#define apModeTypeCBC_CTS 0x04L

#define apModeTypeCFB 0x05L

#define apModeTypeECB 0x02L

#define apModeTypeNone 0x01L

#define apModeTypeOFB 0x06L

#define apModeTypeUnspecified 0x00L
```

## APPaddingEnum Typedef

**Purpose** The type of padding for a cipher. Used by the `APCipherInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Prototype** `typedef uint32_t APPaddingEnum`

**Constants**

```
#define apPaddingTypeNone 0x01L

#define apPaddingTypeOAEP 0x05L
```

```
#define apPaddingTypePKCS1Type1 0x02L

#define apPaddingTypePKCS1Type2 0x03L

#define apPaddingTypePKCS5 0x04L

#define apPaddingTypeSSLv23 0x06L

#define apPaddingTypeUnspecified 0x00L
```

## Import/Export Types

**Purpose** Indicates the type of encoding to use when importing or exporting using one of the `CPMLibImport...` or `CPMLibExport...` functions.

**Declared In** `CPMLibCommon.h`

**Constants** `#define IMPORT_EXPORT_TYPE_DER 1`
A standardized ASN.1 DER encoding

`#define IMPORT_EXPORT_TYPE_RAW 0`
A raw form of import/export as defined by the provider.

`#define IMPORT_EXPORT_TYPE_XML 2`
A standardized XML encoding.

**Comments** A given CPM import/export format, such as XML, is only supported if the provider supports it. `IMPORT_EXPORT_TYPE_RAW` is always supported.

## Cryptographic Provider Functionality Flags

**Purpose** Flags that identify the functionality provided by a given cryptographic provider. The appropriate flags are ORd together to make up the flags field of the `APProviderInfoType` structure.

**Declared In** `CPMLibCommon.h`

**Constants**

`#define APF_CIPHER 0x00000080`
The provider supports encryption and decryption, import and export

`#define APF_HASH 0x00000040`
The provider supports message digests.

`#define APF_HW 0x00000002`
The provider is implemented in hardware (SmartCard).

`#define APF_KEYDERIVE 0x00000020`
The provider supports key derivation, import and export.

`#define APF_KEYGEN 0x00000004`
The provider supports key generation, import and export.

`#define APF_KEYPAIRGEN 0x00000010`
The provider supports key pair generation, import and export.

`#define APF_MAC 0x00000400`
The provider supports MAC.

`#define APF_MP 0x00000001`
Multiple-part operations are supported (Init, Update, Final).

`#define APF_SIGN 0x00000100`
The provider supports signing.

`#define APF_VERIFY 0x00000200`
The provider supports verification.

## Debug Output Levels

**Purpose** Values that specify the level of debug output to be sent from the library using `DbgMessage()`. Set the debug output level by calling `CPMLibSetDebugLevel()`.

**Declared In** `CPMLibCommon.h`

**Constants**

`#define LOG_ALERT 1`
Action must be taken immediately.

`#define LOG_CRIT 2`
Critical conditions.

`#define LOG_DEBUG 7`
Debug-level messages.

`#define LOG_EMERG 0`
The system is unusable.

`#define LOG_ERR 3`
Error conditions.

`#define LOG_INFO 6`
Informational.

`#define LOG_NOTICE 5`
Normal but significant condition.

`#define LOG_WARNING 4`
Warning conditions.

## CPM Library Error Codes

**Purpose** Error codes returned by the various CPM library functions.

**Declared In** `CPMLibCommon.h`

**Constants**

`#define cpmErrAlreadyOpen (cpmErrorClass | 1)`
The CPM library is already open. Usually returned from `CPMLibOpen()` indicating that the library is already open.

`#define cpmErrBadData (cpmErrorClass | 14)`
Data passed to the CPM library was no good.

`#define cpmErrBufTooSmall (cpmErrorClass | 13)`
A buffer passed to the CPM library was too small.

`#define cpmErrKeyExists (cpmErrorClass | 18)`
The key you are trying to import already seems to exist.

`#define cpmErrKeyNotFound (cpmErrorClass | 19)`
The key you are trying to use doesn't seem to exist.

`#define cpmErrNoAppContext (cpmErrorClass | 17)`
The CPM application context could not be found for this operation. Most likely, the CPM library is not open.

`#define cpmErrNoBaseProvider (cpmErrorClass | 5)`
The CPM library cannot load the base provider.

`#define cpmErrNoProviders (cpmErrorClass | 4)`
The CPM library is not aware of any providers. With no providers the CPM library has no functionality.

`#define cpmErrNotOpen (cpmErrorClass | 2)`
The CPM library is not open.

`#define cpmErrOutOfMemory (cpmErrorClass | 12)`
The CPM library is out of dynamic heap.

`#define cpmErrOutOfResources (cpmErrorClass | 11)`
The CPM library is out of resources (such as memory, static heap, and so on).

`#define cpmErrParamErr (cpmErrorClass | 10)`
A CPM library call was made with an invalid parameter.

`#define cpmErrProviderNotFound (cpmErrorClass | 6)`
The CPM library cannot find the specified provider.

`#define cpmErrStillOpen (cpmErrorClass | 3)`
The CPM library is still open after a call to `CPMLibClose()`.

`#define cpmErrUnimplemented (cpmErrorClass | 15)`
A CPM library function is not implemented.

`#define cpmErrUnsupported (cpmErrorClass | 16)`
A CPM library function is unsupported in the current version.

## Miscellaneous CPM Library Constants

**Purpose** The CPMLibCommon.h header file also declares these constants.

**Declared In** `CPMLibCommon.h`

**Constants**

`#define cpmCreator 'cpml'`
: The CPM creator ID. Used for both the databae that contains the Cryptographic Provider Manager Library and its preferences database.

`#define cpmFtrCreator cpmCreator`
: The feature creator. Intended for use with `FtrGet()`.

`#define cpmFtrNumVersion 0`
: The number of the CPM feature that contains the current version of the CPM library. The value returned from this feature has the following format: `0x`*MMmfsbbb*, where *MM* is the major version, *m* is the minor version, *f* is the bug fix level, *s* is the build stage (3 = release, 2 = beta, 1 = alpha, 0 = development), and *bbb* is the build number for non-releases

# 11

# CPM Library Provider



The header file `CPMLibProvider.h` declares the API that this chapter describes.

## CPM Library Provider Structures and Types

### CPMCallerInfoType Struct

**Purpose** This structure is passed to the provider's dispatcher. It contains all the information necessary for the provider to call back into the CPM framework and use the CPM's resources.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
typedef struct _CPMCallerInfoType {
   void *appContext;
   CPMDebugOutProcPtr debugout;
   CPMDispatcherProcPtr dispatcher;
   CPMGenerateRandomBytesProcPtr generateRandom;
   CPMAddRandomSeedProcPtr addSeed;
} CPMCallerInfoType
```

```
typedef CPMCallerInfoType *CPMCallerInfoPtr;
```

**Fields**

`appContext`
: Pointer to a structure containing the provider-specific application context to be used in future CPM calls. This structure is normally created and initialized when the library is opened.

`debugout`
: Pointer to a function that outputs a debug message to the debug device. See `CPMDebugOutProcPtr()`

`dispatcher`
: Pointer to the CPM function dispatcher. See `CPMDispatcherProcPtr()`.

`generateRandom`
: Pointer to a function that can return a requested number of random bytes. See `CPMGenerateRandomBytesProcPtr()`.

`addSeed`
: Pointer to a function that puts a number of seed bytes into the pseudo-random number generator maintained by the CPM. See `CPMAddRandomSeedProcPtr()`

# CPM Library Provider Function Argument Structures

## APCmdPBType Struct

**Purpose** Declares the algorithm provider interface command parameter blocks.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
typedef union APCmdPBType {
   ...
} APCmdPBType, *APCmdPBPtr
```

**Fields** Various structures. Each structure is documented separately within this section.

## APDecrypt Struct

**Purpose** The command paramater block for the `Decrypt` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APCipherInfoType *cipherInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APDecrypt;
```

**Fields**

`keyInfoP`
: The key to be used for the operation.

`cipherInfoP`
: Avalid context for this operation initialized by the initialization operation.

`bufIn`
: The data to add to the operation.

`bufInLen`
: The length of the data to add to this operation.

`bufOut`
: The buffer where the results of the operation are to be placed.

`bufOutLenP`
: The size of the buffer specified by `bufOut`. Also returns the actual number of bytes placed in `bufOut` on return.

**Comments** The `cipherInfoP` field specifies an initialized context which was created in the initialization of this class of functions.

`bufInLen` may not be zero and `bufIn` must be valid.

The provider performs a complete decryption operation using the data in `bufIn` of length `bufInLen` and puts the results in `bufOut`. The actual number of bytes placed in `bufOut` is placed in `bufOutLenP`.

## APDecryptFinal Struct

**Purpose** The command paramater block for the `DecryptFinal` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APCipherInfoType *cipherInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APDecryptFinal;
```

**Fields**

`keyInfoP`
The key to be used for the operation.

`cipherInfoP`
Avalid context for this operation initialized by the initialization operation.

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

`bufOut`
The buffer where the results of the operation are to be placed.

`bufOutLenP`
The size of the buffer specified by `bufOut`. Also returns the actual number of bytes placed in `bufOut` on return.

**Comments** The `cipherInfoP` field specifies an initialized context which was created in the initialization of this class of functions.

`bufInLen` may not be zero and `bufIn` must be valid.

The provider performs a complete decryption operation using the data in `bufIn` of length `bufInLen` and puts the results in `bufOut`. The actual number of bytes placed in `bufOut` is placed in `bufOutLenP`.

## APDecryptInit Struct

**Purpose** The command paramater block for the `DecryptInit` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APCipherInfoType *cipherInfoP;
} APDecryptInit;
```

**Fields**

`keyInfoP`
: The key to be used for the operation.

`cipherInfoP`
: A handle to a context to pass to the other members of the Decrypt functions.

**Comments** The provider is responsible for initializing a context of the appropriate type with the specified `keyInfoP` for the operation and saving it in `cipherInfoP`. This Init operation must be concluded with a single finalization operation with zero or more update operations in between.

## APDecryptUpdate Struct

**Purpose** The command paramater block for the `DecryptUpdate` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APCipherInfoType *cipherInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APDecryptUpdate;
```

**Fields**

`keyInfoP`
: The key to be used for the operation.

`cipherInfoP`
: Avalid context for this operation initialized by the initialization operation.

`bufIn`
: The data to add to the operation.

`bufInLen`
: The length of the data to add to this operation.

`bufOut`
: The buffer where the results of the operation are to be placed.

`bufOutLenP`
: The size of the buffer specified by `bufOut`. Also returns the actual number of bytes placed in `bufOut` on return.

**Comments** The `cipherInfoP` field specifies an initialized context which was created in the initialization of this class of functions.

If `bufInLen` is non-zero, the provider uses `cipherInfoP` to update the operation with the data in `bufIn` of length `bufInLen`.

The provider finalizes the operation which was initialized by the Init operation and places the results of the operation in `bufOut`. The provider updates `bufOutLenP` with the actual number of bytes placed in `bufOut`. The provider is responsible for cleaning up the `cipherInfoP` parameter.

## APDeriveKeyData Struct

**Purpose** The command paramater block for the `DeriveKeyData` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyDerivationEnum kdType;
   APKeyDerivationUsageEnum kdUsage;
   uint8_t *saltP;
   uint32_t saltLen;
   uint32_t iterationCount;
   void *kdInfo;
   uint8_t *keyDataP;
   uint32_t *keyDataLenP;
} APDeriveKeyData;
```

**Fields**

`kdType`
: The type of key derivation to be performed. One of the values specified by `APKeyDerivationEnum`.

`kdUsage`
: The allowed usage of the subsequently derived key. One of the values specified by `APKeyDerivationUsageEnum`.

`saltP`
: Cryptographic salt to be used for the derivation function.

`saltLen`
: The length, in bytes, of the salt.

`iterationCount`
: The number of iterations for this particular derivation operation.

`kdInfo`
: Other derivation-function-specific parameters as defined by both the provider handling the derivation and the derivation function itself.

`keyDataP`
: Pointer to a buffer into which the derived key data is written. Pass `NULL` to determine how large this buffer should be.

`keyDataLenP`
: When calling `DeriveKeyData`, set the variable to which this parameter points to the size of the *`keyDataP`* buffer. Upon return, the variable will be set to the number of bytes written to *`keyDataP`*. If you set *`keyDataP`* to `NULL`, set this variable to 0.

## APEncrypt Struct

**Purpose** The command paramater block for the `Encrypt` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APCipherInfoType *cipherInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APEncrypt;
```

**Fields**

`keyInfoP`
The key to be used for the operation.

`cipherInfoP`
A valid context for this operation initialized by the initialization operation.

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

`bufOut`
The buffer where the results of the operation are to be placed.

`bufOutLenP`
The size of the buffer specified by the `bufOut` parameter. Also returns the actual number of bytes placed in `bufOut` on return.

**Comments** The `cipherInfoP` parameter specifies an initialized context which was created during the initialization of this class of functions.

The parameter `bufInLen` may not be zero and `bufIn` must be valid.

The provider performs a complete encryption operation using the data in `bufIn` of length `bufInLen` and puts the results in `bufOut`. The actual number of bytes placed in `bufOut` is placed in `bufOutLenP`.

## APEncryptFinal Struct

**Purpose** The command paramater block for the `EncryptFinal` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APCipherInfoType *cipherInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APEncryptFinal;
```

**Fields**

`keyInfoP`
The key to be used for the operation.

`cipherInfoP`
A valid context for this operation initialized by the initialization operation.

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

`bufOut`
The buffer where the results of the operation are to be placed.

`bufOutLenP`
The size of the buffer specified by the `bufOut` parameter. Also returns the actual number of bytes placed in `bufOut` on return

**Comments** The `cipherInfoP` parameter specifies an initialized context which was created in the Init of this class of functions.

If the `bufInLen` is non-zero, the provider uses the `cipherInfoP` to update the operation with the data in `bufIn` of length `bufInLen`.

The provider finalizes the operation which was initialized by the Init operation and places the results of the operation in `bufOut`. The provider updates `bufOutLenP` with the actual number of bytes placed in `bufOut`. The provider is responsible for cleaning up the `cipherInfoP` parameter.

## APEncryptInit Struct

**Purpose** The command paramater block for the `EncryptInit` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APCipherInfoType *cipherInfoP;
} APEncryptInit;
```

**Fields**

`keyInfoP`
: The key to be used for the operation.

`cipherInfoP`
: A handle to a context to pass to the other members of the Encrypt functions.

**Comments** The provider is responsible for initializing a context of the appropriate `type` with the specified `keyInfoP` for the operation and saving it in `cipherInfoP`. This initialization operation must be concluded with a single finalization operation with zero or more update operations in between.

## APEncryptUpdate Struct

**Purpose** The command paramater block for the `EncryptUpdate` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APCipherInfoType *cipherInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APEncryptUpdate;
```

**Fields**

`keyInfoP`
: The key to be used for the operation.

`cipherInfoP`
: A valid context for this operation initialized by the initialization operation.

`bufIn`
: The data to add to the operation.

`bufInLen`
: The length of the data to add to this operation.

`bufOut`
: The buffer where the results of the operation are to be placed.

`bufOutLenP`
: The size of the buffer specified by the `bufOut` parameter. Also returns the actual number of bytes placed in `bufOut` on return

**Comments** The `cipherInfoP` parameter specifies an initialized context which was created by the initialization of this class of functions. The provider uses `cipherInfoP` to update the operation with the data in `bufIn` of length `bufInLen`. The provider also updates `cipherInfoP` to reflect the update. Any number of update operations (including zero) may occur between an initialization and a finalization operation. The parameters `bufIn` and `bufInLen` must be valid for the operation or an error is returned.

## APExportCipherInfo Struct

**Purpose** The command paramater block for the `ExportCipherInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *exportDataP;
   uint32_t *dataLenP;
   APCipherInfoType *cipherInfoP;
} APExportCipherInfo;
```

**Fields**

`encoding`
: The encoding of the data being exported.

`exportDataP`
: The data for the export operation.

`dataLenP`
: Length of the exported data.

`cipherInfoP`
A valid context for this operation initialized by the initialization operation.

## APExportHashInfo Struct

**Purpose** The command paramater block for the `ExportHashInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *exportDataP;
   uint32_t *dataLenP;
   APHashInfoType *hashInfoP;
} APExportHashInfo;
```

**Fields**

`encoding`
The encoding of the data being exported.

`exportDataP`
The data for the export operation.

`dataLenP`
Length of the exported data.

`hashInfoP`
An `APHashInfoType` structure, allocated by the application, that holds information about the hashing operation.

## APExportKeyInfo Struct

**Purpose** The command paramater block for the `ExportKey` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   APKeyInfoType *keyInfoP;
   uint8_t *exportDataP;
   uint32_t *dataLenP;
} APExportKeyInfo;
```

**Fields**

`encoding`
The encoding desired for the export data.

`keyInfoP`
Pointer to the key being exported.

`exportDataP`
The data for the export operation.

`dataLenP`
Length of the exported data.

## APExportKeyPairInfo Struct

**Purpose** The command paramater block for the `ExportKeyPair` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *exportDataP;
   uint32_t *dataLenP;
   APKeyInfoType *privateKeyInfoP;
   APKeyInfoType *publicKeyInfoP;
} APExportKeyPairInfo;
```

**Fields**

`encoding`
The encoding of the data being exported.

`exportDataP`
The data for the export operation.

`dataLenP`
Length of the exported data.

`privateKeyInfoP`
: Pointer to the APKeyInfoType structure for the private key.

`publicKeyInfoP`
: Pointer to the APKeyInfoType structure for the public key.

## APExportMacInfo Struct

**Purpose** The command paramater block for the `ExportMAC` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *exportDataP;
   uint32_t *dataLenP;
   APMACInfoType *macInfoP;
} APExportMACInfo;
```

**Fields**

`encoding`
: The encoding of the data being exported.

`exportDataP`
: The data for the export operation.

`dataLenP`
: Length of the exported data.

`macInfoP`
: Pointer to an APMACInfoType structure.

## APExportSignInfo Struct

**Purpose** The command paramater block for the `ExportSignInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *exportDataP;
   uint32_t *dataLenP;
   APSignInfoType *signInfoP;
} APExportSignInfo;
```

**Fields**

`encoding`
: The encoding of the data being exported.

`exportDataP`
The data for the export operation.

`dataLenP`
Length of the exported data.

`signInfoP`
A valid context for this operation initialized by the initialization operation.

## APExportVerifyInfo Struct

**Purpose** The command paramater block for the `ExportVerifyInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *exportDataP;
   uint32_t *dataLenP;
   APVerifyInfoType *verifyInfoP;
} APExportVerifyInfo;
```

**Fields**

`encoding`
The encoding of the data being exported.

`exportDataP`
The data for the export operation.

`dataLenP`
Length of the exported data.

`verifyInfoP`
A valid context for this operation initialized by the initialization operation.

## APGenerateKey Struct

**Purpose** The command paramater block for the `GenerateKey` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t *keyDataP;
   uint32_t dataLen;
   APKeyInfoType *keyInfoP;
} APGenerateKey;
```

**Fields**

`keyDataP`
Pointer to a buffer of seed bytes to be used by the pseudo-random number generator, or `NULL` to have the pseudo-random number generator use the seed data it already has.

`dataLen`
Length of the `keyDataP` data.

`keyInfoP`
An opaque handle to the key generated by the provider.

**Comments** If the newly-generated key is utilized for any cryptographic operations, it must be exported and saved in order to be used again. It is statistically improbably that a generated key could be regenerated.

The provider generates a key of the specified type and returns an opaque handle the key in *`keyInfoP`*.

## APGenerateKeyPair Struct

**Purpose** The command paramater block for the `GenerateKeyPair` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t *keyDataP;
   uint32_t dataLen;
   APKeyInfoType *privateKeyInfoP;
   APKeyInfoType *publicKeyInfoP;
} APGenerateKeyPair;
```

**Fields**

`keyDataP`
: Pointer to a buffer of seed bytes to be used by the pseudo-random number generator, or `NULL` to have the pseudo-random number generator use the seed data it already has.

`dataLen`
: Length of the `keyDataP` data.

`privateKeyInfoP`
: Pointer to the `APKeyInfoType` structure for the private key.

`publicKeyInfoP`
: Pointer to the `APKeyInfoType` structure for the public key.

**Comments** If the newly-generated key pair is utilized for any cryptographic operations, the pair must be exported and saved in order to be used again. It is statistically improbably that a generated key pair could be regenerated.

## APGetProviderInfo Struct

**Purpose** The command paramater block for the `GetProviderInfo` call. The `info` field is filled in with appropriate information. The provider is responsible for filling the correct information.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APProviderInfoType *infoP;
} APGetProviderInfo;
```

**Fields**

`infoP`
: pointer to the `APProviderInfoType` structure.

## APHash Struct

**Purpose** The command paramater block for the `Hash` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APHashEnum type;
   APHashInfoType *hashInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APHash;
```

**Fields**

`type`
The type of hash requested for this operation.

`hashInfoP`
The hash context information for this operation.

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

`bufOut`
The buffer into which the results of the operation are to be placed.

`bufOutLenP`
The size of the buffer specified by the `bufOut` parameter. Returns the actual number of bytes placed in `bufOut`.

**Comments** The `hashInfoP` parameter specifies an initialized context which was created in the initialization of this class of functions.

The parameter `bufInLen` may not be zero and `bufIn` must be valid.

The provider performs a complete operation of `type` using the data in `bufIn` of length `bufInLen` and puts the results in `bufOut`. The actual number of bytes placed in `bufOut` is placed in `bufOutLenP`.

## APHashFinal Struct

**Purpose** The command paramater block for the `HashFinal` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APHashInfoType *hashInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APHashFinal;
```

**Fields**

`hashInfoP`
A valid context for this operation initialized by the HashInit operation.

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

`bufOut`
The buffer where the results of the operation are placed.

`bufOutLenP`
The size of the buffer specified by the `bufOut` parameter. Returns the actual number of bytes placed in `bufOut`.

**Comments** The `hashInfoP` parameter specifies an initialized context which was created in the Init of this class of functions.

If the `bufInLen` is non-zero, the provider uses the `hashInfoP` to update the operation with the data in `bufIn` of length `bufInLen`.

The provider finalizes the operation which was initialized by the Init operation and places the results of the operation in `bufOut`. The provider updates `bufOutLenP` with the actual number of bytes placed in `bufOut`. The provider is responsible for cleaning up the `hashInfoP` parameter.

## APHashInit Struct

**Purpose** The command paramater block for the `HashInit` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APHashInfoType *hashInfoP;
} APHashInit;
```

**Fields**

`hashInfoP`
A pointer to a hash info structure to pass to the other members of the Hash functions.

**Comments** The provider is responsible for initializing a context of the appropriate for the operation and saving it in `*hashInfoP`. This initialization operation must be concluded with a single HashFinal operation with zero or more HashUpdate operations in between.

## APHashUpdate Struct

**Purpose** The command paramater block for the `HashUpdate` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APHashInfoType *hashInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
} APHashUpdate;
```

**Fields**

`hashInfoP`
A valid context for this operation initialized by the HashInit operation.

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

**Comments** The `hashInfoP` parameter specifies an initialized context which was created during the initialization of this class of functions. The provider uses the `hashInfoP` to update the operation with the data in `bufIn` of length `bufInLen`. The provider also updates the `hashInfoP` to reflect the update. Any number of update operations (including zero) may occur between an intialization and a finalization operation. The parameters `bufIn` and `bufInLen` must be valid for the operation or an error is returned.

## APImportCipherInfo Struct

**Purpose** The command paramater block for the `ImportCipherInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *importDataP;
   uint32_t dataLen;
   APCipherInfoType *cipherInfoP;
} APImportCipherInfo;
```

**Fields**

`encoding`
The encoding of the data being imported.

`importDataP`
The data for the import operation.

`dataLen`
Length of the `importDataP` block.

`cipherInfoP`
A valid context for this operation initialized by the initialization operation.

## APImportHashInfo Struct

**Purpose** The command paramater block for the `ImportHashInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *importDataP;
   uint32_t dataLen;
   APHashInfoType *hashInfoP;
} APImportHashInfo;
```

**Fields**

`encoding`
The encoding of the data being imported.

`importDataP`
: The data for the import operation.

`dataLen`
: Length of the `importDataP` block.

`hashInfoP`
: Pointer to an `APHashInfoType` structure that holds information about the hashing operation.

## APImportKeyInfo Struct

**Purpose** The command paramater block for the `ImportKey` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
    uint8_t encoding;
    uint8_t *importDataP;
    uint32_t dataLen;
    APKeyInfoType *keyInfoP;
} APImportKeyInfo;
```

**Fields**

`encoding`
: The encoding of the data being imported.

`importDataP`
: The data for the import operation.

`dataLen`
: Length of the `importDataP` block.

`keyInfoP`
: An opaque handle to the key generated by the provider.

**Comments** The provider imports a key as specified by the data in `importDataP` and returns an opaque handle the key in `keyInfoP`.

## APImportKeyPairInfo Struct

**Purpose** The command paramater block for the `ImportKeyPair` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *importDataP;
   uint32_t dataLen;
   APKeyInfoType *privateKeyInfoP;
   APKeyInfoType *publicKeyInfoP;
} APImportKeyPairInfo;
```

**Parameters**

`encoding`
The encoding of the data being imported.

`importDataP`
The data for the import operation.

`dataLen`
Length of the import data.

`privateKeyInfoP`
Pointer to the `APKeyInfoType` structure for the private key.

`publicKeyInfoP`
Pointer to the `APKeyInfoType` structure for the public key.

## APImportMacInfo Struct

**Purpose** The command paramater block for the `ImportMAC` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *importDataP;
   uint32_t dataLen;
   APMACInfoType *macInfoP;
} APImportMACInfo;
```

**Fields**

`encoding`
The encoding of the data being imported.

`importDataP`
The data for the import operation.

`dataLen`
Length of the `importDataP` block.

`macInfoP`
Pointer to an `APMACInfoType` structure.

## APImportSignInfo Struct

**Purpose** The command paramater block for the `ImportSignInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *importDataP;
   uint32_t dataLen;
   APSignInfoType *signInfoP;
} APImportSignInfo;
```

**Fields**

`encoding`
The encoding of the data being imported.

`importDataP`
The data for the import operation.

`dataLen`
Length of the `importDataP` block.

`signInfoP`
A valid context for this operation initialized by the initialization operation.

## APImportVerifyInfo Struct

**Purpose** The command paramater block for the `ImportVerifyInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   uint8_t encoding;
   uint8_t *importDataP;
   uint32_t dataLen;
   APVerifyInfoType *verifyInfoP;
} APImportVerifyInfo;
```

**Fields**

`encoding`
The encoding of the data being imported.

`importDataP`
The data for the import operation.

`dataLen`
Length of the `importDataP` block.

`verifyInfoP`
A valid context for this operation initialized by the initialization operation.

## APMac Struct

**Purpose** The command paramater block for the `Mac` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APHashInfoType *hashInfoP;
   APMACEnum type;
   APMACInfoType *macInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APMac;
```

**Fields**

`keyInfoP`
Pointer to an `APKeyInfoType` structure, allocated by the application, containing the key to be used for the operation.

`hashInfoP`
: Pointer to an `APHashInfoType` structure, allocated by the application, that holds information about the hashing operation to be used for the operation.

`type`
: One of the values declared by the `APMACEnum` enum.

`macInfoP`
: Pointer to an `APMACInfoType` structure, allocated by the application, to be used in subsequent calls to the same class of operations.

`bufIn`
: Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

`bufInLen`
: The size, in bytes, of the buffer specified by *`bufIn`*. This value must be greater than zero.

`bufOut`
: Pointer to a buffer, allocated by the application, to receive the output of the operation.

`bufOutLenP`
: The size, in bytes, of the buffer specified by the *`bufOut`* parameter.

## APMacFinal Struct

**Purpose** The command paramater block for the `MacFinal` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APMACInfoType *macInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APMacFinal;
```

**Fields** `macInfoP`
: Pointer to the `APMACInfoType` structure that was initialized during the call to `CPMLibMACInit()`.

`bufIn`
: Pointer to a buffer containing the data for the operation, or `NULL` if there is no additional data.

`bufInLen`
: The size, in bytes, of the buffer specified by *`bufIn`*.

`bufOut`
: Pointer to a buffer, allocated by the application, to receive the output of the operation.

`bufOutLenP`
: The size, in bytes, of the buffer specified by the *`bufOut`* parameter.

## APMacInit Struct

**Purpose** The command paramater block for the `MacInit` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType keyInfoP;
   APHashInfoType hashInfoP;
   APMACInfoType *macInfoP;
} APMacInit;
```

**Parameters**

`keyInfoP`
: Pointer to an `APKeyInfoType` structure, allocated by the application, containing the key to be used in the subsequent calls to `CPMLibMACUpdate()` and `CPMLibMACFinal()`.

`hashInfoP`
: Pointer to an `APHashInfoType` structure, allocated by the application, that holds information about the hashing operation for use in the subsequent calls to `CPMLibMACUpdate()` and `CPMLibMACFinal()`.

`macInfoP`
: Pointer to an `APMACInfoType` structure, allocated by the application, to be used in subsequent calls to the same class of operations.

## APMacUpdate Struct

**Purpose** The command paramater block for the `MacUpdate` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APMACInfoType *macInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
} APMacUpdate;
```

**Fields**

`macInfoP`
: Pointer to the `APMACInfoType` structure that was initialized during the call to `CPMLibMACInit()`.

`bufIn`
: Pointer to a buffer containing the data for the operation. This parameter must not be `NULL`.

`bufInLen`
: The size, in bytes, of the buffer specified by *`bufIn`*. This value must be greater than zero.

`bufOut`
: Pointer to a buffer, allocated by the application, to receive the output of the operation.

`bufOutLenP`
: The size, in bytes, of the buffer specified by the *`bufOut`* parameter.

## APReleaseCipherInfo Struct

**Purpose** The command paramater block for the `ReleaseCipherInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APCipherInfoType *cipherInfoP;
} APReleaseCipherInfo;
```

**Parameters**

`cipherInfoP`
: A valid context for this operation initialized by the initialization operation.

## APReleaseHashInfo Struct

**Purpose** The command paramater block for the `ReleaseHashInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APHashInfoType *hashInfoP;
} APReleaseHashInfo;
```

**Fields** `hashInfoP`
Pointer to the `APHashInfoType` structure.

## APReleaseKeyInfo Struct

**Purpose** The command paramater block for the `ReleaseKey` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
} APReleaseKeyInfo;
```

**Fields** `keyInfoP`
Pointer to the key data to be released.

**Comments** The provider releases the key specified by `keyInfoP`. After this call, `keyInfoP` is no longer valid for any operations.

## APReleaseMACInfo Struct

**Purpose** The command paramater block for the `ReleaseMACInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APMACInfoType *macInfoP;
} APReleaseMACInfo;
```

**Fields** `macInfoP`
Pointer to an `APMACInfoType` structure.

## APReleaseSignInfo Struct

**Purpose** The command paramater block for the `ReleaseSignInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APSignInfoType *signInfoP;
} APReleaseSignInfo;
```

**Fields** `signInfoP`
A valid context for this operation initialized by the initialization operation.

## APReleaseVerifyInfo Struct

**Purpose** The command paramater block for the `ReleaseVerifyInfo` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APVerifyInfoType *verifyInfoP;
} APReleaseVerifyInfo;
```

**Fields** `verifyInfoP`
A valid context for this operation initialized by the initialization operation.

## APSign Struct

**Purpose** The command paramater block for the `Sign` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APSignInfoType *signInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
   uint8_t *signature;
   uint32_t *signatureLenP;
} APSign;
```

**Fields**

`keyInfoP`
The key to be used for the operation.

`signInfoP`
A valid context for this operation initialized by the initialization operation.

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

`bufOut`
The buffer where the results of the operation are to be placed.

`bufOutLenP`
The size of the buffer specified by the `bufOut` parameter. Also returns the actual number of bytes placed in `bufOut` on return.

`signature`
Pointer to a buffer, allocated by the application, to receive the calculated signature.

`signatureLenP`
The size, in bytes, of the buffer specified by the *`signature`* parameter.

**Comments** The `signInfoP` parameter specifies an initialized context which was created in the Init of this class of functions.

The parameter bufInLen may not be zero and bufIn must be valid.

The provider performs a complete operation using the data in bufIn of length bufInLen and puts the results in bufOut. The actual number of bytes placed in bufOut is placed in bufOutLenP.

## APSignFinal Struct

**Purpose** The command paramater block for the SignFinal call.

**Declared In** CPMLibProvider.h

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APSignInfoType *signInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
   uint8_t *signature;
   uint32_t *signatureLenP;
} APSignFinal;
```

**Fields**

keyInfoP
: The key to be used for the operation.

signInfoP
: A valid context for this operation initialized by the initialization operation.

bufIn
: The data to add to the operation.

bufInLen
: The length of the data to add to this operation.

bufOut
: The buffer where the results of the operation are to be placed.

bufOutLenP
: The size of the buffer specified by the bufOut parameter. Also returns the actual number of bytes placed in bufOut on return.

`signature`
: Pointer to a buffer, allocated by the application, to receive the calculated signature.

`signatureLenP`
: The size, in bytes, of the buffer specified by the *`signature`* parameter.

## APSignInit Struct

**Purpose** The command paramater block for the `SignInit` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APSignInfoType *signInfoP;
} APSignInit;
```

**Fields**

`keyInfoP`
: The key to be used for the operation.

`signInfoP`
: A handle to a context to pass to the other members of the Sign functions.

**Comments** The provider is responsible for initializing a context with the specified `keyInfoP` for the operation and saving it in `signInfoP`. This initialization operation must be concluded with a single finalization operation with zero or more update operations in between.

## APSignUpdate Struct

**Purpose** The command paramater block for the `SignUpdate` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APSignInfoType *signInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
} APSignUpdate;
```

**Fields**

`keyInfoP`
The key to be used for the operation.

`signInfoP`
A valid context for this operation initialized by the initialization operation.

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

**Comments** The `signInfoP` parameter specifies an initialized context which was during in the initialization of this class of functions. The provider uses `signInfoP` to update the operation with the data in `bufIn` of length `bufInLen`. The provider also updates the `signInfoP` to reflect the update. Any number of update operations (including zero) may occur between an initialization and a finalization operation. The parameters `bufIn` and `bufInLen` must be valid for the operation or an error is returned.

## APVerify Struct

**Purpose** The command paramater block for the `Verify` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APVerifyInfoType *verifyInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
   uint8_t *signature;
   uint32_t signatureLen;
   VerifyResultType *verifyResultP;
} APVerify;
```

**Fields**

`keyInfoP`
: The key to be used for the operation.

`verifyInfoP`
: A valid context for this operation initialized by the intialization operation

`bufIn`
: The data to add to the operation.

`bufInLen`
: The length of the data to add to this operation.

`bufOut`
: The buffer where the results of the operation are to be placed.

`bufOutLenP`
: The size of the buffer specified by the `bufOut` parameter. Also returns the actual number of bytes placed in `bufOut` on return.

`signature`
: Pointer to a buffer containing the previously calculated signature that is being verified.

`signatureLen`
: The length, in bytes, of the buffer specified by *`signature`*.

`verifyResultP`
: Value that should be set by the provider to zero if the signature verifies or 1 if the signature did not verify.

**Comments** The `verifyInfoP` parameter specifies an initialized context which was created during the initialization of this class of functions.

The parameter `bufInLen` may not be zero and `bufIn` must be valid.

The provider performs a complete verification operation using the data in `bufIn` of length `bufInLen` and puts the results in `bufOut`. The actual number of bytes placed in `bufOut` is placed in `bufOutLenP`.

## APVerifyFinal Struct

**Purpose** The command paramater block for the `VerifyFinal` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APVerifyInfoType *verifyInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
   uint8_t *bufOut;
   uint32_t *bufOutLenP;
   uint8_t *signature;
   uint32_t signatureLen;
   VerifyResultType *verifyResultP;
} APVerifyFinal;
```

**Fields**

`keyInfoP`
The key to be used for the operation.

`verifyInfoP`
A valid context for this operation initialized by the intialization operation

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

`bufOut`
The buffer where the results of the operation are to be placed.

`bufOutLenP`
: The size of the buffer specified by the `bufOut` parameter. Also returns the actual number of bytes placed in `bufOut` on return.

`signature`
: Pointer to a buffer containing the previously calculated signature that is being verified.

`signatureLen`
: The length, in bytes, of the buffer specified by *`signature`*.

`verifyResultP`
: Value that should be set by the provider to zero if the signature verifies or 1 if the signature did not verify.

**Comments** The `verifyInfoP` parameter specifies an initialized context which was created during the initialization of this class of functions.

If `bufInLen` is non-zero, the provider uses `verifyInfoP` to update the operation with the data in `bufIn` of length `bufInLen`.

The provider finalizes the operation which was initialized by the initialize operation and places the results of the operation in `bufOut`. The provider updates `bufOutLenP` with the actual number of bytes placed in `bufOut`. The provider is responsible for cleaning up the `verifyInfoP` parameter.

## APVerifyInit Struct

**Purpose** The command paramater block for the `VerifyInit` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APVerifyInfoType *verifyInfoP;
} APVerifyInit;
```

**Fields**

`keyInfoP`
: The key to be used for the operation.

`verifyInfoP`
: A handle to a context to pass to the other members of the Verify functions.

**Comments** The provider is responsible for initializing a context with the specified `keyInfoP` for the operation and saving it in `verifyInfoP`. This initialization operation must be concluded with a single finalize operation with zero or more update operations in between.

## APVerifyUpdate Struct

**Purpose** The command paramater block for the `VerifyUpdate` call.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
struct {
   APKeyInfoType *keyInfoP;
   APVerifyInfoType *verifyInfoP;
   uint8_t *bufIn;
   uint32_t bufInLen;
} APVerifyUpdate;
```

**Fields**

`keyInfoP`
The key to be used for the operation.

`verifyInfoP`
A valid context for this operation initialized by the intialization operation

`bufIn`
The data to add to the operation.

`bufInLen`
The length of the data to add to this operation.

**Comments** The `verifyInfoP` parameter specifies an initialized context which was created during the initialization of this class of functions. The provider uses the `verifyInfoP` to update the operation with the data in `bufIn` of length `bufInLen`. The provider also updates the `verifyInfoP` to reflect the update. Any number of update operations (including zero) may occur between an initialization and a finalization operation. The parameters `bufIn` and `bufInLen` must be valid for the operation or an error is returned.

# CPM Library Provider Constants

## APCmdType Enum

**Purpose** Algorithm provider command selectors.

**Declared In** `CPMLibProvider.h`

**Constants**

`apClose`
: Handles the closing of the library.

`apDecrypt`
: Performs a decryption operation in one pass. Parameters are passed to this function using an `APDecrypt` structure.

`apDecryptFinal`
: Finalizes a multi-part decryption operation. Parameters are passed to this function using an `APDecryptFinal` structure.

`apDecryptInit`
: Begins a multi-part decryption operation with the specified key and returns the context of the decrypt operation. Parameters are passed to this function using an `APDecryptInit` structure.

`apDecryptUpdate`
: Updates a multi-part decryption operation with more data. Parameters are passed to this function using an `APDecryptUpdate` structure.

`apDeriveKeyData`
: Derives a key from the supplied input data. Parameters are passed to this function using an `APDeriveKeyData` structure.

`apEncrypt`
: Performs an encryption operation in one pass. Parameters are passed to this function using an `APEncrypt` structure.

`apEncryptFinal`
: Finalizes a multi-part encryption operation. Parameters are passed to this function using an `APEncryptFinal` structure.

`apEncryptInit`
: Begins a multi-part encryption operation with the specified key and returns the context of the encrypt operation.

Parameters are passed to this function using an `APEncryptInit` structure.

`apEncryptUpdate`
: Updates a multi-part encryption operation with more data. Parameters are passed to this function using an `APEncryptUpdate` structure.

`apExportCipherInfo`
: Creates a storable instance of an `APCipherInfoType` structure. Parameters are passed to this function using an `APExportCipherInfo` structure.

`apExportHashInfo`
: Creates a storable instance of an `APHashInfoType` structure. Parameters are passed to this function using an `APExportHashInfo` structure.

`apExportKeyInfo`
: Creates a storable instance of a key that is already familiar to the CPM framework. Parameters are passed to this function using an `APExportKeyInfo` structure.

`apExportKeyPairInfo`
: Creates a storable instance of a set of `APKeyInfoType` structures representing a private key and a public key. Parameters are passed to this function using an `APExportKeyPairInfo` structure.

`apExportMACInfo`
: Creates a storable instance of an `APMACInfoType` structure. Parameters are passed to this function using an `APExportMacInfo` structure.

`apExportSignInfo`
: Creates a storable instance of an `APSignInfoType` structure. Parameters are passed to this function using an `APExportSignInfo` structure.

`apExportVerifyInfo`
: Creates a storable instance of an `APVerifyInfoType` structure. Parameters are passed to this function using an `APExportVerifyInfo` structure.

`apGenerateKey`
: Generates a new key. Parameters are passed to this function using an `APGenerateKey` structure.

`apGenerateKeyPair`
: Generates a new public/private key pair. Parameters are passed to this function using an `APGenerateKeyPair` structure.

`apGetInfo`
: Not used.

`apGetProviderInfo`
: Gets information about the requested provider. Information returned includes the name of the provider, some additional text about the provider, the “algorithms” supported, and so on. Parameters are passed to this function using an `APGetProviderInfo` structure.

`apHash`
: Performs the hashing operation in one pass. Parameters are passed to this function using an `APHash` structure.

`apHashFinal`
: Finalizes a multi-part hash operation. Parameters are passed to this function using an `APHashFinal` structure.

`apHashInit`
: Begins a multi-part hash operation of a specified type and returns the context of the hash operation. Parameters are passed to this function using an `APHashInit` structure.

`apHashUpdate`
: Updates a multi-part hash operation with more data. Parameters are passed to this function using an `APHashUpdate` structure.

`apImportCipherInfo`
: Initialize the contents of an `APCipherInfoType` structure based upon a storable instance of that structure. Parameters are passed to this function using an `APImportCipherInfo` structure.

`apImportHashInfo`
: Initialize the contents of an `APHashInfoType` structure based upon a storable instance of that structure. Parameters are passed to this function using an `APImportHashInfo` structure.

`apImportKeyInfo`
: Introduces an existing key to the CPM framework. Parameters are passed to this function using an `APImportKeyInfo` structure.

`apImportKeyPairInfo`
: Introduces an existing public/private key pair to the CPM framework. Parameters are passed to this function using an `APImportKeyPairInfo` structure.

`apImportMACInfo`
: Initialize the contents of an `APMACInfoType` structure based upon a storable instance of that structure. Parameters are passed to this function using an `APImportMacInfo` structure.

`apImportSignInfo`
: Initialize the contents of an `APSignInfoType` structure based upon a storable instance of that structure. Parameters are passed to this function using an `APImportSignInfo` structure.

`apImportVerifyInfo`
: Initialize the contents of an `APVerifyInfoType` structure based upon a storable instance of that structure. Parameters are passed to this function using an `APImportVerifyInfo` structure.

`apMAC`
: Performs the message authentication operation in one pass. Parameters are passed to this function using an `APMac` structure.

`apMACFinal`
: Finalizes a multi-part message authentication operation. Parameters are passed to this function using an `APMacFinal` structure.

`apMACInit`
: Begins a multi-part message authentication operation with the specified key and hash info and returns the context of the operation. Parameters are passed to this function using an `APMacInit` structure.

`apMACUpdate`
Updates a multi-part message authentication operation with more data. Parameters are passed to this function using an `APMacUpdate` structure.

`apOpen`
Handles the opening of the library.

`apReleaseCipherInfo`
Allows the CPM and the provider(s) to clean up before the application frees the `APICipherInfoType` structure. Parameters are passed to this function using an `APReleaseCipherInfo` structure.

`apReleaseHashInfo`
Allows the CPM and the provider(s) to clean up before the application frees the `APIHashInfoType` structure. Parameters are passed to this function using an `APReleaseHashInfo` structure.

`apReleaseKeyInfo`
Allows the CPM and the provider(s) to clean up before the application frees the `APKeyInfoType` structure. Parameters are passed to this function using an `APReleaseKeyInfo` structure.

`apReleaseMACInfo`
Allows the CPM and the provider(s) to clean up before the application frees the `APMACInfoType` structure. Parameters are passed to this function using an `APReleaseMACInfo` structure.

`apReleaseSignInfo`
Allows the CPM and the provider(s) to clean up before the application frees the `APSignInfoType` structure. Parameters are passed to this function using an `APReleaseSignInfo` structure.

`apReleaseVerifyInfo`
Allows the CPM and the provider(s) to clean up before the application frees the `APVerifyInfoType` structure. Parameters are passed to this function using an `APReleaseVerifyInfo` structure.

`apSign`
Performs the signing operation in one pass. Parameters are passed to this function using an `APSign` structure.

`apSignFinal`
: Finalizes a multi-part signing operation. Parameters are passed to this function using an `APSignFinal` structure.

`apSignInit`
: Begins a multi-part signing operation with the specified key and returns the context of the signing operation. Parameters are passed to this function using an `APSignInit` structure.

`apSignUpdate`
: Updates a multi-part signing operation with more data. Parameters are passed to this function using an `APSignUpdate` structure.

`apStatus`

`apVerify`
: Performs the verify operation in one pass. Parameters are passed to this function using an `APVerify` structure.

`apVerifyFinal`
: Finalizes a multi-part verification operation. Parameters are passed to this function using an `APVerifyFinal` structure.

`apVerifyInit`
: Begins a multi-part verification operation with the specified key and returns the context of the verification operation. Parameters are passed to this function using an `APVerifyInit` structure.

`apVerifyUpdate`
: Updates a multi-part verification operation with more data. Parameters are passed to this function using an `APVerifyUpdate` structure.

`apLast`
: Not a value that represents an actual function, this is one greater than the final function selector value.

`apZero`

### Miscellaneous CPM Library Provider Constants

**Purpose** The `CPMLibProvider.h` file also declares the following constants.

**Declared In** `CPMLibProvider.h`

**Constants** `#define cpmProviderResourceID 0`
Defines the resource ID of the code resource.

`#define cpmProviderResourceType 'cpmp'`
Defines the type (creator/type) of the provider.

## Application-Defined Functions

### APDispatchProcPtr Function

**Purpose** Pointer to the provider's command dispatch function. Each provider must export a function matching this signature as a dispatcher for the commands and command parameter blocks that will be sent to the provider by the CPM.

**Declared In** `CPMLibProvider.h`

**Prototype**
```
status_t (*APDispatchProcPtr)
    (CPMCallerInfoPtr info, APCmdType cmd,
    APCmdPBPtr pbP)
```

**Parameters**

→ *info*
Pointer to a `CPMCallerInfoType` structure that contains all the information necessary for the provider to call back into the CPM's framework and use the CPM's resources.

→ *cmd*
One of the values listed under "CPM Library Provider Constants" on page 281.

↔ *pbP*
Pointer to the appropriate parameter block structure for the command. The documentation for each command identifies the parameter block structure to use with that command.

**Returns** `errNone` if the operation completed successfully, or an appropriate error code (usually, one of the codes listed under "CPM Library Error Codes" on page 239).

**See Also** `CPMDispatcherProcPtr()`

## CPMAddRandomSeedProcPtr Function

**Purpose** Pointer to a function that puts a number of seed bytes into the pseudo-random number generator maintained by the CPM.

**Declared In** `CPMLibProvider.h`

**Prototype**
```
status_t (*CPMAddRandomSeedProcPtr)
   (void *appContext, uint8_t *buffer,
   uint32_t buflen)
```

**Parameters**

→ *appContext*
Pointer to a structure containing the provider-specific application context to be used in future CPM calls. This structure is normally created and initialized when the library is opened.

→ *buffer*
Pointer to a buffer of seed bytes.

→ *buflen*
The number of bytes in *buffer*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**See Also** `CPMAddRandomSeedProcPtr()`

## CPMDebugOutProcPtr Function

**Purpose** Pointer to a function that sends a null-terminated string to the current debug device, if the level set using

CPMLibSetDebugLevel() is less than or equal to the message's debug output level.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
status_t (*CPMDebugOutProcPtr) (void *appContext,
    uint8_t level, char *fmtstring, ...)
```

**Parameters**

→ *appContext*
Pointer to a structure containing the provider-specific application context to be used in future CPM calls. This structure is normally created and initialized when the library is opened.

→ *level*
The debug output level of the message. One of the values listed under "Debug Output Levels" on page 239.

→ *fmtstring*
A printf-style format string for the debug message being sent.

→ *...*
Zero or more arguments to be substituted as appropriate in the format string.

**Returns** `errNone`.

**See Also** CPMLibSetDebugLevel()

## CPMDispatcherProcPtr Function

**Purpose** Pointer to the Cryptographic Provider Manager's function dispatcher. This function attempts to locate a provider that implements the supplied command. If it is successful, it directs the command and parameter block to that provider.

**Declared In** `CPMLibProvider.h`

**Prototype**

```
status_t (*CPMDispatcherProcPtr)
    (void *appContext, APCmdType cmd,
    APCmdPBType *cmdPB, uint32_t *id)
```

**Parameters**

→ *appContext*
Pointer to a structure containing the provider-specific application context to be used. This structure is normally created and initialized when the library is opened.

→ *cmd*
One of the values listed under "CPM Library Provider Constants" on page 281.

→ *cmdPB*
Pointer to the appropriate parameter block structure for the command. The documentation for each command identifies the parameter block structure to use with that command.

→ *id*
Pointer to a `uint32_t` containing the ID of a specific provider to which the command is to be directed. If `*id` is 0, the CPM directs the command to the first provider that implements the requested functionality.

**Returns** The return value passed back from the provider (which may be `cpmErrUnimplemented` if the provider recognizes but does not implement the supplied command), or one of the following if the command couldn't be directed to a provider:

`cpmErrProviderNotFound`
A provider ID was supplied, but the CPM couldn't find a provider with the supplied ID.

`cpmErrNoProviders`
Given the supplied application context, the CPM is unaware of any providers.

`cpmErrNoAppContext`
The *appContext* parameter is `NULL`.

`cpmErrParamErr`
The supplied command isn't a valid command.

`cpmErrUnsupported`
The supplied command isn't supported.

**Comments** If `*id` is `0`, the CPM's function dispatcher attempts to locate a provider that implements *cmd*. If it finds one, that provider's dispatch function (an `APDispatchProcPtr`) is called with the specified command and command parameters.

**See Also** `APDispatchProcPtr`

## CPMGenerateRandomBytesProcPtr Function

**Purpose** Pointer to a function that returns a requested number of random bytes.

**Declared In** `CPMLibProvider.h`

**Prototype**
```
status_t (*CPMGenerateRandomBytesProcPtr)
    (void *appContext, uint8_t *buffer,
    uint32_t *buflenP)
```

**Parameters**

→ *appContext*
: Pointer to a structure containing the provider-specific application context to be used in future CPM calls. This structure is normally created and initialized when the library is opened.

← *buffer*
: Pointer to a buffer, allocated by the application, into which the random bytes are written.

↔ *buflenP*
: When this function is called, the variable pointed to by this parameter is set to the size of *buffer*. Upon return, the variable contains the number of random bytes written to *buffer*.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

**Comments** If there are not *bufLenP* random bytes available, this function returns the available random bytes and returns the number of available bytes in *bufLenP*.

**See Also** `CPMAddRandomSeedProcPtr()`

# 12

# Encrypt

These functions allow you to encrypt or digest strings. The header file `Encrypt.h` only declares functions, so this chapter only consists of a single section:



The header file `Encrypt.h` declares the API that this chapter describes.

## Encrypt Functions and Macros

### EncDES Function

**Purpose** Perform a reversible encryption or decryption of an 8-byte string using an 8-byte key.

**Declared In** `Encrypt.h`

**Prototype**
```
status_t EncDES (uint8_t *srcP, uint8_t *keyP,
   uint8_t *dstP, Boolean encrypt)
```

**Parameters**

→ *srcP*
The 8-byte string to be encrypted.

→ *keyP*
The 8-byte key with which to encrypt the string in *srcP*.

← *dstP*
The 8-byte encrypted result.

→ *encrypt*
Pass `true` to encrypt, `false` to decrypt.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## EncDigestMD4 Function

**Purpose** Digest a string of bytes to produce a 128-bit result using the MD4 algorithm.

**Declared In** `Encrypt.h`

**Prototype**
```
status_t EncDigestMD4 (uint8_t *strP,
   uint16_t strLen, uint8_t digestP[16])
```

**Parameters**

→ *strP*
The string to be digested.

→ *strLen*
The length of the string passed in `strP`.

← *digestP[16]*
The resulting 128-bit (16 byte) digest.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

## EncDigestMD5 Function

**Purpose** Digest a string of bytes to produce a 128-bit result using the MD5 algorithm.

**Declared In** `Encrypt.h`

**Prototype**
```
status_t EncDigestMD5 (uint8_t *strP,
   uint16_t strLen, uint8_t digestP[16])
```

**Parameters**

→ *strP*
The string to be digested.

→ *strLen*
The length of the string passed in `strP`.

← *digestP[16]*
The resulting 128-bit (16 byte) digest.

**Returns** `errNone` if the operation completed successfully, or one of the error codes listed under "CPM Library Error Codes" on page 239 otherwise.

# 13

# Password

The APIs declared in `Password.h` and documented in this chapter are provided for compatibility with previous versions of Palm OS. Applications written specifically for Palm OS Cobalt should make use of the Authentication Manager instead.

The contents of this chapter are organized into the following sections:



The header file `Password.h` declares the API that this chapter describes.

## Password Constants

### Miscellaneous Password Constants

**Purpose** The following constants are declared in `Password.h`.

**Declared In** `Password.h`

**Constants** `#define pwdLength 32`

The maximum length of the password string passed to `PwdVerify()`.

# Password Functions and Macros

## PwdExists Function

**Purpose** Determine if the system password is set.

**Declared In** `Password.h`

**Prototype** `Boolean PwdExists (void)`

**Parameters** None.

**Returns** Returns `true` if the system password is set, `false` otherwise.

**Compatibility** This function is provided for compatibility with previous versions of Palm OS only. Palm OS Cobalt applications should make use of the APIs provided by the Authentication Manager instead.

## PwdRemove Function

**Purpose** Remove the encrypted password string and recover data hidden in databases.

**Declared In** `Password.h`

**Prototype** `void PwdRemove (void)`

**Parameters** None.

**Returns** Nothing.

**Comments** **IMPORTANT:** When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

**Compatibility** This function is provided for compatibility with previous versions of Palm OS only. Palm OS Cobalt applications should make use of the APIs provided by the Authentication Manager instead.

## PwdSet Function

**Purpose** Use a passed string as the new password.

**Declared In** `Password.h`

**Prototype**
```
void PwdSet (char *oldPassword,
    char *newPassword)
```

**Parameters**
→ *`oldPassword`*
The old password. It must be successfully verified or the new password isn't accepted

→ *`newPassword`*
A string to use as the password. `NULL` means no password.

**Returns** Nothing.

**Comments** The password is stored in an encrypted form.

**IMPORTANT:** When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

**Compatibility** This function is provided for compatibility with previous versions of Palm OS only. Palm OS Cobalt applications should make use of the APIs provided by the Authentication Manager instead.

## PwdVerify Function

**Purpose** Verify that a passed string matches the system password.

**Declared In** `Password.h`

**Prototype**
```
Boolean PwdVerify (char *string)
```

**Parameters**
→ *`string`*
String to compare to the system password. `NULL` means no current password.

**Returns** Returns `true` if the string matches the system password.

**Comments**

**IMPORTANT:** When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

**Compatibility**

This function is provided for compatibility with previous versions of Palm OS only. Palm OS Cobalt applications should make use of the APIs provided by the Authentication Manager instead.

# 14

# Security Services

The Security Services manage the device settings regarding the user's level of concern for security. This includes the overall security level set for the device as well as the device lockout settings.

This chapter provides reference documentation for the Security Services. It is organized into the following sections:



The header file `SecurityServices.h` declares the API that this chapter describes.

# Security Services Structures and Types

## SecSvcsDecodeLockoutTimePtrType Typedef

**Purpose** Pointer to the `SecSvcsDecodeLockoutTime()` function.

**Declared In** `SecurityServices.h`

**Prototype**
```
typedef status_t
      (*SecSvcsDecodeLockoutTimePtrType)
      (uint32_t encoded_level,
      SecSvcsDeviceLockoutEnum *lockoutType,
      uint32_t *hours, uint32_t *minutes)
```

## SecSvcsEncodeLockoutTimePtrType Typedef

**Purpose** Pointer to the `SecSvcsEncodeLockoutTime()` function.

**Declared In** `SecurityServices.h`

**Prototype**
```
typedef status_t
      (*SecSvcsEncodeLockoutTimePtrType)
      (SecSvcsDeviceLockoutEnum lockoutType,
      uint32_t *encoded_level, uint32_t hours,
      uint32_t minutes)
```

## SecSvcsGetDeviceLockoutPtrType Typedef

**Purpose** Pointer to the `SecSvcsGetDeviceLockout()` function.

**Declared In** `SecurityServices.h`

**Prototype**
```
typedef status_t
      (*SecSvcsGetDeviceLockoutPtrType)
      (uint32_t *encoded_level)
```

## SecSvcsGetDevicePoliciesPtrType Typedef

**Purpose** Pointer to the `SecSvcsGetDevicePolicies()` function.

**Declared In** `SecurityServices.h`

**Prototype**

```
typedef status_t
       (*SecSvcsGetDevicePoliciesPtrType)
       (uint32_t creatorID, uint8_t *buffer,
       uint32_t *buflen)
```

## SecSvcsGetDeviceSettingPtrType Typedef

**Purpose** Pointer to the `SecSvcsGetDeviceSetting()` function.

**Declared In** `SecurityServices.h`

**Prototype**

```
typedef status_t
       (*SecSvcsGetDeviceSettingPtrType)
       (SecSvcsDeviceSettingEnum *level)
```

## SecSvcsIsDeviceLockedPtrType Typedef

**Purpose** Pointer to the `SecSvcsIsDeviceLocked()` function.

**Declared In** `SecurityServices.h`

**Prototype**

```
typedef Boolean (*SecSvcsIsDeviceLockedPtrType)
       (void)
```

## SecSvcsSetDeviceLockedPtrType Typedef

**Purpose** Pointer to the `SecSvcsSetDeviceLocked()` function.

**Declared In** `SecurityServices.h`

**Prototype**

```
typedef status_t (*SecSvcsSetDeviceLockedPtrType)
       (Boolean locked)
```

## SecSvcsSetDeviceLockoutPtrType Typedef

**Purpose** Pointer to the `SecSvcsSetDeviceLockout()` function.

**Declared In** `SecurityServices.h`

**Prototype**

```
typedef status_t
       (*SecSvcsSetDeviceLockoutPtrType)
       (uint32_t encoded_level)
```

### SecSvcsSetDeviceSettingPtrType Typedef

**Purpose** Pointer to the `SecSvcsSetDeviceSetting()` function.

**Declared In** `SecurityServices.h`

**Prototype**

```
typedef status_t
      (*SecSvcsSetDeviceSettingPtrType)
      (SecSvcsDeviceSettingEnum level)
```

## Security Services Constants

### Security Services Entry Points

**Purpose** Each of the Security Services functions is identified by its entry point. These constants define those entry points.

**Declared In** `SecurityServices.h`

**Constants**

```
#define entryNumSecSvcsDecodeLockoutTime (6)

#define entryNumSecSvcsEncodeLockoutTime (5)

#define entryNumSecSvcsGetDeviceLockout (3)

#define entryNumSecSvcsGetDevicePolicies (0)

#define entryNumSecSvcsGetDeviceSetting (1)

#define entryNumSecSvcsSetDeviceLockout (4)

#define entryNumSecSvcsSetDeviceSetting (2)
```

### Security Services Errors

**Purpose** Error codes returned by the various Security Services functions.

**Declared In** `SignVfy.h`

**Constants**

```
#define secSvcsErrBufferTooSmall
  ((status_t)(secSvcsErrorClass | 2))
```

The supplied buffer was too small. The required size has been returned through the length parameter.

`#define secSvcsErrInvalid ((status_t)(secSvcsErrorClass | 7))`
The specified security level or lockout type isn't one of the allowable values.

`#define secSvcsErrNoPolicies ((status_t)(secSvcsErrorClass | 3))`
The licensee has not specified any policies for the device.

`#define secSvcsErrNotImplemented ((status_t)(secSvcsErrorClass | 1))`
A requested service is not implemented.

`#define secSvcsErrOutOfMemory ((status_t)(secSvcsErrorClass | 5))`
There was insufficient memory to complete the operation.

`#define secSvcsErrServiceNotStarted ((status_t)(secSvcsErrorClass | 6))`
The Security Services process has not started.

`#define secSvcsErrUnauthorized ((status_t)(secSvcsErrorClass | 4))`
The caller is not authorized to perform the requested operation.

## Miscellaneous Security Services Constants

**Purpose** The SecurityServices.h header file also declares these constants.

**Declared In** `SecurityServices.h`

**Constants** `#define SecSvcsServiceName "pSysSecSvcs"`
The name under which the Security Services are registered with the Service Manager.

## SecSvcsDeviceLockoutEnum Enum

**Purpose** “Lockout type” values that specify when the device will be automatically locked by the operating system.

**Declared In** `SecurityServices.h`

**Constants** `SecSvcsDeviceLockoutNever = 0`
The device never locks.

`SecSvcsDeviceLockoutPowerOff`
The device locks upon power off.

`SecSvcsDeviceLockoutAt`
The device is locked at the specified time.

`SecSvcsDeviceLockoutAfter`
The device is locked after the specified amount of time has elapsed.

## SecSvcsDeviceSettingEnum Enum

**Purpose** The user’s “paranoia level.”

**Declared In** `SecurityServices.h`

**Constants** `SecSvcsDeviceSecurityNone = 0`
No security needed: the device and services should be as open as possible.

`SecSvcsDeviceSecurityMedium`
The user would like to be notified whenever security-related operations take place, and where appropriate given the opportunity to veto the operation.

`SecSvcsDeviceSecurityHigh`
The user is extremely security-conscious and likely wants all security-related operations denied.

# Security Services Functions and Macros

## SecSvcsDecodeLockoutTime Function

**Purpose** Decodes the 32-bit `encoded_level` value obtained from `SecSvcsGetDeviceLockout()` into a set of lockout parameters (*lockoutType*, *hours*, and *minutes*).

**Declared In** `SecurityServices.h`

**Prototype**

```
status_t SecSvcsDecodeLockoutTime
      (uint32_t encoded_level,
      SecSvcsDeviceLockoutEnum *lockoutType,
      uint32_t *hours, uint32_t *minutes)
```

**Parameters**

→ *encoded_level*
The encoded lockout parameters.

← *lockoutType*
One of the `SecSvcsDeviceLockoutEnum` values that specifies when the device will be locked.

← *hours*
In conjunction with *minutes*, the time when the device will lock or the amount of time that must elapse before the device will lock. See the Comments section, below, for more information.

← *minutes*
In conjunction with *hours*, the time when the device will lock or the amount of time that must elapse before the device will lock. See the Comments section, below, for more information.

**Returns** Always returns `errNone`.

**Comments** If the lockout type is `SecSvcsDeviceLockoutAt`, the device will lock at the time specified by *hours* and *minutes*. If the lockout type is `SecSvcsDeviceLockoutAfter`, the device will lock after the specified number of hours and minutes have elapsed.

**See Also** `SecSvcsEncodeLockoutTime()`

## SecSvcsEncodeLockoutTime Function

**Purpose** Encodes the lockout parameters into a 32-bit value for use with `SecSvcsSetDeviceLockout()`.

**Declared In** `SecurityServices.h`

**Prototype**

```
status_t SecSvcsEncodeLockoutTime
      (SecSvcsDeviceLockoutEnum lockoutType,
      uint32_t *encoded_level, uint32_t hours,
      uint32_t minutes)
```

**Parameters**

→ *`lockoutType`*
One of the `SecSvcsDeviceLockoutEnum` values that specifies when the device will be locked.

← *`encoded_level`*
The encoded lockout parameters.

→ *`hours`*
In conjunction with *`minutes`*, the time when the device will lock or the amount of time that must elapse before the device will lock. See the Comments section, below, for more information.

→ *`minutes`*
In conjunction with *`hours`*, the time when the device will lock or the amount of time that must elapse before the device will lock. See the Comments section, below, for more information.

**Returns** Always returns `errNone`.

**Comments** If the lockout type is `SecSvcsDeviceLockoutAt`, the device will lock at the time specified by *`hours`* and *`minutes`*. If the lockout type is `SecSvcsDeviceLockoutAfter`, the device will lock after the specified number of hours and minutes have elapsed.

**See Also** `SecSvcsDecodeLockoutTime()`

## SecSvcsGetDeviceLockout Function

**Purpose** Gets the lockout parameters as currently set for the device.

**Declared In** `SecurityServices.h`

**Prototype**
```
status_t SecSvcsGetDeviceLockout
       (uint32_t *encoded_level)
```

**Parameters**

← *encoded_level*
The lockout parameters. These are encoded to save space and must be decoded using `SecSvcsDecodeLockoutTime()`.

**Returns** Always returns `errNone`.

**See Also** `SecSvcsSetDeviceLockout()`

## SecSvcsGetDevicePolicies Function

**Purpose** Obtain the security policies defined for the device.

**Declared In** `SecurityServices.h`

**Prototype**
```
status_t SecSvcsGetDevicePolicies
       (uint32_t creatorID, uint8_t *buffer,
       uint32_t *buflen)
```

**Parameters**

→ *creatorID*
Specifies the particular policies being requested. Note that this is not specifically related to actual creator ID values but rather agreed-upon and documented names. In this way a manager can utilize multiple sets of policies.

← *buffer*
A caller-allocated buffer that will receive the requested policies, or `NULL` to request the necessary buffer size (which is then returned through *buflen*).

↔ *buflen*
When calling this function, this is the size of *buffer*. Upon return it is set to the total size of the returned policies.

**Returns** Returns `errNone` if the operation completed successfully, `secSvcsErrNoPolicies` if the device has no policies, or one of the following otherwise:

`secSvcsErrBufferTooSmall`
The specified buffer is too small to contain the security policies. `*buflen` has been set to the needed buffer size.

**Comments** Upon calling this function, you can expect one of three "valid" returns:

- No policies returned (`secSvcsErrNoPolicies`). The licensee has not specified any policies and the service or manager should behave openly (that is, with no restrictions).
- A multiple of 20 bytes is returned, with each 20 byte boundary representing a valid certificate ID.
- One 20 byte wildcard certificate is returned. A wildcard certificate represents a slightly higher level of security than the totally open device. The wildcard means that the service or manager will check to see if the code is signed by a signer already in the list of trusted roots. If the code is signed by one of the trusted roots, then allow the operation. If the code is not signed by one of the trusted roots or not signed at all then annoy the user with as much information about the code as possible (not signed, signed by whom, etc.) and ask the user if the code should be applied.

## SecSvcsGetDeviceSetting Function

**Purpose** Obtain the current security services setting.

**Declared In** `SecurityServices.h`

**Prototype**
```
status_t SecSvcsGetDeviceSetting
      (SecSvcsDeviceSettingEnum *level)
```

**Parameters** ← *level*
One of the SecSvcsDeviceSettingEnum values.

**Returns** Always returns `errNone`.

**Comments** Generally, for `SecSvcsDeviceSecurityNone` the user does not care and the device and services should be as open as possible. For `SecSvcsDeviceSecurityMedium` the user is requesting notification for operations performed by various services, perhaps a yes/no dialog before the operation takes place. For

`SecSvcsDeviceSecurityHigh` the user is “paranoid” and probably wants the operations denied

See Also [`SecSvcsGetDevicePolicies()`](), [`SecSvcsIsDeviceLocked()`](), [`SecSvcsSetDeviceSetting()`]()

## SecSvcsIsDeviceLocked Function

**Purpose** Determine whether the device is currently in a locked state.

**Declared In** `SecurityServices.h`

**Prototype** `Boolean SecSvcsIsDeviceLocked (void)`

**Parameters** None.

**Returns** Returns `true` if the device is locked, `false` if it is not.

**Comments** This function is intended to be used for low-level modules that control whether or not the device is locked.

**See Also** [`SecSvcsGetDeviceLockout()`](), [`SecSvcsSetDeviceLocked()`]()

## SecSvcsSetDeviceLocked Function

**Purpose** Lock or unlock the device.

**Declared In** `SecurityServices.h`

**Prototype** `status_t SecSvcsSetDeviceLocked (Boolean` *`locked`*`)`

**Parameters** → *`locked`*
Supply a value of `true` to lock the device, or `false` to unlock it.

**Returns** Returns `errNone` if the operation completed successfully, or `secSvcsErrServiceNotStarted` if the Security Services process has not started.

**Comments** This function is intended to be used for low-level modules that control whether or not the device is locked.

**See Also** [`SecSvcsIsDeviceLocked()`](), [`SecSvcsSetDeviceLockout()`]()

## SecSvcsSetDeviceLockout Function

**Purpose** Sets the current lockout parameters for the device.

**Declared In** `SecurityServices.h`

**Prototype**
```
status_t SecSvcsSetDeviceLockout
        (uint32_t encoded_level)
```

**Parameters**
→ *`encoded_level`*
The lockout parameters. These must have been encoded using `SecSvcsEncodeLockoutTime()`.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`amErrUserCancel`
The user canceled the operation.

`secSvcsErrInvalid`
The specified lockout type isn't one of the allowable values.

`secSvcsErrServiceNotStarted`
The Security Services process has not started.

**Comments**
**IMPORTANT:** When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

**See Also** `SecSvcsGetDeviceLockout()`

## SecSvcsSetDeviceSetting Function

**Purpose** Change the current security services setting.

**Declared In** `SecurityServices.h`

**Prototype**
```
status_t SecSvcsSetDeviceSetting
        (SecSvcsDeviceSettingEnum level)
```

**Parameters**
→ *`level`*
One of the `SecSvcsDeviceSettingEnum` values.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`amErrUserCancel`
The user canceled the operation.

`secSvcsErrInvalid`
The specified level isn't one of the allowable values.

**Comments**

**IMPORTANT:** When called from the main application thread, this function may block. While blocked, the application will not receive events and won't redraw its windows. As well, deferred sublaunches and notifications won't execute while the main application thread is blocked.

**See Also**

`SecSvcsGetDeviceSetting()`

# 15

# Signature Verification Library

Signed code in Palm OS Cobalt is used to validate the authenticity of a program resource. There are several types of resources that can be signed in Palm OS Cobalt, including applications, system patches, shared libraries, system components (add-ons), and system drivers. All of these resources are packaged as PRC files and then loaded onto the device. The APIs in the Signature Verification Library can be used to verify a PRC's signature on the device.

This chapter provides reference documentation for the Signature Verification Library. This chapter is organized into the following sections:



The header file `SignVfy.h` declares the API that this chapter describes.

# Signature Verification Library Structures and Types

## SignCertificateBlockType Struct

**Purpose** Data structure that references a signature block.

**Declared In** `SignVfy.h`

**Prototype**

```
typedef struct {
   uint16_t encoding;
   SignCertificateIDType certificateID;
} SignCertificateBlockType
```

**Fields**

`encoding`
: The certificate encoding.

`certificateID`
: The certificate ID.

**Comments** The certificate structure returned by `SignGetCertificateByIndex()` and `SignGetCertificateByID()` is byte buffer that contains the X.509 representation of the certificate. The signature verification library does not attempt to interpret the X.509 representation of the certificate; that task is left up to the certificate manager.

## SignCertificateIDType Typedef

**Purpose** Contains a certificate ID.

**Declared In** `SignVfy.h`

**Prototype** `typedef uint8_t SignCertificateIDType[20]`

**Comments** The certificate ID holds the SHA1 digest of the certificate public key.

## SignSignatureBlockType Struct

**Purpose** Data structure that references a signature block.

**Declared In** `SignVfy.h`

**Prototype**

```
typedef struct {
   uint32_t index;
   SignCertificateIDType certificateID;
   uint32_t signingDate;
} SignSignatureBlockType
```

**Fields**

`index`
: Index position of the signature block in the sign resource.

`certificateID`
: ID of the certificate used to verify this signature.

`signingDate`
: Date when the PRC file was signed.

## SignGetNumSignaturesPtrType Typedef

**Purpose** Pointer to the `SignGetNumSignatures()` function.

**Declared In** `SignVfy.h`

**Prototype**

```
typedef status_t (*SignGetNumSignaturesPtrType)
      (DmOpenRef dbP, uint16_t *sigCountP)
```

## SignGetShLibCertIdListPtrType Typedef

**Purpose** Pointer to the `SignGetShLibCertIdListPtrType()` function.

**Declared In** `SignVfy.h`

**Prototype**

```
typedef status_t (*SignGetShLibCertIdListPtrType)
      (DmOpenRef dbP, uint8_t *certIdList,
      uint32_t *certIdListSize)
```

### SignVerifySignatureByIDPtrType Typedef

**Purpose** Pointer to the `SignVerifySignatureByID()` function.

**Declared In** `SignVfy.h`

**Prototype**
```
typedef status_t (*SignVerifySignatureByIDPtrType)
      (DmOpenRef dbP,
      const SignCertificateIDType certificateID)
```

### SignVerifySignatureByIndexPtrType Typedef

**Purpose** Pointer to the `SignVerifySignatureByIndex()` function.

**Declared In** `SignVfy.h`

**Prototype**
```
typedef status_t
      (*SignVerifySignatureByIndexPtrType)
      (DmOpenRef dbP, uint16_t index)
```

## Signature Verification Library Constants

### Signature Verification Library Entry Points

**Purpose** Each of the functions in the Signature Verification Library is identified by its entry point. These constants define those entry points.

**Declared In** `SignVfy.h`

**Constants**
```
#define entryNumSignGetCertificateByID (7)
#define entryNumSignGetCertificateByIndex (5)
#define entryNumSignGetDigest (8)
#define entryNumSignGetNumCertificates (3)
#define entryNumSignGetNumSignatures (2)
#define entryNumSignGetOverlayCertIdList (9)
#define entryNumSignGetShLibCertIdList (10)
#define entryNumSignGetSignatureByID (6)
```

```
#define entryNumSignGetSignatureByIndex (4)
#define entryNumSignVerifySignatureByID (1)
#define entryNumSignVerifySignatureByIndex (0)
```

## Signature Verification Library Errors

**Purpose** Error codes returned by the various Signature Verification Library functions.

**Declared In** `SignVfy.h`

**Constants**

`#define signErrBufferTooSmall (signErrorClass | 10)`
: The supplied buffer was too small. The required size has been returned through the length parameter.

`#define signErrDigestMismatch (signErrorClass | 7)`
: The signed digest does not match the calculated PRC digest.

`#define signErrIndexOutOfBounds (signErrorClass | 3)`
: The specified index is outside the range of certificate or signature indexes for the PRC.

`#define signErrInvalidCertResource (signErrorClass | 5)`
: The `'cert'` resource is malformed, or invalid in some way.

`#define signErrInvalidParams (signErrorClass | 9)`
: One or more function parameters is invalid.

`#define signErrInvalidResourceInDB (signErrorClass | 12)`
: The PRC contains an invalid resource.

`#define signErrInvalidSignatureBlock (signErrorClass | 6)`
: The signature block is invalid.

`#define signErrInvalidSignResource (signErrorClass | 4)`
: The `'sign'` resource is malformed, or invalid in some way.

`#define signErrNoCertResource (signErrorClass | 2)`
: No `'cert'` resource exists in the PRC file.

`#define signErrNoSignResource (signErrorClass | 1)`
No `'sign'` resource exists in the PRC file.

`#define signErrNotFound (signErrorClass | 8)`
A certificate or signature with the specified ID was not found.

`#define signErrOutOfMemory (signErrorClass | 11)`
There was insufficient memory to complete the operation.

# Signature Verification Library Functions and Macros

## SignGetCertificateByID Function

**Purpose** Get a certificate by its ID.

**Declared In** `SignVfy.h`

**Prototype**

```
status_t SignGetCertificateByID (DmOpenRef dbP,
      const SignCertificateIDType certificateID,
      SignCertificateBlockType *certificateBlock,
      uint32_t *certificateLength,
      uint8_t *certificateData)
```

→ *dbP*
Pointer to an open PRC database from which to get certificates.

→ *certificateID*
The 20-byte ID of the certificate.

← *certificateBlockP*
The PRC's certificate block. See `SignCertificateBlockType`.

↔ *certificateLength*
When calling this function, `*`*certificateLength* should contain the size of the buffer indicated by *certificateData*. Upon return, it contains the length of the returned certificate data. If a `NULL` pointer was passed in for *certificateData*, `signErrBufferTooSmall` is returned and the required length is returned through this parameter.

← *`certificateData`*
A pointer to a caller-allocated buffer to receive the certificate data. To determine how large this buffer should be, set this parameter to `NULL`; upon return `*`*`certificateLength`* will contain the needed buffer size. After allocating a buffer of the proper size, call this function again to obtain the certificate.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrInvalidParameter`
*`certificateBlockP`* is `NULL`.

`signErrNoCertResource`
No `'cert'` resource exists in the PRC file.

`signErrInvalidCertResource`
The `'cert'` resource is malformed, or invalid in some way.

`signErrNotFound`
A certificate with the specified ID was not found.

`signErrBufferTooSmall`
The supplied buffer was too small. The required size has been returned through the *`certificateLength`* parameter.

`signErrInvalidParameter`
The *`certificateLength`* parameter was set to `NULL`.

**Comments** The ID of a certificate is the SHA1 digest of the DER encoded `SubjectPublicKeyInfo` field in the certificate, including the sequence tag and length.

The certificate structure returned by this function is a byte buffer that contains the X.509 representation of the certificate. The signature verification library does not attempt to interpret the X.509 representation of the certificate; that task is left up to the Certificate Manager.

**See Also** `SignGetCertificateByIndex()`, `SignGetNumCertificates()`, `SignGetSignatureByID()`

## SignGetCertificateByIndex Function

**Purpose** Get a certificate given its index in the sign resource's certificate block list.

**Declared In** `SignVfy.h`

**Prototype**

```
status_t SignGetCertificateByIndex (DmOpenRef dbP,
      uint16_t index,
      SignCertificateBlockType *certificateBlock,
      uint32_t *certificateLength,
      uint8_t *certificateData)
```

**Parameters**

→ *dbP*
Pointer to an open PRC database from which to get certificates.

→ *index*
The position of the certificate within the certificate block list.

← *certificateBlockP*
The PRC's certificate block. See `SignCertificateBlockType`.

↔ *certificateLength*
When calling this function, `*`*certificateLength* should contain the size of the buffer indicated by *certificateData*. Upon return, it contains the length of the returned certificate data. If a `NULL` pointer was passed in for *certificateData*, `signErrBufferTooSmall` is returned and the required length is returned through this parameter.

← *certificateData*
A pointer to a caller-allocated buffer to receive the certificate data. To determine how large this buffer should be, set this parameter to `NULL`; upon return `*`*certificateLength* will contain the needed buffer size. After allocating a buffer of the proper size, call this function again to obtain the certificate.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrInvalidParameter`
*certificateBlockP* is `NULL`.

`signErrNoCertResource`
: No `'cert'` resource exists in the PRC file.

`signErrInvalidCertResource`
: The `'cert'` resource is malformed, or invalid in some way.

`signErrIndexOutOfBounds`
: The specified index is outside the range of certificate indexes for the PRC's certificate block.

`signErrBufferTooSmall`
: The supplied buffer was too small. The required size has been returned through the *`certificateLength`* parameter.

`signErrInvalidParameter`
: The *`certificateLength`* parameter was set to `NULL`.

**Comments** The ID of a certificate is the SHA1 digest of the DER encoded `SubjectPublicKeyInfo` field in the certificate, including the sequence tag and length.

The certificate structure returned by this function is a byte buffer that contains the X.509 representation of the certificate. The signature verification library does not attempt to interpret the X.509 representation of the certificate; that task is left up to the Certificate Manager.

**See Also** `SignGetCertificateByID()`, `SignGetNumCertificates()`, `SignGetSignatureByIndex()`

## SignGetDigest Function

**Purpose** Calculate the digest of a PRC.

**Declared In** `SignVfy.h`

**Prototype**
```
status_t SignGetDigest (DmOpenRef dbP,
      APHashInfoType *hashinfo)
```

**Parameters**

→ *`dbP`*
: Pointer to an open PRC database for which the digest is to be calculated.

↔ *`hashinfo`*
: An initialized `APHashInfoType` structure.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrOutOfMemory`
: There was insufficient memory to complete the operation.

`signErrInvalidResourceInDB`
: The PRC contains an invalid resource.

**Comments** The caller is responsible for calling `CPMLibHashFinal()` and releasing any resources allocated by the Crytographic Provider Manager with `CPMLibReleaseHashInfo()`.

## SignGetNumCertificates Function

**Purpose** Get the number of certificates in the `'cert'` resource.

**Declared In** `SignVfy.h`

**Prototype**

```
status_t SignGetNumCertificates (DmOpenRef dbP,
    uint16_t *num)
```

**Parameters**

→ *dbP*
: Pointer to an open PRC database from which to get certificates.

← *num*
: The number of certificates in the `'cert'` resource.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrInvalidParameter`
: *num* is `NULL`.

`signErrNoCertResource`
: No `'cert'` resource exists in the PRC file.

`signErrInvalidCertResource`
: The `'cert'` resource is malformed, or invalid in some way.

**See Also** `SignGetCertificateByIndex()`, `SignGetNumSignatures()`

## SignGetNumSignatures Function

**Purpose** Get the number of signatures in the `'sign'` resource.

**Declared In** `SignVfy.h`

**Prototype**
```
status_t SignGetNumSignatures (DmOpenRef dbP,
      uint16_t *sigCountP)
```

**Parameters**

→ *dbP*
Pointer to an open PRC database from which to get signatures.

← *sigCountP*
The number of signature blocks in the `'sign'` resource.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrInvalidParameter`
*sigCountP* is `NULL`.

`signErrNoSignResource`
No `'sign'` resource exists in the PRC file.

`signErrInvalidSignResource`
The `'sign'` resource is malformed, or invalid in some way.

**See Also** `SignGetNumCertificates()`, `SignGetSignatureByIndex()`

## SignGetOverlayCertIdList Function

**Purpose** Get the list of certificate IDs that will validate an overlay for a signed base PRC.

**Declared In** `SignVfy.h`

**Prototype**
```
status_t SignGetOverlayCertIdList (DmOpenRef dbP,
      uint8_t *certIdList,
   uint32_t *certIdListSize)
```

**Parameters**

→ *dbP*
Pointer to an open PRC database from which to get certificates.

← *certIdList*
Pointer to a byte buffer tht receives the certificate IDs. To determine how large this buffer should be, set this parameter

to `NULL`; upon return `*certIdListSize` will contain the needed buffer size. After allocating a buffer of the proper size, call this function again to obtain the certificate ID list.

↔ *`certIdListSize`*
When calling this function, `*certIdListSize` should contain the size of the buffer indicated by *`certIdList`*. Upon return, it contains the length of the returned certificate data. If a `NULL` pointer was passed in for *`certIdList`*, `signErrBufferTooSmall` is returned and the required length is returned through this parameter.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrInvalidParameter`
*`certIDListSize`* is `NULL`.

`signErrNoSignResource`
No `'sign'` resource exists in the PRC file.

`signErrInvalidSignResource`
The `'sign'` resource is malformed, or invalid in some way.

`signErrBufferTooSmall`
The supplied buffer was too small. The required size has been returned through the *`certIdListSize`* parameter.

`signErrInvalidParameter`
The *`certIdListSize`* parameter was set to `NULL`.

**See Also** `SignGetShLibCertIdList()`

## SignGetShLibCertIdList Function

**Purpose** Get the list of certificate IDs that will validate a shared library (patch) for the signed base PRC.

**Declared In** `SignVfy.h`

**Prototype**
```
status_t SignGetShLibCertIdList (DmOpenRef dbP,
      uint8_t *certIdList,
   uint32_t *certIdListSize)
```

**Parameters** → *`dbP`*
Pointer to an open PRC database from which to get certificates.

← *`certIdList`*

Pointer to a byte buffer tht receives the certificate IDs. To determine how large this buffer should be, set this parameter to `NULL`; upon return `*`*`certIdListSize`* will contain the needed buffer size. After allocating a buffer of the proper size, call this function again to obtain the certificate ID list.

↔ *`certIdListSize`*

When calling this function, `*`*`certIdListSize`* should contain the size of the buffer indicated by *`certIdList`*. Upon return, it contains the length of the returned certificate data. If a `NULL` pointer was passed in for *`certIdList`*, `signErrBufferTooSmall` is returned and the required length is returned through this parameter.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrInvalidParameter`
*`certIDListSize`* is `NULL`.

`signErrNoSignResource`
No `'sign'` resource exists in the PRC file.

`signErrInvalidSignResource`
The `'sign'` resource is malformed, or invalid in some way.

`signErrBufferTooSmall`
The supplied buffer was too small. The required size has been returned through the *`certIdListSize`* parameter.

`signErrInvalidParameter`
The *`certIdListSize`* parameter was set to `NULL`.

**See Also** `SignGetOverlayCertIdList()`

## SignGetSignatureByID Function

**Purpose** Get a signature block given the ID of the signing certificate.

**Declared In** `SignVfy.h`

**Prototype**

```
status_t SignGetSignatureByID (DmOpenRef dbP,
      const SignCertificateIDType certificateID,
      SignSignatureBlockType *signatureBlockP)
```

**Parameters**

→ *dbP*
: Pointer to an open PRC database from which to get signatures.

→ *certificateID*
: The 20-byte ID of the certificate.

← *signatureBlockP*
: The returned signature block, which contains meta-data about the signature.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrInvalidParameter`
: *signatureBlockP* is `NULL`.

`signErrNoSignResource`
: No `'sign'` resource exists in the PRC file.

`signErrInvalidSignResource`
: The `'sign'` resource is malformed, or invalid in some way.

`signErrNotFound`
: A signature with the specified certificate ID was not found.

**Comments** A signature block ID is the ID of the public certificate that can be used to verify the signature.

**See Also** `SignGetCertificateByID()`, `SignGetNumSignatures()`, `SignGetSignatureByIndex()`, `SignVerifySignatureByID()`

## SignGetSignatureByIndex Function

**Purpose** Get a signature block given its index position in the signature block list.

**Declared In** `SignVfy.h`

**Prototype**
```
status_t SignGetSignatureByIndex (DmOpenRef dbP,
      uint16_t index,
      SignSignatureBlockType *signatureBlockP)
```

**Parameters**

→ *dbP*
: Pointer to an open PRC database from which to get signatures.

→ *index*
: The position of the signature within the signature block list.

← *signatureBlockP*
: The returned signature block, which contains meta-data about the signature.

**Returns** Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`signErrInvalidParameter`
: *signatureBlockP* is `NULL`.

`signErrNoSignResource`
: No `'sign'` resource exists in the PRC file.

`signErrInvalidSignResource`
: The `'sign'` resource is malformed, or invalid in some way.

`signErrIndexOutOfBounds`
: The specified index is outside the range of signature indexes for the PRC's signature block.

**Comments** A signature block ID is the ID of the public certificate that can be used to verify the signature.

**See Also** `SignGetCertificateByIndex()`, `SignGetNumSignatures()`, `SignGetSignatureByID()`, `SignVerifySignatureByIndex()`

## SignVerifySignatureByID Function

**Purpose** Verify the signature block referenced by the specified ID. The ID is that of the certificate used for verification of the digital signature block.

**Declared In** `SignVfy.h`

**Prototype**
```
status_t SignVerifySignatureByID (DmOpenRef dbP,
      const SignCertificateIDType certificateID)
```

**Parameters**

→ *dbP*
: Pointer to an open PRC database from which to get signatures.

→ *certificateID*
: The 20-byte ID of the certificate.

**Returns** Returns `errNone` if the signature block is valid, or one of the following if an error occurred:

`signErrNoSignResource`
: No `'sign'` resource exists in the PRC file.

`signErrInvalidSignResource`
: The `'sign'` resource is malformed, or invalid in some way.

`signErrInvalidSignatureBlock`
: The signature block is invalid.

`signErrDigestMismatch`
: The signed digest does not match the calculated PRC digest.

**See Also** `SignGetSignatureByID()`, `SignVerifySignatureByIndex()`

## SignVerifySignatureByIndex Function

**Purpose** Verify the signature block referenced by the specified index.

**Declared In** `SignVfy.h`

**Prototype**
```
status_t SignVerifySignatureByIndex
      (DmOpenRef dbP, uint16_t index)
```

**Parameters**

→ *dbP*
: Pointer to an open PRC database from which to get signatures.

→ *`index`*
The position of the signature within the signature block list.

**Returns** Returns `errNone` if the signature block is valid, or one of the following if an error occurred:

`signErrNoSignResource`
No `'sign'` resource exists in the PRC file.

`signErrInvalidSignResource`
The `'sign'` resource is malformed, or invalid in some way.

`signErrIndexOutOfBounds`
The specified index is outside the range of certificate indexes for the PRC's certificate block.

`signErrInvalidSignatureBlock`
The signature block is invalid.

`signErrDigestMismatch`
The signed digest does not match the calculated PRC digest.

**See Also** `SignGetSignatureByIndex()`, `SignVerifySignatureByID()`

# 16

# SSL Library

This chapter contains reference documentation for the APIs defined in `SslLib.h`. The contents of this chapter are organized into the following sections:



The header file `SslLib.h` declares the API that this chapter describes.

For information on making use of the APIs documented in this chapter, see Chapter 2, "SSL Concepts," on page 55. Much of what you do when working with the SSL library involves working with attributes; reference documentation for the macros that you use to set and get attribute values can be found in Chapter 17, "SSL Library Macros," on page 385.

## SSL Library Structures and Types

### SslAttribute Typedef

**Purpose** Container for an SslLib or SslContext attribute value.

**Declared In** `SslLib.h`

**Prototype** `typedef uint32_t SslAttribute`

**Comments** Attribute values are listed in "SSL Library Macro Constants" on page 385. Descriptions of each attribute can be found under "Attributes" on page 59.

## SslCallback Struct

**Purpose** Structure used when the SSL library transfers control back to the application via a callback function.

**Declared In** `SslLib.h`

**Prototype**

```
struct SslCallback_st {
   void *reserved;
   SslCallbackFunc callback;
   void *data;
   SslContext *ssl;
}
typedef struct SslCallback_st SslCallback
```

**Fields**

`reserved`
  Reserved for internal use by SslLib.

`callback`
  A function pointer of the function to be called. See `SslCallbackFunc()`.

`data`
  Can be set to a value by the application, and will remain unchanged by SslLib. This value will then be available to the callback function. Use this field to communicate context information from the application to the callback. An example of its use is in a 'diagnostic' callback, in which the application could use this field to provide a handle to the logging routines to be used.

`ssl`
  SslLib sets this field to be the `SslContext` if the callback is related to an SslContext. If it is not (due to being related to a SslLib), it is set to `NULL`.

**Comments** This structure is used when the SSL library transfers control back to the application via a callback. A callback is a function that the application supplies to SslLib that will be called when specific situations occur during the SSL protocol. The callbacks are specific to the particular SslContext or SslLib they are registered with. Examples of callbacks used by the SslLib are an 'information' callback and a 'certificate validation' callback.

## SslCipherSuiteInfo Struct

**Purpose** Encapsulates information about the current cipher suite.

**Declared In** `SslLib.h`

**Prototype**

```
typedef struct SslCipherSuiteInfo_st {
   uint8_t cipherSuite[2];
   uint16_t cipher;
   uint16_t digest;
   uint16_t keyExchange;
   uint16_t authentication;
   uint16_t version;
   uint16_t cipherBitLength;
   uint16_t cipherKeyLength;
   uint16_t keyExchangeLength;
   uint16_t authenticationLength;
   uint16_t exportCipher;
} SslCipherSuiteInfo
```

**Fields**

`cipherSuite`
: The two bye cipher suite value for the current cipher suite. See the `CipherSuite` attribute for more details.

`cipher`
: A number indicating which cipher is being used for this connection. One of the `sslCsiCipher...` constants listed under "Cipher Suite Info Constants" on page 344.

`digest`
: The digest value. One of the `sslCsiDigest...` constants listed under "Cipher Suite Info Constants" on page 344.

`keyExchange`
: The key exchange type which was used. One of the `sslCsiKeyExch...` constants listed under "Cipher Suite Info Constants" on page 344.

`authentication`
: The authentication type used. One of the `sslCsiAuth...` constants listed under "Cipher Suite Info Constants" on page 344.

`version`
: The SSL version being used.

`cipherBitLength`
: The number of bits of key material used for encryption key generation. For export ciphers this will be either 40 or 56 bits.

`cipherKeyLength`
: The length of the key for the underlying cipher. For an export RC4 cipher, the `cipherBitLength` would be 40, but the `cipherKeyLength` would be 128. This is because while the SSL protocol would be using 128-bit keys to encrypt and decrypt with RC4, only 40 bits of random data would be used to generate the 128-bit key.

`keyExchangeLength`
: The length in bits of the public key used to establish a shared secret.

`authenticationLength`
: The length of the public key used to ensure the key exchange was not tampered with. For export ciphers, the `keyExchangeLength` is often shorter than the `authenticationLength`.

`exportCipher`
: Set when an export cipher is being used.

**Comments** This structure differs from most others in that the application passes in a structure to be populated from the `SslContext`. Normally the `SslContext` returns a pointer to an internal data structure.

## SslContext Struct

**Purpose** An opaque data structure that represents an SSL connection.

**Declared In** `SslLib.h`

**Prototype** `typedef struct SslContext_st SslContext`

**Fields** None.

## SslIoBuf Struct

**Purpose** Passed to the 'info' functions when I/O is being done.

**Declared In** `SslLib.h`

**Prototype**

```
typedef struct SslIoBuf_st {
   SslContext *ssl;
   uint8_t *ptr;
   uint32_t outNum;
   uint32_t inNum;
   uint32_t max;
   uint32_t err;
   uint32_t flags;
} SslIoBuf
```

**Fields**

`ssl`
: The SslContext for which this callback is being called from. It should be the same value that is held in the `SslCallback` structure's `ssl` field.

`ptr`
: The buffer being used to store the bytes.

`outNum`
: The number of bytes read or written; it is 0 in the 'Before' states.

`inNum`
: The number of bytes to be read or written.

`max`
: The maximum read that could be performed. `inNum` is the number of bytes that the SSL library needs right now, but `max` bytes could be read if they are available. Often the read operation will read more bytes than are needed, so the `outNum` field which returns the number of bytes read or written, can, for the read case, be larger than `inNum`.

`err`
: In the 'After' *`argi`* states, `err` will contain the error code from the I/O operation. If there was no error, it will be 0 (`errNone`).

`flags`
: The flags field is not currently used and is set to zero.

**Comments** The 'Before' *argi* values indicate that the passed `SslIoBuf` indicates the I/O operation about to be performed. The 'After' *argi* values indicate that the `SslIoBuf` contains the results of the just-performed I/O operation.

## SslLib Struct

**Purpose** An opaque data structure that represents the SSL library.

**Declared In** `SslLib.h`

**Prototype** `typedef struct SslLib_st SslLib`

**Fields** None.

## SslSession Struct

**Purpose** Holds all the security information associated with a particular SSL connection.

**Declared In** `SslLib.h`

**Prototype**

```
typedef struct SslSession_st {
   uint32_t length;
   uint16_t version;
   unsigned char cipherSuite[2];
   unsigned char compression;
   unsigned char sessionId[33];
   unsigned char masterSecret[48];
   unsigned char time[8];
   unsigned char timeout[4];
   uint16_t certificateOffset;
   uint16_t extraData;
} SslSession
```

**Fields**

`length`

`version`

`cipherSuite`
: The cipher suite. One of the values listed under "Cipher Suites" on page 344.

`compression`

`sessionId`
  The session ID. The first byte is the length.

`masterSecret`
  Master secret.

`time`
  Host-specific start time.

`timeout`
  Timeout, in seconds.

`certificateOffset`
  Optional Peer certificate; this is the offset from the front of the structure to a `SslExtendedItems` structure. If the offset if 0, it does not exist

`extraData`
  Can be used to store anything, such as the host name of the peer. Application defined.

## SslSocket Struct

**Purpose** The structure used to hold the arguments to be passed to any `sys/socket` calls.

**Declared In** `SslLib.h`

**Prototype**

```
typedef struct SslSocket_st {
   int32_t socket;
   uint16_t flags;
   uint16_t addrLen;
   status_t err;
   int32_t timeout;
   unsigned char addr[8];
} SslSocket
```

**Comments** When setting this structure, the `socket` field will not be copied in. To set the socket to use for network connections, see the Socket attribute. The `socket`, `flags`, `addrLen`, `err`, and `addr` fields all correspond to the arguments passed to `sendto()` and `recvfrom()`. Read further on those functions for more details. The `SslSocket` passed into the `SslContext` will be copied into the

SslContext's SslSocket structure and the SslSocket pointer returned refers to an internal SslContext data structure. When the application calls a function such as SslReceive(), the arguments passed to that function will overwrite the internal SslSocket values, so a subsequent call to SslContextGet_IoStruct() will return the newly updated fields.

# SSL Library Constants

## SSL Open Mode Flags

**Purpose** Flags used to specify how the connection should be started when calling SslOpen().

**Declared In** SslLib.h

**Constants**

`#define sslOpenBufferedReuse 0x0040`
This is the same as `SslContextSet_BufferedReuse(ssl,1);`

`#define sslOpenDelayHandshake 0x0080`
Do not perform the handshake now

`#define sslOpenModeClear 0x0001`
Turn off the SSL protocol.

`#define sslOpenModeSsl 0x0002`
Turn on the SSL protocol.

`#define sslOpenNewConnection 0x0004`
Perform a new SSL handshake, clearing any previous SslSession value. This is the same as `SslContextSet_SslSession(ssl,NULL)`.

`#define sslOpenNoAutoFlush 0x0008`
This is the same as `SslContextSet_AutoFlush(ssl,0);`

`#define sslOpenUseDefaultTimeout 0x0020`
Use the SslContext timeout value instead of timeout parameter.

## SSL Close Mode Flags

**Purpose** Flags that allow you to specify how to perform the SSL Protocol shutdown when calling `SslClose()`.

**Declared In** `SslLib.h`

**Constants**

`#define sslCloseDontSendShutdown 0x0001`
: Do not send the SSL shutdown message to the server.

  `SslContextSet_DontSendShutdown(ssl,1);`

`#define sslCloseDontWaitForShutdown 0x0002`
: Don't wait for the server to send a shutdown message.

  `SslContextSet_DontWaitForShutdown(ssl,1);`

`#define sslCloseUseDefaultTimeout 0x0020`
: Use the timeout value set against the SslContext, not the *`timeout`* parameter.

## Mode Attribute Values

**Purpose** Values that control the SslContext's operating mode.

**Declared In** `SslLib.h`

**Constants**

`#define sslModeClear 0x0000`
: Causes the SSL protocol to be bypassed.

`#define sslModeFlush 0x8000`
: Causes any data in the internal data buffers to be cleared.

`#define sslModeSsl 0x0002`
: A subset value of `sslModeSslClient`

`#define sslModeSslClient 0x000A`

**Comments** `SslModeSsl` is a subset value of `sslModeSslClient`. In a future release of SslLib, the server side of the SSL protocol may be supported in which case `sslModeSslServer` would be added. The application can use code like the following to determine if the SSL protocol is being used:

```
If (SslContextGet_Mode(ssl) & sslModeSsl)
   /* SSL protocol enabled */
else
   /* Using cleartext */
```

A comparison with `sslModeSslClient` could be used to determine if the client or server side of the protocol is being used for that particular SslContext.

The `sslModeFlush` flag is special. When used in `SslContextSet_Mode()`, it causes any data in the internal data buffers to be cleared. This is normally required when reusing a SslContext for a new connection. If an application is using a SslContext for cleartext, and then wants to enable SSL on the same connection, this flag should not be used.

## Protocol Versions

**Purpose** The version of the SSL protocol to use.

**Declared In** `SslLib.h`

**Constants** `#define sslVersionSSLv3 0x0300`
Version SSLv3 of the SSL protocol.

`#define sslVersionTLSv1 0x0301`
Version TLSvs, or SSLv3.1 of the SSL protocol.

**Comments** SslLib sends a TLSv1 ClientHello message by default. Note that in Palm OS Cobalt version 6.0 an attempt to change this protocol version to SSLv3 via `SslContextSet_ProtocolVersion()` has no effect—SslLib continues to send a TLSv1 ClientHello message.

## Protocol Variants

**Purpose** The protocol variants supported by the library.

**Declared In** `SslLib.h`

**Constants** `#define sslSupport_anonDHKeyExchange 0x0008`

`#define sslSupport_Both (sslSupport_SSLv3Protocol | sslSupport_TLSv1Protocol)`
Enables both the SSLv3 and TLSv1 protocols.

`#define sslSupport_DHKeyExchange 0x0002`

`#define sslSupport_DSASign 0x0010`

`#define sslSupport_Ex1024 0x0040`

`#define sslSupport_Ex512 0x0020`

`#define sslSupport_RSAKeyExchange 0x0001`

`#define sslSupport_RSASign 0x0004`

`#define sslSupport_SSLv2Header 0x0400`

```
#define sslSupport_SSLv3 (_sslSupport_SslLib |
  sslSupport_SSLv3Protocol)
```

Enables the SSLv3 protocol.

`#define sslSupport_SSLv3Protocol 0x0100`

```
#define sslSupport_TLSv1 (_sslSupport_SslLib |
  sslSupport_TLSv1Protocol)
```

Enables the TLSv1 protocol.

`#define sslSupport_TLSv1Protocol 0x0200`

**Comments** The protocol variants differ from the protocol version. The ProtocolHello/ProtocolVersion is what you use to talk to the peer, while the protocol variants determine what sections of the library code are turned on or off. These values are used with the `ProtocolSupport` attribute; set this attribute with `SslContextSet_ProtocolSupport()` and get it with `SslContextGet_ProtocolSupport()`.

**WARNING!** Do not disable things that you don't know anything about. Also, do not turn off the `Ex512/Ex1024` bits without also removing the relevant ciphers from the cipher suite list.

Use `sslSupport_SSLv3`, `sslSupport_TLSv1`, or `sslSupport_Both` to enable SSLv3, TLSv1, or both protocols. The default is `sslSupport_Both`.

## Compatibility Flags

**Purpose** Compat attribute flags that turn on compatibility with incorrect SSL protocol implementations.

**Declared In** `SslLib.h`

**Constants**

`#define sslCompat1RecordPerMessage 0x0004`
Some servers do not like to receive SSL protocol messages separated into multiple SSL records. This option stops the write buffers being set to a size less than 1024 bytes which ensures this problem will not occur.

`#define sslCompatAll 0xffff`
This value enables all the bug compatibility flags.

`#define sslCompatBigRecords 0x0008`
Some old Microsoft servers would send data to the SSL Client in records larger than 16k bytes in size. This is not legal in the SSLv3 protocol. This flag makes SslLib tolerate these large records.

`#define sslCompatNetscapeCaDnBug 0x0002`
Enables support for some old versions of Netscape servers which encoded certificate requests Distinguished names wrongly. This is not currently a problem for SslLib since it does not support Client certificates.

`#define sslCompatReuseCipherBug 0x0001`
Enables support for servers that change cipher suites on session-reuse. They should not be doing this.

**Comments** These bugs will not normally be encountered while using the SSL protocol, but if desired, it is worth enabling the compatibility in case old buggy servers are being accessed.

## SSL Callback Commands

**Purpose** General commands that all callbacks should expect to receive. There are normally related to creation and destruction of the structure that holds the callback. These commands are used in conjunction with `SslCallbackFunc()`.

**Declared In** `SslLib.h`

**Constants**

`#define sslCmdFree 0x0002`
: Called when the callback is 'destroyed', normally due to its parent SslLib or SslContext being destroyed.

`#define sslCmdGet 0x0004`
: Called to return a value from the callback to SslLib. The *`argi`* value for this callback is callback-specific.

`#define sslCmdInfo 0x0012`
: An Info callback. See "The Info Callback" on page 375 for more information.

`#define sslCmdNew 0x0001`
: Called when the callback is 'copied' into an SslLib or SslContext.

`#define sslCmdRead 0x0010`

`#define sslCmdReset 0x0003`
: Called when the SslContext has been reset, which occurs when a new connection is being started. It is called instead of `sslCmdNew`. `sslCmdNew` will be called only once; `sslCmdReset` will be called subsequently to 'reset' the state associated with the callback.

`#define sslCmdSet 0x0005`
: Called to pass a value from SslLib to the callback. The *`argi`* value would specify the *`argv`* parameter. The *`argi`* parameter would be callback-specific.

`#define sslCmdVerify 0x0013`
: A Verify callback. See "The Verify Callback" on page 376 for more information.

`#define sslCmdWrite 0x0011`

## Cipher Suite Info Constants

**Purpose** Constants used with various fields of the `SslCipherSuiteInfo` structure.

**Declared In** `SslLib.h`

**Constants**

```
#define sslCsiAuthNULL 0x00

#define sslCsiAuthRsa 0x01

#define sslCsiCipherNull 0x00

#define sslCsiCipherRc4 0x01

#define sslCsiDigestMd2 0x03

#define sslCsiDigestMd5 0x01

#define sslCsiDigestNull 0x00

#define sslCsiDigestSha1 0x02

#define sslCsiKeyExchNull 0x00

#define sslCsiKeyExchRsa 0x01
```

## Cipher Suites

**Purpose** SSL cipher suites that the SSL protocol can attempt to use.

**Declared In** `SslLib.h`

**Constants**

```
#define sslCs_RSA_3DES_168_SHA1 0x00, 0x0A
#define sslCs_RSA_DES_40_SHA1 0x00, 0x08
#define sslCs_RSA_DES_56_SHA1 0x00, 0x09
#define sslCs_RSA_RC4_128_MD5 0x00,0x04
```

```
#define sslCs_RSA_RC4_128_SHA1 0x00,0x05
#define sslCs_RSA_RC4_40_MD5 0x00,0x03
#define sslCs_RSA_RC4_56_SHA1 0x00,0x64
```

**Comments** Set the cipher suites using either `SslLibSet_CipherSuites()` or `SslContextSet_CipherSuites()`.

## Ciphers

**Purpose**

**Declared In** `SslLib.h`

**Constants**

```
#define sslCs_ExportCiphers
   "\x00\x06\x00\x64\x00\x03\x00\x08"

#define sslCs_StrongCiphers
   "\x00\x06\x00\x0A\x00\x05\x00\x04"

#define sslCs_WeakExportCiphers
   "\x00\x04\x00\x03\x00\x08"
```

## Info Callbacks

**Purpose** The `SslCallbackFunc()` callback is called when various situations occur during the usage of a SslContext. It is primarily intended for debugging and feedback purposes. If the callback returns a non-zero value, this error will be returned back out the SslLib API. The callback will be called with a command argument of `sslCmdInfo`. The constants listed in this section represent the possible *`argi`* values.

**Declared In** `SslLib.h`

**Constants**

`#define sslArgInfoAlert 0x0002`
: Notification of an Alert in the SSL protocol. The `sslArgInfoAlert` notification is called with a `NULL` value for the *`argv`* parameter. The application can get the

LastAlert attribute from the SslContext to determine which alert was received.

`#define sslArgInfoCert 0x0003`
Notification of peer certificate. The `sslArgInfoCert` call is made after the server's certificate chain has been verified. The *argv* parameter is a `SslExtendedItems` pointer, which points to the remote server's certificate.

`#define sslArgInfoHandshake 0x0001`
Notification of a state change in the SSL protocol. The `sslArgInfoHandshake` will be called upon each handshake state change. The *argv* parameter will be `NULL`, but the `HsState` attribute can be interrogated to read the current state.

`#define sslArgInfoReadAfter (sslCmdRead | 0x8000)`
Notification after a `recvfrom()`, `recv()`, or `read()` `sys/socket` call. See the Comments section, below, for more information.

`#define sslArgInfoReadBefore sslCmdRead`
Notification before a `recvfrom()`, `recv()`, or `read()` `sys/socket` call. See the Comments section, below, for more information.

`#define sslArgInfoWriteAfter (sslCmdWrite | 0x8000)`
Notification after a `sendto()`, `send()`, or `write()` `sys/socket` call. See the Comments section, below, for more information.

`#define sslArgInfoWriteBefore sslCmdWrite`
Notification before a `sendto()`, `send()`, or `write()` `sys/socket` call. See the Comments section, below, for more information.

**Comments** The `sslArgInfo[Read | Write] [Before | After]` callback is called twice for each network I/O operation. The first call is made before the call to the underlying `sys/socket` send or receive function. The second is made after the call has completed. If the callback returns a non-zero value, this value will be returned by original SslLib call the application made.

The *argv* parameter is a `SslIoBuf` structure. This structure's `ssl` field is the SslContext that the I/O operation is being performed by.

`ptr` points to the space used, or to be used, in the operation. `outNum` is the number of bytes processed. It is only set in the After calls. `inNum` is the number of bytes to be read or written in the call. `max` is the maximum number of bytes that could be read. It can be larger than `inNum`. `err` is the error value, if any. This value is only set in the After calls. `flags` is currently not used and is set to 0.

## InfoInterest Values

**Purpose** Values used to specify the events for which of the Info Callbacks will be called. The `InfoInterest` value is the logical OR of these values.

**Declared In** `SslLib.h`

**Constants** `#define sslFlgInfoAlert 0x0001`
The `sslArgInfoAlert` callback.

`#define sslFlgInfoCert 0x0008`
The `sslArgInfoCert` callback.

`#define sslFlgInfoHandshake 0x0002`
The `sslArgInfoHandshake` callback.

`#define sslFlgInfoIo 0x0004`
The `sslArgInfoReadAfter`, `sslArgInfoReadBefore`, `sslArgInfoWriteAfter`, and `sslArgInfoWriteBefore` callbacks.

## LastApi Attribute Values

**Purpose** The last SslLib API call made. This attribute can be useful in event-driven programs.

**Declared In** `SslLib.h`

`#define sslLastApiFlush 0x04`

`#define sslLastApiNone 0x00`

`#define sslLastApiOpen 0x01`

`#define sslLastApiRead 0x02`
: Set if `SslRead()`, `SslPeek()` or `SslReceive()` was last called.

`#define sslLastApiShutdown 0x05`

`#define sslLastApiWrite 0x03`
: Set if `SslWrite()` or `SslSend()` was last called.

## LastIO Attribute Values

**Purpose** The last network operation performed.

**Declared In** `SslLib.h`

`#define sslLastIoNone 0x00`
: No I/O operations have been performed since the context was last reset.

`#define sslLastIoRead 0x01`
: A read operation.

`#define sslLastIoWrite 0x02`
: A write operation.

**Comments** Since most of the SslLib API I/O functions can cause an SSL handshake to be performed, it is often not possible to know if the reason that a `SslSend()` returned `netErrWouldBlock` is because the Send operation failed or a Receive operation failed (because a SSL Handshake was being performed). This attribute allows the application to determine which I/O operation was being called if a network error is returned. If the application is using `select()`, this attribute is very important. Because this attribute returns the last network operation performed, `sslLastIoNone` will only be returned if the SslContext has not performed any I/O operations since its last reset.

## SSL Protocol States

**Purpose** These constants indicate the state that the SSL protocol is currently in. See the SSL protocol specification for clarification on what the values mean.

**Declared In** `SslLib.h`

**Constants**

```
#define sslHsStateCert 7
#define sslHsStateCertB 8
#define sslHsStateCertReq 13
#define sslHsStateCertReqB 14
#define sslHsStateCkEx 17
#define sslHsStateCleanup 25
#define sslHsStateClientCert 16
#define sslHsStateClientHello 2
#define sslHsStateClosed 28
#define sslHsStateDone 26
#define sslHsStateFinished 19
#define sslHsStateFlush 4
#define sslHsStateGenerateKeys 21
#define sslHsStateHelloRequest 29
#define sslHsStateNone 0
#define sslHsStateReadCcs 20
#define sslHsStateReadFinished 22
#define sslHsStateReadFinishedB 23
#define sslHsStateReadFinishedC 24
#define sslHsStateServerDone 15
#define sslHsStateServerHello 3
#define sslHsStateShutdown 27
#define sslHsStateSkEx 9
#define sslHsStateSkExAnonDh 12
```

```
#define sslHsStateSkExDh 11
#define sslHsStateSkExRsa 10
#define sslHsStateStart 1
#define sslHsStateWrite 6
#define sslHsStateWriteCcs 18
#define sslHsStateWriteClose 30
#define sslHsStateWriteFlush 5
```

## SSL Server Alerts

**Purpose** Alert values received from the server. These are the defined Sslv3/ TLSv1 alerts as defined in the SSLv3 and TLSv1 specifications. For their meanings, refer to those specifications.

**Declared In** `SslLib.h`

**Constants**

```
#define sslAlertAccessDenied (0x0200+49)
#define sslAlertBadCertificate (0x0100+42)
#define sslAlertBadRecordMac (0x0200+20)
#define sslAlertCertificateExpired (0x0100+45)
#define sslAlertCertificateRevoked (0x0100+44)
#define sslAlertCertificateUnknown (0x0100+46)
#define sslAlertCloseNotify (0x0100+ 0)
#define sslAlertDecodeError (0x0200+50)
#define sslAlertDecompressionFailure (0x0200+30)
#define sslAlertDecryptError (0x0200+51)
#define sslAlertDecryptionFailed (0x0200+21)
#define sslAlertExportRestricion (0x0200+60)
#define sslAlertHandshakeFailure (0x0200+40)
#define sslAlertIllegalParameter (0x0200+47)
#define sslAlertInsufficientSecurity (0x0200+71)
```

```
#define sslAlertInternalError (0x0200+80)
#define sslAlertNoCertificate (0x0100+41)
#define sslAlertNoRenegotiation (0x0100+100)
#define sslAlertProtocolVersion (0x0200+70)
#define sslAlertRecordOverflow (0x0200+22)
#define sslAlertUnexpectedMessage (0x0200+10)
#define sslAlertUnknownCa (0x0200+48)
#define sslAlertUnsupportedCertificate (0x0100+43)
#define sslAlertUserCancled (0x0100+90)
```

Comments
The alert values are received from the server and are of two types, fatal and non-fatal.

The non-fatal alerts have a value of the form `0x01`*XX*, while fatal Alerts have the form `0x02`*XX*. SslLib will fail on fatal alerts and continue on non-fatal alerts.

## SSL Library Errors

**Purpose** Error codes returned by the various SSL library functions.

**Declared In** `SslLib.h`

**Constants**

`#define sslErrBadArgument (sslErrorClass+17)`
An invalid argument was provided to the function.

`#define sslErrBadDecode (sslErrorClass+9)`
An error occurred while decoding values during certificate verification.

`#define sslErrBadLength (sslErrorClass+13)`
A length argument was invalid.

`#define sslErrBadOption (sslErrorClass+18)`
An invalid argument was provided to the function.

`#define sslErrBadPeerFinished (sslErrorClass+46)`
The final check of the SSL handshake failed. This indicates that there was a problem establishing a shared secret value. It could be caused by the server using a certificate that does not match its private key.

`#define sslErrBadSignature (sslErrorClass+47)`
An invalid signature was found on a ephemeral Cipher Suite message.

`#define sslErrBufferTooSmall (sslErrorClass+11)`
A supplied buffer was not large enough for the output data.

`#define sslErrCbAbort (sslErrorClass+4)`
This error code would be returned by an applications callback function to indicate a desire to exit. This error may not be fatal, depending on the callback that generated the error.

`#define sslErrCert (sslErrorClass+39)`
A generic error occurred inside the SslLib certificate library.

`#define sslErrCertDecodeError (sslErrorClass+51)`
The Servers certificate could not be decoded.

`#define sslErrCsp (sslErrorClass+38)`
A generic error occurred inside the SslLib cryptographic library.

`#define sslErrDivByZero (sslErrorClass+7)`
Something went wrong in the Math library. These error will normally only be generated by certificates which have invalid public keys.

`#define sslErrEof (sslErrorClass+2)`
Error returned by SslLib functions when either the SSL protocol has been closed or the underlying socket has been closed. This error indicates that the current SslContext is unable to read or write any more data bytes.

`#define sslErrExtraHandshakeData (sslErrorClass+43)`
Extra data was found in the SSL handshake messages that should not have been there.

`#define sslErrFailed (sslErrorClass+1)`
A generic error.

`#define sslErrFatalAlert (sslErrorClass+45)`
A fatal alert was received by the SSL protocol.

`#define sslErrHandshakeEncoding (sslErrorClass+40)`
An error occurred during decoding of SSL handshake messages.

`#define sslErrHandshakeProtocol (sslErrorClass+42)`
An error occurred while processing the decoded SSL handshake messages.

`#define sslErrInitNotCalled (sslErrorClass+10)`
An internal SslLib error.

`#define sslErrInternalError (sslErrorClass+21)`
An internal SslLib error.

`#define sslErrIo (sslErrorClass+5)`
This error code is returned when an underlying `sys/socket` function call returned an error that is not fatal. A timeout, or other such non-fatal network errors will be reclassified as this error type. A function that returns this error type can be re-called once the error condition has disappeared.

`#define sslErrMissingCipherSuite (sslErrorClass+80)`

`#define sslErrMissingProvider (sslErrorClass+41)`
An internal SslLib error.

`#define sslErrNoDmem (sslErrorClass+14)`
An internal SslLib error.

`#define sslErrNoMethodSet (sslErrorClass+15)`
An internal SslLib error.

`#define sslErrNoModInverse (sslErrorClass+8)`
Something went wrong in the Math library. These error will normally only be generated by certificates which have invalid public keys.

`#define sslErrNoRandom (sslErrorClass+16)`
A problem with the random number source.

`#define sslErrNotFound (sslErrorClass+6)`
Returned on an internal SslLib search that did not find a valid entry. Consider this an internal SslLib error.

`#define sslErrNotImplemented (sslErrorClass+19)`
An internal SslLib error.

`#define sslErrNullArg (sslErrorClass+12)`
An passed argument was `NULL` that should not have been `NULL`.

`#define sslErrOk (sslErrorClass+0)`
Not an error.

`#define sslErrOutOfMemory (sslErrorClass+3)`
Returned if a dynamic memory allocation failed. This is normally considered a very bad error.

`#define sslErrReadAppData (sslErrorClass+50)`
Application data was read by the SSL protocol when it was expecting handshake messages.

`#define sslErrReallocStaticData (sslErrorClass+20)`
An internal SslLib error

`#define sslErrRecordError (sslErrorClass+37)`
An invalid record was received in the SslContext.

`#define sslErrUnexpectedRecord (sslErrorClass+49)`
A record of the wrong was received inside the SSL protocol.

`#define sslErrUnsupportedCertType (sslErrorClass+52)`
The Servers certificate contains a public key we cannot decode.

`#define sslErrUnsupportedProtocol (sslErrorClass+54)`

`#define sslErrUnsupportedSignatureType (sslErrorClass+53)`
We have been send a certificate with a signature type we do not recognize.

`#define sslErrVerifyCallback (sslErrorClass+128)`

`#define sslErrWrongMessage (sslErrorClass+44)`
An invalid SSL message was received.

### Miscellaneous SSL Library Constants

**Purpose** These constants are also defined in `SslLib.h`.

**Declared In** `SslLib.h`

**Constants**

`#define kSslDBName "SslLib"`
The SSL library's database name.

`#define kSslLibCreator 'ssl0'`
The SSL library's creator ID.

`#define kSslLibType sysFileTLibrary`
The SSL library's type.

# SSL Library Functions

### SslClose Function

**Purpose** Performs the shutdown part of the SSL protocol.

**Declared In** `SslLib.h`

**Prototype**

```
status_t SslClose (SslContext *ctx,
    uint16_t mode, uint32_t timeout)
```

**Parameters**

→ *ctx*
The context to query.

→ *mode*
Flags that specify how to perform the SSL Protocol shutdown. A combination of the values listed under "SSL Close Mode Flags" on page 339.

→ *timeout*
Timeout, in system ticks, to use for final message exchange

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** This usually involves message exchanges. This function can be repeatedly called after a timeout until either a network error is reported or the final SSL shutdown message exchange has been completed. The mode values can be logically OR'ed together.

**Example** The following code excerpt show how you might use this function:

```
Err err;
SslContext *ssl;

err = SslOpen(ssl,0,20*SysTicksPerSecond());
/* Perform SSL IO */
/* Shutdown the protocol but don't linger waiting for a
 * response from the server */
err = SslClose(ssl,sslCloseDontWaitForshutdown,
   20*SysTicksPerSecond());
```

**See Also** SslOpen(), SslContextSetLong(), SslContextSetPtr()

## SslConsume Function

**Purpose** Removes up to a specified number of bytes from the buffered read bytes in the passed `SslContext`.

**Declared In** `SslLib.h`

**Prototype** `void SslConsume (SslContext *ctx, int32_t number)`

**Parameters**

→ *ctx*
: The SslContext to operate on.

→ *number*
: The number of bytes to remove from the internal buffer.

**Returns** Nothing.

**Comments** This function is normally used in conjunction with SslPeek().

**Example** The following code excerpt shows how this function might be used.

```
Err err;
void *data;
Int32 *dataLen;

err=SslPeek(ssl,&data,*dataLen,16*1024);
/* Process the dataLen bytes located at data */
SslConsume(ssl,dataLen);
```

**See Also** SslPeek(), SslRead(), SslReceive()

## SslContextCreate Function

**Purpose** Creates a new SSL Context.

**Declared In** `SslLib.h`

**Prototype**
```
status_t SslContextCreate (SslLib *lib,
    SslContext **ctx)
```

**Parameters**

→ *lib*
The SSL library structure.

← *ctx*
Where to deposit the SslContext pointer.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** A SslContext is the data structure used to encapsulates all aspects of a SSL connection. This routine will deposit a pointer to the newly created structure at the address given by the *ctx* argument. Various default values will be inherited from the passed SslLib.

**See Also** `SslOpen()`, `SslContextDestroy()`, `SslContextSetLong()`, `SslContextSetPtr()`

## SslContextDestroy Function

**Purpose** Destroys the SSL Context.

**Declared In** `SslLib.h`

**Prototype**
```
void SslContextDestroy (SslContext *ctx)
```

**Parameters**

→ *ctx*
The SslContext to destroy.

**Returns** Nothing.

**Comments** This routine will free the memory associated with the passed SslContext. This function will not close the network connection or shutdown the SSL protocol. See `SslClose()` for information on shutting down the SSL Protocol.

**See Also** `SslContextCreate()`, `SslClose()`

## SslContextGetLong Function

**Purpose** Retrieve an integer attribute value from the passed SslContext structure.

**Declared In** `SslLib.h`

**Prototype**
```
int32_t SslContextGetLong (SslContext *lib,
   SslAttribute attr)
```

**Parameters**

→ *lib*
: The SslContext from which the value is to be retrieved.

→ *attr*
: Attribute to retrieve.

**Returns** The value of the attribute is returned. If a non-existent attribute was requested, -1 is returned. This could give incorrect values so an application should make sure to call this routine with the correct arguments.

**Comments** This function is not normally used directly, but via pre-defined macros.

**Example** The following example shows the use of one of the macros that make use of this function:

```
/* Is the SslContext configured to do ssl? */
if (!(SslContextGet_Mode(lib) & sslModeSsl))
   return(WE_ARE_NOT_USING_SSL);
```

**See Also** `SslContextSetLong()`, `SslContextGetLong()`

## SslContextGetPtr Function

**Purpose** Retrieve a pointer to an attribute value from the passed SslContext structure.

**Declared In** `SslLib.h`

**Prototype**
```
status_t SslContextGetPtr (SslContext *lib,
   SslAttribute attr, void **value)
```

**Parameters**

→ *lib*
: The SslContext to retrieve the attribute from.

→ *attr*
The attribute to retrieve

← *value*
A pointer to an attribute specific pointer

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** This function is not normally used directly, but via pre-defined macros. The type of the pointer returned is specific to the attribute being requested.

**Example** The following example shows the use of one of the macros that makes use of this function:

```
SslSession *session;
Err err;

err = SslContextGet_SslSession(ssl,&session);
```

**See Also** `SslContextSetPtr()`, `SslLibGetPtr()`

## SslContextSetLong Function

**Purpose** Modify one of the numeric attributes of a SslContext structure.

**Declared In** `SslLib.h`

**Prototype**
```
status_t SslContextSetLong (SslContext *lib,
    SslAttribute attr, long value)
```

**Parameters**

→ *lib*
The SslContext on which to operate.

→ *attr*
The attribute to modify.

→ *value*
The new value.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** This function is not normally used directly, rather it is used via pre-defined macros. The `attr` parameter specifies the SslContext attribute that will be set to the value passed in `value`.

**Example** The following code excerpt shows how this function is used.

```
SslContext *ssl;
Err err;

err = SslContextCreate(lib,&ssl);
/*modify output buffer size */
err = SslContextSet_WbufSize(lib,8*1024);
```

**See Also** `SslContextSetPtr()`, `SslContextGetPtr()`, `SslContextGetLong()`

## SslContextSetPtr Function

**Purpose** Update one of the non-integer attributes of a SslContext.

**Declared In** `SslLib.h`

**Prototype**
```
status_t SslContextSetPtr (SslContext *lib,
   SslAttribute attr, void *value)
```

**Parameters**

→ *lib*
The SslContext to modify.

→ *attr*
The attribute to update.

→ *value*
The value to update, specific to the SslAttribute.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** The `attr` value defines the type of the `value` parameter. This function is not normally used directly, rather it is used via pre-defined macros.

**Example** The following code excerpt shows how this function is used.

```
SslContext *ssl;
SslCallback cb;
```

```
Err err;

err = SslContextCreate(lib,&ssl);

/* Configure to have 'debugCallbackFunction' called for each
 * SSL protocol handshake state change */
cb.callback = debugCallbackFunction;
cb.data = NULL;
err = SslContextSet_InfoInterest(lib, sslInfoHandshake);
err = SslContextSet_InfoCallback(lib,(void *)&cb);
```

**See Also** `SslContextGetLong()`, `SslContextSetLong()`, `SslContextGetPtr()`

## SslFlush Function

**Purpose** Cause an immediate write of any data buffered in the SslContext to the network.

**Declared In** `SslLib.h`

**Prototype**
```
status_t SslFlush (SslContext *ctx,
    int32_t *outstanding)
```

**Parameters**

→ *ctx*
: The SslContext to operate on.

← *outstanding*
: The number of byte still unflushed after this call.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** An SslContext can be set into "no AutoFlush" mode. This means that the `SslSend()` and `SslWrite()` operations will not cause an immediate write to the network. If this mode is enabled, then explicit `SslFlush` calls need to be made to ensure that the data buffered in the SslContext is sent to the network. The main use of "no AutoFlush" is to allow multiple `SslWrite()/SslSend()` commands to have their output buffered in the SslContext's output buffer. This improves the SSL Protocols efficiency and is generally a good policy if lots of small write operations are being performed by the application.   The number of bytes that can be written to the

SslContext write buffer is a few tens of bytes less than the output buffer size. This means that if the application is writing less than this number of bytes, no network errors can occur until the `SslFlush()` call is made. The outstanding parameter will be updated to contain the number of buffered bytes that are still buffered in the SslContext. If this value is non-zero, the next `SslWrite()`, `SslSend()`, `SslFlush()` operation will attempt to write those bytes to the network.

**Example** The following code excerpt shows how this function can be used:

```
Err err;

SslContextSet_AutoFlush(ssl,0);
SslWrite(ssl,”GET “,4);
SslWrite(ssl,url,StrLen(url));
SslWrite(ssl,” HTTP/1.0\r\n\r\n”,13);

err=SslFlush(ssl,NULL);
```

**See Also** `SslWrite()`, `SslSend()`, `SslContextSetLong()`

## SslLibClose Function

**Purpose** SSL library’s shared library close function.

**Declared In** `SslLib.h`

**Prototype** `status_t SslLibClose (void)`

**Parameters** None.

**Returns**

## SslLibCreate Function

**Purpose** Creates the SSL library context.

**Declared In** `SslLib.h`

**Prototype** `status_t SslLibCreate (SslLib **lib)`

**Parameters** ← *lib*

`Where to deposit the SslLib pointer`

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** This routine will deposit the newly-created SslLib at the address given by the *`lib`* argument. This routine is generally the first call made when performing SSL functionality. Various default configuration values can be set against this structure. These values will be inherited by SslContext structures created against the SslLib.

**See Also** `SslLibDestroy()`, `SslContextCreate()`, `SslLibSetLong()`, `SslLibSetPtr()`

## SslLibDestroy Function

**Purpose** Destroys the context represented by *`lib`*.

**Declared In** `SslLib.h`

**Prototype** `void SslLibDestroy (SslLib *lib)`

**Parameters** → *`lib`*
SslLib structure to be destroyed.

**Returns** Nothing.

**See Also** `SslLibCreate()`

## SslLibGetLong Function

**Purpose** Retrieve an integer attribute value from the passed SslLib structure.

**Declared In** `SslLib.h`

**Prototype**
```
int32_t SslLibGetLong (SslLib *lib,
    SslAttribute attr)
```

**Parameters** → *`lib`*
The SslLib from which the value is to be retrieved.

→ *`attr`*
Attribute to retrieve.

**Returns** The value of the attribute is returned. If a non-existent attribute was requested, -1 is returned.

**Comments** This function is not normally used directly, but via pre-defined macros.

**See Also** `SslContextGetPtr()`, `SslLibSetLong()`, `SslContextGetLong()`

## SslLibGetPtr Function

**Purpose** Retrieve an pointer attribute value from the passed SslLib structure.

**Declared In** `SslLib.h`

**Prototype**

```
status_t SslLibGetPtr (SslLib *lib,
   SslAttribute attr, void **value)
```

**Parameters**

→ *lib*
The SslLib to retrieve the attribute from.

→ *attr*
The attribute to retrieve.

← *value*
A pointer to a attribute specific pointer

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** This function is not normally used directly, but via pre-defined macros. The type of the pointer returned is specific to the attribute being requested.

**Example** The following code excerpt shows how this function is used.

```
SslCallback *cb;
Err err;

err = SslLibGet_InfoCallback(lib,&cb);
```

**See Also** `SslLibSetPtr()`, `SslContextGetPtr()`

## SslLibName Function

**Purpose**

**Declared In** `SslLib.h`

**Prototype** `status_t SslLibName (void)`

**Parameters** None.

**Returns**

## SslLibOpen Function

**Purpose** SSL library's shared library open function.

**Declared In** `SslLib.h`

**Prototype** `status_t SslLibOpen (void)`

**Parameters** None.

**Returns**

## SslLibSetLong Function

**Purpose** Modify one of the numeric attributes of a SslLib structure.

**Declared In** `SslLib.h`

**Prototype**

```
status_t SslLibSetLong (SslLib *lib,
    SslAttribute attr, int32_t value)
```

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *attr*
The attribute to modify.

→ *value*
The new value.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** This function is not normally used directly, rather it is used via pre-defined macros. The `attr` parameter specifies the SslLib attribute that will be set to the value passed in `value`.

**Example** The following code excerpt shows how this function is used.

```
SslLib *lib;
Err err;

err = SslLibCreate(&lib);
err = SslLibSet_AutoFlush(lib,0); /* Turn of auto flushing */
```

**See Also** `SslLibCreate()`, `SslLibSetPtr()`, `SslLibGetPtr()`, `SslLibGetLong()`

## SslLibSetPtr Function

**Purpose** Update one of the non-integer attributes of a SslLib.

**Declared In** `SslLib.h`

**Prototype**
```
status_t SslLibSetPtr (SslLib *lib,
   SslAttribute attr, void *value)
```

**Parameters**

→ *lib*
The SslLib to operate on.

→ *attr*
The attribute to update.

→ *value*
The value to update, specific to the SslAttribute.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under “SSL Library Errors” on page 351.

**Comments** The `attr` value defines the type of the `value` parameter. This function is not normally used directly, rather it is used via pre-defined macros.

**Example** The following code excerpt shows how this function is used.

```
SslLib *lib;
SslCallback cb;
Err err;
```

```
err = SslLibCreate(&lib);

/* Configure to have 'debugCallbackFunction' called for each
 * SSL protocol handshake state change */
cb.callback = debugCallbackFunction;
cb.data = NULL;
err = SslLibSet_InfoInterest(lib, sslInfoHandshake);
err = SslLibSet_InfoCallback(lib,(void *)&cb);
```

**See Also** `SslLibGetLong()`, `SslLibSetLong()`, `SslLibGetPtr()`

## SslLibSleep Function

**Purpose** SSL library's shared library sleep function.

**Declared In** `SslLib.h`

**Prototype** `status_t SslLibSleep (void)`

**Parameters** None.

**Returns**

## SslLibWake Function

**Purpose** SSL library's shared library wake function.

**Declared In** `SslLib.h`

**Prototype** `status_t SslLibWake (void)`

**Parameters** None.

**Returns**

## SslOpen Function

**Purpose** Initializes the passed SslContext.

**Declared In** `SslLib.h`

**Prototype**
```
status_t SslOpen (SslContext *ctx, uint16_t mode,
   uint32_t timeout)
```

**Parameters**

↔ *ctx*
The SslContext to start a SSL Handshake with.

→ *mode*
How we should 'start' this connection. A combination of the values listed under "SSL Open Mode Flags" on page 338.

→ *timeout*
Optional timeout (in system ticks).

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** Depending on the mode, `SslOpen()` may or may not actually send the handshake messages during this function call, but may delay the handshake until the first `SslSend()`/`SslReceive()`. It may not even use the SSL protocol. The mode values can be logically OR'ed together, and their values affect the functionality of the `SslOpen()` call. If non-fatal network errors occur (timeouts), the function can be re-called. When the function finally returns `errNone`, the SSL handshake will have completed and any `SslContext` will be able to return a valid `SslSession` structure. If one re-calls `SslOpen()`, make sure to not have either of `sslOpenModeClear` or `SslOpenModeSsl` set or the connection will be reset.

Quite a few of the mode parameters set flags against the `SslContext`.

**Returns** `SslClose()`, `SslContextSetLong()`, `SslContextSetPtr()`

## SslPeek Function

**Purpose** Obtains a pointer into the buffered data that is located in the SslContext.

**Declared In** `SslLib.h`

**Prototype**

```
status_t SslPeek (SslContext *ctx,
    void **buffer_ptr, int32_t *availableBytes,
    int32_t readSize)
```

**Parameters**

→ *ctx*
The SslContext to operate on.

← *buffer_ptr*
The location to place the data pointer.

← *availableBytes*
The location to place the number of bytes available in `*buffer_ptr`.

→ *readSize*
The maximum number of bytes to return.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, this function returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** This function returns a pointer to available data bytes and assigns the number available to *availableBytes*. This function does not copy any data bytes from the `SslContext`, rather it returns a pointer into the buffered data that is located in the `SslContext`. If there were no data bytes in the `SslContext`, data will be read from the network until there are data bytes available. Repeated calls to `SslPeek()` will return the same *buffer_ptr* value until a `SslConsume()` call is done to indicate that bytes no longer need to be buffered. *readSize* is the maximum number of available bytes that will be reported as being available.

This is a more advanced function but is used internally, along with `SslConsume()`, to build the `SslRead()` and `SslReceive()` functions. Its main use is for 'streaming' input data where the application does not need to allocate it's own data storage buffers since it can read directly from the `SslContext` buffers. Once a quantity of data is reported as available in `*availableBytes`, the total will not increase until that number of bytes has been 'consumed'.

**See Also** `SslConsume()`, `SslRead()`

## SslRead Function

**Purpose** Receives data.

**Declared In** `SslLib.h`

**Prototype**
```
int32_t SslRead (SslContext *ctx, void *buffer,
   int32_t bufferLen, status_t *errRet)
```

**Parameters**

→ *ctx*
: The SslContext to read from.

← *buffer*
: Buffer into which read data will be placed.

→ *bufferLen*
: Size of buffer (max bytes read).

← *errRet*
: This will contain an error code if return is -1.

**Returns** Returns the number of bytes successfully received, or -1 if an error occurred.

**Comments** Performs the same functionality as `SslReceive()`. This call will use the timeout set earlier against the `SslContext`.

**See Also** `SslWrite()`, `SslSend()`

## SslReceive Function

**Purpose** Receives data.

**Declared In** `SslLib.h`

**Prototype**
```
int16_t SslReceive (SslContext *ctx, void *buffer,
   uint16_t bufferLen, uint16_t flags,
   void *fromAddr, uint16_t *fromLen,
   int32_t timeout, status_t *errRet)
```

**Parameters**

→ *ctx*
: The SslContext to use.

← *buffer*
Buffer into which received data will be placed.

→ *bufferLen*
Size of buffer (max bytes received).

→ *flags*
One or more `MSG_`*xxx* flags (defined in `sys/socket.h`).

← *fromAddr*
Buffer to hold address of sender (`sockaddr`).

↔ *fromLen*
On entry, size of *fromAddr* buffer. On exit, actual size of returned address in *fromAddr* buffer

→ *timeout*
Max timeout in system ticks. -1 means wait forever.

← *errRet*
This will contain an error code if return is -1.

---

**NOTE:** In Palm OS Cobalt the *flags* and *timeout* parameter values are ignored.

---

**Returns** Returns the number of bytes successfully received, or -1 if an error occurred.

**Comments** The function returns either the number of bytes successfully received or -1. If -1, there was an error. In that case, an error code will be deposited at the address given by *errRet*.

**See Also** `SslSend()`, `read()`

## SslSend Function

**Purpose** Sends data over the network.

**Declared In** `SslLib.h`

**Prototype**

```
int16_t SslSend (SslContext *ctx,
   const void *buffer, uint16_t bufferLen,
   uint16_t flags, void *toAddr, uint16_t toLen,
   int32_t timeout, status_t *errRet)
```

**Parameters**

→ *ctx*
The SslContext to use.

→ *buffer*
Buffer containing data to send.

→ *bufferLen*
Length, in bytes, of data to send.

→ *flags*
One or more `MSG_`*xxx* flags (defined in `sys/socket.h`).

→ *toAddr*
Address to send to. See the `sendto()` manpage.

→ *toLen*
Size of *toAddr* buffer

→ *timeout*
Max timeout in system ticks. -1 means wait forever.

← *errRet*
This will contain an error code if return is -1.

**NOTE:** In Palm OS Cobalt the *flags* and *timeout* parameter values are ignored.

**Returns** Returns the number of bytes successfully sent, or -1 if an error occurred.

**Comments** This function mirrors the `sendto()` function and has similar arguments and semantics.

The function returns either the number of bytes successfully sent or -1. If -1, there was an error. In that case, an error code will be deposited at the address given by *errRet*. The other parameters are the same as for `sendto()` and are used when the data bytes are written to the network.

**See Also** `SslReceive()`, `sendto()`

### SslWrite Function

**Purpose** Sends data over the network.

**Declared In** `SslLib.h`

**Prototype**

```
int32_t SslWrite (SslContext *ctx,
    const void *buffer, int32_t bufferLen,
    status_t *errRet)
```

**Parameters**

← *ctx*
The SslContext to write to.

→ *buffer*
Buffer containing data to write.

→ *bufferLen*
Length, in bytes, of data to write.

← *errRet*
This will contain an error code if return is -1.

**Returns** Returns the number of bytes successfully sent, or -1 if an error occurred.

**Comments** Performs the same functionality as `SslSend()`. This call will use the timeout set earlier against the `SslContext`.

**See Also** `SslRead()`, `SslSend()`

## Application-Defined Functions

### SslCallbackFunc Function

**Purpose** A function that the application supplies to SslLib that will be called when specific situations occur during the SSL protocol. The callbacks are specific to the particular SslContext or SslLib they are registered with.

**Declared In** `SslLib.h`

**Prototype**

```
int32_t (*SslCallbackFunc) (SslCallback *cb,
   int32_t command, int32_t argi, void *argv)
```

**Parameters**

→ *cb*
: The `SslCallback` structure itself.

→ *command*
: A command which specifies the reason for the callback. A single callback structure can be used to handle several different types of SslLib callbacks. In this case, the function must have conditional logic to distinguish between the different commands. The command is used to interpret the remaining two parameters, *argi* and *argv*.

→ *argi*
: A command-specific 32-bit integer, normally used to specify more information about the reason for the callback.

↔ *argv*
: Pointer to a value that is normally determined by the *command* and/or the *argi* arguments.

**Returns** Returns `errNone` if the callback command was process without error, or a command-specific error code value otherwise.

**Comments** An application will supply an `SslCallback` structure to the SslLib library. When SslLib needs to then invoke the callback, the callback function is called with four arguments.

When an `SslCallback` is passed into SslLib, a copy is taken of the structure. This means that the structure passed in can be thought of as a template. It is important to remember that the data field will be copied, so if the object this element points to must be destroyed, additional logic will be required. When a SslContext is created, the `SslCallback` structures supplied to the SslLib are copied into the SslContext. This could cause problems if not handled correctly if the data pointed to by the data field is dynamic memory.

There are several general ‘commands’ that all callbacks should expect to receive; these commands are listed under “SSL Callback Commands” on page 342. There are normally related to creation and destruction of the structure that holds the callback. If the callback does need to perform any action due to these conditions, return 0.

**Example** If a callback returns a non-zero value, the SSL library will treat this as an error and return this value back out to the application. This can be used to implement abort functionality. While in the callback, any SslLib functions can be called to retrieve further information. If an `sslCmdInfoAlert` command is being processed, `SslContextGet_LastAlert()` can be called to retrieve the alert message that was received as shown here:

```
alert=SslContextGet_LastAlert(cb->ssl);
```

The alert values that can be returned are listed under "SSL Server Alerts" on page 350.

There are two defined callbacks currently used by SslLib: "Info" and "Verify."

### The Info Callback

For the Info callback, the *`command`* parameter is set to `sslCmdInfo`. The *`argi`*, and *`argv`* values passed are as follows:

| argi | argv type |
|---|---|
| `sslArgInfoHandshake` | `NULL` |
| `sslArgInfoAlert` | `NULL` |
| `sslArgInfoReadBefore` | `SslIoBuf` |
| `sslArgInfoReadAfter` | `SslIoBuf` |
| `sslArgInfoWriteBefore` | `SslIoBuf` |
| `sslArgInfoWriteAfter` | `SslIoBuf` |
| `sslArgCert` | `SslExtendedItems` – the certificate sent by the server. |

See "Info Callbacks" on page 345 for more information on these *`argi`* values.

This wealth of information makes it possible for the application to receive notification of state changes in the SSL protocol, receive any SSL protocol alert messages, and track the I/O operation that the SSL protocol is performing. This callback is primarily intended to aid in debugging applications or to provide visual feedback to the progress of the SSL protocol.

### The Verify Callback

For the Verify callback, the *`command`* parameter is set to `sslCmdVerify`. The *`argi`*, and *`argv`* values passed are as follows:

| argi | argv type |
|---|---|
| `CertMgrVerifyFailSignature` | `SslVerify` |
| `CertMgrVerifyFailUnknownIssuer` | `SslVerify` |
| `CertMgrVerifyFailNotAfter` | `SslVerify` |
| `CertMgrVerifyFailNotBefore` | `SslVerify` |
| `CertMgrVerifyFailBasicConstraints` | `SslVerify` |
| `CertMgrVerifyFailCriticalExtension` | `SslVerify` |
| `errNone` | `SslVerify` |

During the SSL handshake the server side sends a certificate to the client. This certificate contains the server's public key. SslLib attempts to verify that the certificate is valid. During this certificate verification process, if there are any errors, the Verify callback is called.

The application can, through this callback, override any of the error conditions reported during verification. If there is no Verify callback associated with an SslContext, any errors will immediately be returned to the application.

The Verify callback will be called as each certificate in the certificate chain is verified with any error values encountered passed in *`argi`* until the certificate is verified. If the certificate verifies ok, the 0 value is passed. This process is repeated for each certificate. This means that even if the certificate chain verifies without an error, the callback will be called once for each certificate (with a 0 *`argi`* value). If an Info callback is also registered, it would be called once after the certificate chain has been verified with the server's certificate. If there is no verification callback, and an error occurs, the application can 'clear' the error and re-call the relevant SslLib function. The verification will proceed from where it was up to.

#### *The SslVerify Structure*

The `SslVerify` structure is defined as follows:

```
typedef struct SslVerify_st {
   SslExtendedItems *certificate;
   SslExtendedItems *fieldItems;   /* Problem field base */
   UInt32 field;                   /* Problem field */
   SslExtendedItems *ex;           /* Extension */
   UInt32 depth;                   /* Certificate depth */
   UInt32 state;                   /* Verification state */
} SslVerify;
```

**NOTE:** The `SslVerify` structure is not declared in the Palm OS header files. In order to use it you'll have to declare it yourself.

`certificate` is a pointer to a structure containing the certificate currently being processed. `fieldItems` is a pointer to a structure that contains the data element that is currently causing a problem. `field` is the index into `fieldItems` of the erroneous data element. The `ex` field, if there is an error in extension processing, contains the data element that makes up the X509 extension that just failed. `depth` is the level of the certificate being processed, where 0 is the server's certificate, and higher numbers are certificates being used to chain to a trusted root certificate.

The following #defines represent the possible values of the `state` field. They indicate which section of the certificate verification failed.

```
#define sslVerifyFindParent 1
#define sslVerifySignature 2
#define sslVerifyNotBefore 3
#define sslVerifyNotAfterFindParent 4
#define sslVerifyExtensions 5
#define sslVerifyDone 6
```

> **NOTE:** The above #defines are not declared in the Palm OS header files. In order to use them you'll have to declare them yourself.

`fieldItems` will not always be the same pointer as `certificate`. This is especially true during extension errors. If we have an error in an extension, and the extension has been "decomposed," the "decomposed" elements will be in the `ex` field. The object identifier that identifies the extension "decomposed" in `ex` would be `verify->fieldItems.item[verify->field]`.

The following table lists the elements identified by the `fieldItems` and `field` values.

| | |
|---|---|
| `sslErrCertBadSignature` | The server's certificate, which contains the public key entries. |
| `SslErrCertNoTrustedRoot` | `NULL` |
| `SslErrCertNotAfter` | `sslExItemTypeX509`, `asn1FldX509NotAfter` |
| `SslErrCertNotBefore` | `sslExItemTypeX509`, `asn1FldX509NotBefore` |
| `sslErrCertConstraintViolation` | `asn1ExItemTypeX509Ex` start location |
| `sslErrCertUnknownCriticalExtension` | `asn1ExItemTypeX509Ex` start location |
| `SslErrOk` | `NULL` |

See the following section, "Extensions and Critical Extensions," for background on the cases of the returned field being the "`asn1ExItemTypeX509Ex` start location."

### *Extensions and Critical Extensions*

A certificate can have zero or more **extensions**. These extensions specify extra information to be used during evaluation of a certificate. Each extension consists of an "Object identifier," an optional Boolean "critical extension" flag, and the data bytes. The `fieldItems->item[field]` values in this case points to the `SslExtendedItem` that contains the “Object identifier” for that extension. `field+1` will reference either the optional Boolean field that flags the extension as critical or the data bytes.

```
SslVerify *verify;
SslExtendedItem *oid,*critical,*data;

oid= &(verify->fieldItems[verify->field]);
if (verify->fieldItems[verify->field+1].data_type ==
asn1Boolean)
   {
   critical=(verify->fieldItems[verify->field+1]);
   data=(verify->fieldItems[verify->field+2]);
   }
else
   {
   critical=NULL;
   data= &(verify->fieldItems[verify->field+1]);
   }
/* The data bytes for the ‘data’ is located at
 * ((Uint8 *)verify->fieldItems)+data->offset;
```

SslLib attempts to interpret only critical extensions, so the critical field should always be present. If a critical extension is not understood, the certificate should be rejected. These callback values allow the application to accept a certificate with critical extensions that the application SslLib does not understand.

SslLib recognizes three extensions at this point in time, taken from the X.509 standard:

| | |
|---|---|
| 2.5.29.15 | `KeyUsage` |
| 2.5.29.37 | `ExtKeyUsage` |
| 2.5.29.19 | `BasicConstraints` |

If any of these constraints are flagged as critical, an error will not occur (assuming they are valid).

**IMPORTANT:** In Palm OS Cobalt version 6.1 and earlier, the SSL library does not process the `BasicConstraints` or `KeyUsage` extensions. If the SSL library finds a critical extension of any type, `CertMgrVerifyFailCriticalExtension` is returned to the application. For more information, see "Critical Extensions" on page 58.

`BasicConstraints` is the only extension currently verified. It specifies if a certificate can be used for signing other certificates. If the certificate is being used incorrectly, an `sslErrCertConstraintViolation` error will be generated. For this error, the `ex` field of the `SslVerify` structure will potentially contain

```
asn1ExItemTypeX509ExData, asn1FldX509ExBasicConstraintsCa
asn1ExItemTypeX509ExData, asn1FldX509ExBasicConstraintsPathLenConstraint
```

Note that if these optional fields are not in the certificate, they will not be present in the `SslExtendedItems`. The `PathLenConstraint` will also not contain any data bytes; rather,. the numeric value this field contains will be encoded in the `len` field of the `SslExtendedItem`. If this was not the case, the application would have to learn all about decoding ASN.1 integers. The depth field relates to this certificate.

If this error occurs, the application should not override the error, due to its serious nature.

### *The SslExtendedItems Structure*

The `SslExtendedItems` structure is defined as follows:

```
typedef struct SslExtendedItems_st {
   UInt32 length;
   UInt32 num;
   SslExtendedItem eitem[1];
} SslExtendedItems;
```

**NOTE:** The `SslExtendedItems` structure is not declared in the Palm OS header files. In order to use it you'll have to declare it yourself.

The `SslExtendedItems` structure is used to hold a set of `SslExtendedItem` structures. The `eitem` field, while defined as a size of one, is actually large enough to hold `num` entries. The `length` field is the total size of the structure. The structure can be copied by allocating `length` bytes and then copying `length` bytes from the `SslExtendedItems` pointer into the new location. An `SslExtendedItems` structure is used to hold sets of related data elements. A set of such values may contain a RSA public key, a certificate, and a certificate extension all in the same `SslExtendedItems` structure.

### *The SslExtendedItem Structure*

Each `SslExtendedItem` belongs to a type that is predefined for each of these objects.

| | |
|---|---|
| `sslExItemTypeX509` | X.509 Certificate. |
| `SslExItemTypeRSA` | RSA public key. |
| `sslExItemTypeRDN` | An X.509 Relative Distinguished Name (RDN). This is a complex way of saying a certificate name. Each certificate contains two names, the Subject of the certificate and the Issuer of the certificate. Both are encoded as RDNs which contain multiple fields. |
| `sslExItemTypeX509Ex` | X.509 certificates can contain what are called Extensions. A certificate can contain multiple extensions. This type is used to specify extensions. |
| `SslExItemTypeE509ExData` | Used to group 'decomposed' X.509 extensions. |

The `SslExtendedItem` structure is defined as follows:

```
typedef struct SslExtendedItem_st {
   UInt16 type;
   UInt16 field;
   UInt16 dataType;
   UInt16 len;
   UInt32 offset;
} SslExtendedItem;
```

**NOTE:** The `SslExtendedItem` structure is not declared in the Palm OS header files. In order to use it you'll have to declare it yourself.

An `SslExtendedItem` structure is a single data element. The `type` field values (as specified in the preceding table) are used to group related items. A single `SslExtendedItems` structure can contain multiple `SslExtendedItem` structures with different `type` values. In this way a single structure can contain elements referring to both a certificate and an RSA public key.

The `field` field contains a type-specific value that is used to identify the `SslExtendedItem`. The values for this field are defined specifically for each type.

The `dataType` field specifies the encoding type of the data. For the cases being used by SslLib, the value is the ASN.1 encoding type. These values are defined in the `SslLibAsn1.h` header file. They are relevant primarily if the application is attempting to display the data bytes.

The `len` field is the length of the data in the `SslExtendedItem`

The `offset` field is the offset from the start of the parent `SslExtendedItems` to the data field.

To access all the data bytes in an `SslExtendedItems` structure:

```
SslExtendedItems *ei;
UInt16 i,j;
UInt8 *p;

for (i=0; i<ei->num; i++) {
   p=((Uint8 *)ei)+ei->eitem[i].offset;

   for (j=0; j<ei->eitem[i].len; j++)
```

```
      doSomething(p[j]);
}
```

An `SslExtendedItems` structure will often contain multiple types. When SslLib returns an `SslExtendedItems` structure for a certificate, it will usually contain the types `sslExItemTypeX509`, `sslExItemTypeRSA`, and `sslExItemTypeRDN` (for the subject name).

**Example** Following is an example of usage from the SslLib, for the ‘information’ callback:

```
Int32 info_callback(SslCallback *cb,Int32 command,
   Int32 argi,void *argv);

SslCallback infoCB;

infoCB.callback=info_callback;
SslContextSet_InfoCallback(ssl,&infoCB);
SslContextSet_InfoInteresrt(ssl,
   sslFlgInfoAlert|
   sslFlgInfoHandshake|
   sslFlgInfoIo);

/* We have now configured the SslContext so that
 * info_callback will be called when ‘interesting’ events
 * occur. */

Int32 info_callback(SslCallback *cb,Int32 command,
   Int32 argi, void *argv)
   {
   UInt32 alert;

   switch (command){
   case sslCmdInfo:
      /* We have received an ‘info’ call */
      switch (argi){
      case sslArgInfoHandshake:
         /* The SslContext is in the handshake stage
            of connection establishment. */
         break;
      case sslArgInfoAlert:
         /* An Alert message was received by the SslContext*/
         alert=SslContextGet_LastAlert(cb->ssl);
         break;
      case sslArgInfoReadBefore:
```

```
      case sslArgInfoReadAfter:
      case sslArgInfoWriteBefore:
      case sslArgInfoWriteAfter:
         /* The SslContext is doing network operations */
         break;
      }
      break;
   case sslCmdNew:   /* Called when we are 'copied in' */
   case sslCmdFree:   /* Called when we are 'finished' */
   case sslCmdReset:   /* Called instead of 'sslCmdNew' to
      reset the settings */
   case sslCmdSet:   /* Set a value */
   case sslCmdGet:   /* Get a value */
      break;
   }
   return(0);
}
```

# 17
# SSL Library Macros

This chapter provides reference documentation for the macros that your application uses to set and get `SslLib` and `SslContext` attribute values. As well, the constants representing those attributes are listed in this chapter.

This chapter is divided into the following sections:



The header file `SslLibMac.h` declares the API that this chapter describes.

Documentation for the functions that these macros employ can be found in Chapter 16, "SSL Library," on page 331. A detailed description of each attribute can be found under "Attributes" on page 59.

## SSL Library Macro Constants

### Attribute Values

**Purpose** These constants represent the attributes that you can set or get by calling `SslContextGet...()`, `SslContextSet...()`, `SslLibGet...()`, or `SslLibSet...()`. The macros documented in "SSL Library Macros" on page 388 call the appropriate function and supply the proper attribute value. Accordingly, applications should use these macros instead of calling the set or get functions directly.

Complete documentation for the attributes that these values represent can be found in Chapter 2, "SSL Concepts," on page 55.

**Declared In** `SslLibMac.h`

**Constants** `#define sslAttrAppInt32 0x0F0A0A13`

```
#define sslAttrAppPtr 0x0F090911
#define sslAttrAutoFlush 0x0F070712
#define sslAttrBufferedReuse 0x0F080812
#define sslAttrCertPeerCert 0x0100FF01
#define sslAttrCertPeerCertInfoType 0x0102FF01
#define sslAttrCertPeerCommonName 0x0180FF01
#define sslAttrCertSslVerify 0x0101FF04
#define sslAttrCertVerifyChain 0x0103FF01
#define sslAttrClientCertRequest 0x0F20FF12
#define sslAttrCompat 0x0F010113
#define sslAttrCspCipherSuite 0x0001FF01
#define sslAttrCspCipherSuiteInfo 0x0082FF01
#define sslAttrCspCipherSuites 0x00008101
#define sslAttrCspSslSession 0x00808001
#define sslAttrDelayReadServerFinished 0x0F232312
#define sslAttrDontSendShutdown 0x0F212112
#define sslAttrDontWaitForShutdown 0x0F222212
#define sslAttrError 0x0F111113
#define sslAttrErrorState 0x0F05FF14
#define sslAttrHelloVersion 0x0F242412
#define sslAttrHsState 0x0F12FF13
#define sslAttrInfoCallback 0x0F0E0E15
#define sslAttrInfoInterest 0x0F020213
#define sslAttrIoFlags 0x04030303
#define sslAttrIoSocket 0x04010103
#define sslAttrIoStruct 0x04008001
#define sslAttrIoTimeout 0x04020203
#define sslAttrLastAlert 0x0F131313
```

```
#define sslAttrLastApi 0x0F1FFF12
#define sslAttrLastIo 0x0F1EFF12
#define sslAttrLibAppInt32 0x0F0F0F03
#define sslAttrLibAppPtr 0x0F0E0E01
#define sslAttrLibAutoFlush 0x0F0C0C02
#define sslAttrLibBufferedReuse 0x0F0D0D02
#define sslAttrLibCompat 0x0F010103
#define sslAttrLibDelayReadServerFinished
  0x0F141402
#define sslAttrLibDontSendShutdown 0x0F121202
#define sslAttrLibDontWaitForShutdown 0x0F131302
#define sslAttrLibHelloVersion 0x0F151503
#define sslAttrLibInfoCallback 0x0F080805
#define sslAttrLibInfoInterest 0x0F020203
#define sslAttrLibMode 0x0F040403
#define sslAttrLibProtocolSupport 0x0F161603
#define sslAttrLibProtocolVersion 0x0F030303
#define sslAttrLibRbufSize 0x0F101003
#define sslAttrLibReadStreaming 0x0F0B0B02
#define sslAttrLibVerifyCallback 0x0F0A0A05
#define sslAttrLibWbufSize 0x0F111103
#define sslAttrMode 0x0F048013
#define sslAttrProtocolSupport 0x0F252512
#define sslAttrProtocolVersion 0x0F030313
#define sslAttrRbufSize 0x0F1B8113
#define sslAttrReadBufPending 0x0F16FF13
#define sslAttrReadOutstanding 0x0F18FF13
#define sslAttrReadRecPending 0x0F17FF13
#define sslAttrReadStreaming 0x0F060612
```

```
#define sslAttrSessionReused 0x0F14FF12
#define sslAttrStreaming 0x0F1DFF12
#define sslAttrVerifyCallback 0x0F101015
#define sslAttrWbufSize 0x0F1C8213
#define sslAttrWriteBufPending 0x0F15FF13
```

# SSL Library Macros

## SslContextGet_AppInt32 Macro

**Purpose** Obtain the value of the AppInt32 attribute, an arbitrary 32-bit value that an application can attach to an SslContext.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_AppInt32 (`*`lib`*`)`

**Parameters** → *lib*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns the 32-bit value.

**Comments** `SslContextDestroy()` does not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

**See Also** "AppInt32" on page 67, `SslContextSet_AppInt32()`, `SslLibGet_AppInt32()`

## SslContextGet_AppPtr Macro

**Purpose** Obtain the value of the AppPtr attribute, an arbitrary pointer value that an application can attach to an SslContext.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_AppPtr (`*`lib`*`, `*`v`*`)`

**Parameters** → *lib*
The `SslContext` from which the value is to be retrieved.

→ *v*
The address of a pointer variable into which the `AppPtr` attribute value is written.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** `SslContextDestroy()` does not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

**See Also** "AppPtr" on page 67, `SslContextSet_AppPtr()`, `SslLibGet_AppPtr()`

## SslContextGet_AutoFlush Macro

**Purpose** Determine whether `SslSend()` and `SslWrite()` attempt to immediately send the supplied data bytes to the network.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_AutoFlush (`*`ssl`*`)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns 0 if data is buffered, or 1 if the data is sent immediately.

**Comments** It is very important to remember to use `SslFlush()` when AutoFlush is disabled.

**See Also** "AutoFlush" on page 61, `SslContextSet_AutoFlush()`, `SslLibGet_AutoFlush()`

## SslContextGet_BufferedReuse Macro

**Purpose** Determine if the last message in an SslSession-reused handshake should be buffered instead of being sent over the network.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_BufferedReuse (`*`ssl`*`)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns 0 if the last message is *not* buffered, or a non-zero value if it is.

**See Also** "BufferedReuse" on page 75, `SslContextSet_BufferedReuse()`, `SslLibGet_BufferedReuse()`

## SslContextGet_CertChain Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_CertChain (`*`ssl`*`, `*`v`*`)`

**Parameters**

→ *`ssl`*
The `SslContext` from which the value is to be retrieved.

→ *`v`*

**Returns**

## SslContextGet_CipherSuite Macro

**Purpose** Identify the cipher suite being used by the current connection.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_CipherSuite (`*`ssl`*`, `*`v`*`)`

**Parameters**

→ *`ssl`*
The `SslContext` from which the value is to be retrieved.

→ *`v`*
Pointer to a pointer variable; upon return the pointer variable points to two bytes which identify the cipher suite being used by the current connection. If these two bytes are both set to zero, no cipher suite is being used. Otherwise, see "Cipher Suites" on page 344 for the list of possible cipher suites.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "CipherSuite" on page 68

## SslContextGet_CipherSuiteInfo Macro

**Purpose** Obtain information relevant to the current cipher suite. This macro populates a `SslCipherSuiteInfo` structure that must have been allocated by the caller.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_CipherSuiteInfo (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` from which the value is to be retrieved.

→ *v*
Pointer to the `SslCipherSuiteInfo` structure that is to be filled in.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "CipherSuiteInfo" on page 68, `SslContextGet_CipherSuite()`

## SslContextGet_CipherSuites Macro

**Purpose** Obtain the list of cipher suites that the SSL protocol is attempting to use.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_CipherSuites (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` from which the value is to be retrieved.

→ *v*
Supply the address of a pointer variable; upon return the pointer variable will point to a series of bytes, where the first two bytes indicate the number of bytes that follow, and each pair of bytes after that is one of the values listed under "Cipher Suites" on page 344. See "CipherSuites" on page 61 for more details on this attribute.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "CipherSuites" on page 61, `SslContextSet_CipherSuites()`, `SslLibGet_CipherSuites()`

## SslContextGet_ClientCertRequest Macro

**Purpose** Determine whether or not the SSL server requested a client certificate.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_ClientCertRequest (`*`ssl`*`)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns 0 if the server did not request a client certificate, or 1 if the server did request one.

**See Also** "ClientCertRequest" on page 68

## SslContextGet_Compat Macro

**Purpose** Determine which SSL protocol compatibility flags are set. These flags enable compatibility with certain incorrect SSL protocol implementations.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_Compat (`*`ssl`*`)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns a 32-bit integer value that is the logical OR of those compatibility flags that have been set. See "Compatibility Flags" on page 342 for the defined constants that correspond to the compatibility flags.

**See Also** "Compat" on page 69, `SslContextSet_Compat()`, `SslLibGet_Compat()`

## SslContextGet_DelayReadServerFinished Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype**

```
#define SslContextGet_DelayReadServerFinished
    (ssl)
```

**Parameters**

→ *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns**

## SslContextGet_DontSendShutdown Macro

**Purpose** Determine whether or not an `SslClose()` will send a shutdown message to the server.

**Declared In** `SslLibMac.h`

**Prototype**

```
#define SslContextGet_DontSendShutdown (ssl)
```

**Parameters**

→ *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns zero if `SslClose()` does send a shutdown message to the server, or a non-zero value if it doesn't.

**See Also** "DontSendShutdown" on page 76, `SslContextSet_DontSendShutdown()`, `SslLibGet_DontSendShutdown()`, `SslContextGet_DontWaitForShutdown()`

## SslContextGet_DontWaitForShutdown Macro

**Purpose** Determine whether or not the `SslContext` will wait for a shutdown message in `SslClose()`.

**Declared In** `SslLibMac.h`

**Prototype**

```
#define SslContextGet_DontWaitForShutdown (ssl)
```

**Parameters**

→ *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns zero if the `SslContext` waits for a shutdown message, or a non-zero value if it doesn't.

**See Also** "DontWaitForShutdown" on page 76, `SslLibGet_DontWaitForShutdown()`, `SslContextGet_DontSendShutdown()`, `SslContextSet_DontWaitForShutdown()`

## SslContextGet_Error Macro

**Purpose** Obtain the error value produced when a fatal error occurs while using an `SslContext`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_Error (`*`ssl`*`)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns 0 if there was no error, or one of the error codes listed under "SSL Library Errors" on page 351 if an error did occur.

**Comments** Once the error attribute is set, the SslLib network APIs will continue to return this error (unless the error is a non-fatal error) until either an SSL Reset is performed on the `SslContext` or the error is cleared.

**See Also** "Error" on page 62, `SslContextSet_Error()`

## SslContextGet_HelloVersion Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_HelloVersion (`*`ssl`*`)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

**Returns**

## SslContextGet_HsState Macro

**Purpose** Determine the state that the SSL protocol is currently in.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_HsState (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** One of the values are defined under "SSL Protocol States" on page 349 to indicate the SSL protocol handshake.

**See Also** "HsState" on page 69

## SslContextGet_InfoCallback Macro

**Purpose** Obtain a pointer to the callback function called when various situations occur during the usage of an `SslContext`. This callback is primarily intended for debugging and feedback purposes.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_InfoCallback (ssl, v)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

→ *v*
Pass the address of a pointer variable; upon return this variable will point to the callback function. The callback function is of type `SslCallbackFunc()`.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** See "InfoCallback" on page 69 and `SslCallbackFunc()` (documented on page 373) for more on how this callback function is used.

**See Also** "InfoCallback" on page 69, `SslContextSet_InfoCallback()`, `SslLibGet_InfoCallback()`

## SslContextGet_InfoInterest Macro

**Purpose** Obtain the flags that specify the events for which the Info callback will be called.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_InfoInterest (`*`ssl`*`)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

**Returns** The logical OR of the `sslFlgInfo`*xxx* values listed under "InfoInterest Values" on page 347.

**See Also** "InfoInterest" on page 70, `SslContextSet_InfoInterest()`, `SslLibGet_InfoInterest()`

## SslContextGet_IoFlags Macro

**Purpose** Obtain the flags value that is passed to `sys/socket` calls.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_IoFlags (`*`ssl`*`)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

**Returns** The 32-bit flags value that is passed to `sys/socket` calls.

**Comments** Since we will normally be using TCP connections with SSL, this attribute is more included for completeness rather than utility. Read about this flags value in the `sendto()` and `recvfrom()` man page.

**NOTE:** The `MSG_OOB` and `MSG_PEEK` values are not valid and will be silently removed.

**See Also** "IoFlags" on page 70, `SslContextSet_IoFlags()`

## SslContextGet_IoStruct Macro

**Purpose** Obtain the `SslContext`'s internal `SslSocket` structure.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_IoStruct (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` from which the value is to be retrieved.

→ *v*
Pass the address of a pointer variable. Upon return the variable will point to the `SslContext`'s `SslSocket` structure.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "IoStruct" on page 70, `SslContextSet_IoStruct()`

## SslContextGet_IoTimeout Macro

**Purpose** Obtain the `SslContext`'s internal timeout value.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_IoTimeout (ssl)`

**Parameters**

→ *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** The internal timeout value, in seconds.

**Comments** When a call is made into the SslLib API which does not specify a timeout, this internal value is used. If the API call has a timeout value, it overrides this internal value.

By default, the `SslContext`'s internal timeout value is 10 seconds.

**See Also** "IoTimeout" on page 71, `SslContextSet_IoTimeout()`

## SslContextGet_LastAlert Macro

**Purpose** Obtain the last alert value received from the server.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_LastAlert (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** One of the values listed under "SSL Server Alerts" on page 350.

**Comments** Non-fatal alerts have a value of the form `0x01`*XX*, while fatal alerts have the form `0x02`*XX*. SslLib will fail on fatal alerts and continue on non-fatal alerts.

**See Also** "LastAlert" on page 71, `SslContextSet_LastAlert()`

## SslContextGet_LastApi Macro

**Purpose** Identify the last SslLib API call that was made.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_LastApi (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns one of the values listed under "LastApi Attribute Values" on page 347.

**See Also** "LastApi" on page 71

## SslContextGet_LastIo Macro

**Purpose** Identify the last network operation performed.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_LastIo (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns one of the values listed under “LastIO Attribute Values” on page 348.

**See Also** “LastIo” on page 71

## SslContextGet_Mode Macro

**Purpose** Obtain the value of the Mode attribute, which controls whether the SSL protocol is on or off.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_Mode (ssl)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns one of the values listed under “Mode Attribute Values” on page 339.

**See Also** “Mode” on page 63, `SslContextSet_Mode()`, `SslLibGet_Mode()`

## SslContextGet_PeerCert Macro

**Purpose** Obtain the certificate supplied by the other end of the SSL connection, if one is available.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_PeerCert (ssl, v)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

→ *`v`*
Pass the address of a pointer variable. Upon return that variable will reference an `SslExtendedItems` data structure internal to the `SslContext` that contains the peer certificate.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under “SSL Library Errors” on page 351.

**Comments** The returned pointer references an `SslExtendedItems` data structure which is internal to the `SslContext` and will be disposed of by the `SslContext`. If a new connection is established with the `SslContext`, previously returned `PeerCert` pointers will become invalid. If the application wishes to preserve the certificate for an extended period, it should make a local copy.

**See Also** "PeerCert" on page 72, "The SslExtendedItems Structure" on page 380, `SslContextGet_PeerCertInfoType()`, `SslContextGet_PeerCommonName()`

## SslContextGet_PeerCertInfoType Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_PeerCertInfoType (ssl, v)`

**Parameters**

→ *ssl*
: The `SslContext` from which the value is to be retrieved.

→ *v*

**Returns**

## SslContextGet_PeerCommonName Macro

**Purpose** Obtain the server certificate's common name.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_PeerCommonName (ssl, v)`

**Parameters**

→ *ssl*
: The `SslContext` from which the value is to be retrieved.

→ *v*
: Supply the address of a pointer variable. Upon return the variable will point to an `SslExtendedItems` structure containing the common name for the server's certificate (or, if there is no common name available, the variable will be set to `NULL`.)

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Example** The following code shows how to access the common name from within the `SslExtendedItem` structure:

```
SslExtendedItems *cert;
SslExtendedItem *commonName;
uint16_t length;
uint8_t *bytes;

SslContextGet_PeerCert(ssl, &cert);
if (cert == NULL) goto err;
SslContextGet_PeerCommonName(ssl,&commonName);
length=commonName->len;
bytes=((Int8 *)cert)+commonName->offset;
// bytes now points to the common name, and length contains
// the length of the common name string.
```

**See Also** "PeerCommonName" on page 72, "The SslExtendedItems Structure" on page 380

## SslContextGet_ProtocolSupport Macro

**Purpose** Determine which variants of the SSL protocol are being used.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_ProtocolSupport (`*`ssl`*`)`

**Parameters** → *`ssl`*
The `SslContext` from which the value is to be retrieved.

**Returns** One or more of the values listed under "Protocol Variants" on page 340, OR'd together.

**See Also** `SslContextGet_ProtocolVersion()`, `SslContextSet_ProtocolSupport()`, `SslLibGet_ProtocolSupport()`

## SslContextGet_ProtocolVersion Macro

**Purpose** Determine which version of the SSL protocol is being used.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_ProtocolVersion (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** One of the values listed under "Protocol Versions" on page 340.

**See Also** "ProtocolVersion" on page 73, `SslContextSet_ProtocolVersion()`, `SslLibGet_ProtocolVersion()`

## SslContextGet_RbufSize Macro

**Purpose** Obtain the size, in bytes, of the `SslContext`'s read buffer.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_RbufSize (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** The read buffer size. This is a value that ranges from 0 to 16384.

**See Also** "RbufSize" on page 64, `SslContextSet_RbufSize()`, `SslLibGet_RbufSize()`

## SslContextGet_ReadBufPending Macro

**Purpose** Obtains the number of data bytes that are currently buffered for reading from the `SslContext`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_ReadBufPending (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** The number of buffered read bytes.

**Comments** The returned number of bytes also includes bytes used for encoding SSL records.

**See Also** "ReadBufPending" on page 77

## SslContextGet_ReadOutstanding Macro

**Purpose** Obtain the number of bytes in the current record that have not been read from the network

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_ReadOutstanding (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** The number of bytes outstanding in current SSL record.

**See Also** "ReadOutstanding" on page 77

## SslContextGet_ReadRecPending Macro

**Purpose** Obtain the number of application data bytes that are buffered, awaiting the application to read.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_ReadRecPending (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** The number of buffered SSL record bytes.

**See Also** "ReadRecPending" on page 77,
`SslContextGet_ReadOutstanding()`

## SslContextGet_ReadStreaming Macro

**Purpose** Determine whether data can be returned to the application from the SSL connection before the full record has been downloaded.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_ReadStreaming (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns a zero value if data is only returned when the full record has been downloaded, or a non-zero value otherwise.

**See Also** "ReadStreaming" on page 77, `SslContextSet_ReadStreaming()`, `SslLibGet_ReadStreaming()`

## SslContextGet_SessionReused Macro

**Purpose** Determine whether the SSL handshake was able to perform a truncated handshake by re-using the SSL session values in the `SslContext`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_SessionReused (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** Returns a non-zero value if the SSL handshake was able to perform a truncated handshake by re-using the SSL session values in the `SslContext`, or zero if it wasn't.

**See Also** "SessionReused" on page 74

## SslContextGet_Socket Macro

**Purpose** Obtain the socket that the `SslContext` should use to perform its network I/O operations.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_Socket (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** The descriptor for the `SslContext`'s socket.

**See Also** "Socket" on page 65, `SslContextSet_Socket()`

## SslContextGet_SslSession Macro

**Purpose** Get the `SslContext`'s `SslSession` structure. This is either the `SslSession` currently being used, or the `SslSession` for this `SslContext` to use to establish its next connection.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_SslSession (ssl, v)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

→ *v*
Pass the address of a pointer variable; upon return this variable will point to the `SslContext`'s `SslSession` structure.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "SslSession" on page 74, `SslContextSet_SslSession()`

## SslContextGet_SslVerify Macro

**Purpose** Obtain a pointer to the structure containing the verification state. This structure can be helpful when attempting to resolve any problems that SslLib may have encountered during certificate verification.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_SslVerify (ssl, v)`

**Parameters** → *ssl*

The `SslContext` from which the value is to be retrieved.

→ *v*

Supply the address of a pointer variable. Upon return the variable will point to an `SslVerify` structure containing the preserved state. See "The SslVerify Structure" on page 377 for a description of this structure.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "SslVerify" on page 74

## SslContextGet_Streaming Macro

**Purpose** Determine if the current `SslContext` is doing read-streaming.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_Streaming (ssl)`

**Parameters** → *ssl*

The `SslContext` from which the value is to be retrieved.

**Returns** Returns 1 if the current `SslContext` is doing read-streaming.

**See Also** "Streaming" on page 78

## SslContextGet_VerifyCallback Macro

**Purpose** Obtain a pointer to the callback function used to assist with certificate verification.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_VerifyCallback (ssl, v)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

→ *v*
Pass the address of a pointer variable; upon return this variable will point to the callback function. The callback function is of type `SslCallbackFunc()`.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** See "VerifyCallback" on page 65 and `SslCallbackFunc()` (documented on page 373) for more on how this callback function is used to verify a certificate.

**See Also** "VerifyCallback" on page 65, `SslContextSet_VerifyCallback()`, `SslLibGet_VerifyCallback()`

## SslContextGet_WbufSize Macro

**Purpose** Obtain the size, in bytes, of the `SslContext`'s write buffer.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_WbufSize (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** The write buffer size. This is a value that ranges from 0 to 16384.

**See Also** "WbufSize" on page 66, `SslContextSet_WbufSize()`, `SslLibGet_WbufSize()`

## SslContextGet_WriteBufPending Macro

**Purpose** Obtains the number of bytes in the `SslContext`'s write buffer waiting to be sent to the remote machine.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextGet_WriteBufPending (ssl)`

**Parameters** → *ssl*
The `SslContext` from which the value is to be retrieved.

**Returns** The number of buffered bytes waiting to be sent.

**Comments** This value will normally be zero unless **AutoFlush** is disabled and/or non-blocking I/O is being used.

**See Also** "WriteBufPending" on page 78, `SslContextGet_AutoFlush()`

## SslContextSet_AppInt32 Macro

**Purpose** Set the value of the `AppInt32` attribute, an arbitrary 32-bit value that an application can attach to an `SslContext`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_AppInt32 (ssl, v)`

**Parameters** → *ssl*
The `SslContext` on which to operate.

→ *v*
The 32-bit value.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** `SslContextDestroy()` does not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

**See Also** "AppInt32" on page 67, `SslContextGet_AppInt32()`, `SslLibSet_AppInt32()`

## SslContextSet_AppPtr Macro

**Purpose** Set the value of the AppPtr attribute, an arbitrary pointer value that an application can attach to an SslContext.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_AppPtr (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
The `AppPtr` attribute value.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** `SslContextDestroy()` does not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

**See Also** "AppPtr" on page 67, `SslContextGet_AppPtr()`, `SslLibSet_AppPtr()`

## SslContextSet_AutoFlush Macro

**Purpose** Specify whether `SslSend()` and `SslWrite()` should attempt to immediately send the supplied data bytes to the network.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_AutoFlush (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
Pass 0 to have the data buffered, or 1 to have the data sent immediately.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** It is very important to remember to use `SslFlush()` when AutoFlush is disabled.

**See Also** "AutoFlush" on page 61, `SslContextGet_AutoFlush()`, `SslLibSet_AutoFlush()`

## SslContextSet_BufferedReuse Macro

**Purpose** Specify whether the last message in an SslSession-reused handshake should be buffered instead of being sent over the network.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_BufferedReuse (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
Supply a value of 0 if the last message should *not* be buffered, or a non-zero value if it should.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "BufferedReuse" on page 75, `SslContextGet_BufferedReuse()`, `SslLibSet_BufferedReuse()`

## SslContextSet_CipherSuites Macro

**Purpose** Specify the list of cipher suites that the SSL protocol should attempt to use.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_CipherSuites (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
Pointer to an array of byte pairs. The first two bytes contain the number of bytes that follow, and each successive pair is one of the values listed under "Cipher Suites" on page 344.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "CipherSuites" on page 61, `SslContextGet_CipherSuites()`, `SslLibSet_CipherSuites()`

## SslContextSet_Compat Macro

**Purpose** Determine how compatible the SSL protocol should be with certain incorrect SSL protocol implementations.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_Compat (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
The logical OR of those compatibility flags that correspond to the SSL protocol incompatibilities that should be accommodated. See "Compatibility Flags" on page 342 for the defined constants that correspond to the compatibility flags.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "Compat" on page 69, `SslContextGet_Compat()`, `SslLibSet_Compat()`

## SslContextSet_DelayReadServerFinished Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype**
```
#define SslContextSet_DelayReadServerFinished
    (ssl, v)
```

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

## SslContextSet_DontSendShutdown Macro

**Purpose** Specify whether or not an `SslClose()` should send a shutdown message to the server.

**Declared In** `SslLibMac.h`

**Prototype**
```
#define SslContextSet_DontSendShutdown (ssl, v)
```

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
Zero if `SslClose()` should send a shutdown message, or a non-zero value if it shouldn't.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "DontSendShutdown" on page 76, `SslContextGet_DontSendShutdown()`, `SslLibSet_DontSendShutdown()`, `SslContextSet_DontWaitForShutdown()`

## SslContextSet_DontWaitForShutdown Macro

**Purpose** Specify whether or not the `SslContext` should wait for a shutdown message in `SslClose()`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_DontSendShutdown (ssl, v)`

**Parameters**

→ *ssl*
  The `SslContext` on which to operate.

→ *v*
  Zero if the `SslContext` waits for a shutdown message, or a non-zero value if it doesn't.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "DontWaitForShutdown" on page 76, `SslContextGet_DontWaitForShutdown()`, `SslContextSet_DontSendShutdown()`, `SslLibSet_DontWaitForShutdown()`

## SslContextSet_Error Macro

**Purpose** Associate a new error value with an `SslContext`. This value overrides any error value produced when a fatal error occurs while using an `SslContext`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_Error (ssl, v)`

**Parameters**

→ *ssl*
  The `SslContext` on which to operate.

→ *v*
  The new error value. A value of 0 corresponds to "no error."

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** Once the error attribute is set, the SslLib network APIs will continue to return this error (unless the error is a non-fatal error) until either an SSL Reset is performed on the `SslContext` or the error is cleared.

**See Also** "Error" on page 62, `SslContextGet_Error()`

## SslContextSet_HelloVersion Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_HelloVersion (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

## SslContextSet_InfoCallback Macro

**Purpose** Specify the callback function called when various situations occur during the usage of an `SslContext`. This callback is primarily intended for debugging and feedback purposes.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_InfoCallback (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
A pointer to a callback function of type `SslCallbackFunc()`.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** See "InfoCallback" on page 69 and `SslCallbackFunc()` (documented on page 373) for more on how this callback function is used.

**See Also** "InfoCallback" on page 69, `SslContextGet_InfoCallback()`, `SslLibSet_InfoCallback()`

## SslContextSet_InfoInterest Macro

**Purpose** Specify the events for which the Info callback will be called.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_InfoInterest (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
The logical OR of the `sslFlgInfo`*xxx* values listed under "InfoInterest Values" on page 347.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "InfoInterest" on page 70, `SslContextGet_InfoInterest()`, `SslLibSet_InfoInterest()`

## SslContextSet_IoFlags Macro

**Purpose** Specify the flags value that is to be passed to `sys/socket` calls.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_IoFlags (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
The 32-bit flags value to be passed to `sys/socket` calls.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** Since we will normally be using TCP connections with SSL, this attribute is more included for completeness rather than utility. Read about this flags value in the `sendto()` and `recvfrom()` man page.

> **NOTE:** The MSG_OOB and MSG_PEEK values are not valid and will be silently removed.

**See Also** "IoFlags" on page 70, `SslContextGet_IoFlags()`

## SslContextSet_IoStruct Macro

**Purpose** Specify the `SslSocket` structure to be used by an `SslContext`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_IoStruct (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
Pointer to the `SslSocket` structure.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "IoStruct" on page 70, `SslContextSet_IoStruct()`

## SslContextSet_IoTimeout Macro

**Purpose** Specify the `SslContext`'s internal timeout value.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_IoTimeout (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
The new internal timeout value, in seconds.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** When a call is made into the SslLib API which does not specify a timeout, this internal value is used. If the API call has a timeout value, it overrides this internal value.

By default, the `SslContext`'s internal timeout value is 10 seconds.

**See Also** "IoTimeout" on page 71, `SslContextGet_IoTimeout()`

## SslContextSet_LastAlert Macro

**Purpose** Override the last alert value received from the server.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_LastAlert (`*`ssl`*`, `*`v`*`)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
The new "last alert" value. This should be one of the values listed under "SSL Server Alerts" on page 350.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** Non-fatal alerts have a value of the form `0x01`*XX*, while fatal alerts have the form `0x02`*XX*. SslLib will fail on fatal alerts and continue on non-fatal alerts.

**See Also** "LastAlert" on page 71, `SslContextGet_LastAlert()`

## SslContextSet_Mode Macro

**Purpose** Specify the value of the Mode attribute, which controls whether the SSL protocol is on or off.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_Mode (ssl, v)`

**Parameters** → *ssl*
The `SslContext` on which to operate.

→ *v*
One of the values listed under "Mode Attribute Values" on page 339.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "Mode" on page 63, `SslContextGet_Mode()`, `SslLibSet_Mode()`

## SslContextSet_ProtocolSupport Macro

**Purpose** Specify the SSL protocol variants to use.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_ProtocolSupport (ssl, v)`

**Parameters** → *ssl*
The `SslContext` on which to operate.

→ *v*
One or more of the values listed under "Protocol Variants" on page 340, OR'd together.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** Use `sslSupport_SSLv3`, `sslSupport_TLSv1`, or `sslSupport_Both` to enable SSLv3, TLSv1, or both protocols. The default is `sslSupport_Both`.

**WARNING!** Do not disable things that you don't know anything about. Also, do not turn off the `Ex512/Ex1024` bits without also removing the relevant ciphers from the cipher suite list.

**See Also** `SslContextGet_ProtocolSupport()`, `SslContextSet_ProtocolVersion()`, `SslLibSet_ProtocolSupport()`

## SslContextSet_ProtocolVersion Macro

**Purpose** Specify the version of the SSL protocol to use.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_ProtocolVersion (ssl, v)`

**Parameters**

→ *`ssl`*
The `SslContext` on which to operate.

→ *`v`*
One of the values listed under "Protocol Versions" on page 340.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** SslLib sends a TLSv1 ClientHello message by default. Note that in Palm OS Cobalt version 6.0 an attempt to change this protocol version to SSLv3 has no effect—SslLib continues to send a TLSv1 ClientHello message.

**See Also** "ProtocolVersion" on page 73, `SslContextGet_ProtocolVersion()`, `SslLibSet_ProtocolVersion()`

## SslContextSet_RbufSize Macro

**Purpose** Specify the size, in bytes, of the `SslContext`'s read buffer.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_RbufSize (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
The read buffer size. This is a value that ranges from 0 to 16384.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "RbufSize" on page 64, `SslContextGet_RbufSize()`, `SslLibSet_RbufSize()`

## SslContextSet_ReadStreaming Macro

**Purpose** Specify whether data can be returned to the application from the SSL connection before the full record has been downloaded.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_ReadStreaming (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
A zero value if data is only to be returned when the full record has been downloaded, or a non-zero value otherwise.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "ReadStreaming" on page 77, `SslContextGet_ReadStreaming()`, `SslLibSet_ReadStreaming()`

## SslContextSet_Socket Macro

**Purpose** Specify the socket that the `SslContext` should use to perform its network I/O operations.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_Socket (ssl, v)`

**Parameters**

→ *`ssl`*
The `SslContext` on which to operate.

→ *`v`*
The descriptor referencing the socket.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "Socket" on page 65, `SslContextGet_Socket()`

## SslContextSet_SslSession Macro

**Purpose** Specify the `SslContext`'s `SslSession` structure. This is the `SslSession` for this `SslContext` to use to establish its next connection.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_SslSession (ssl, v)`

**Parameters**

→ *`ssl`*
The `SslContext` on which to operate.

→ *`v`*
A pointer to the `SslSession` structure.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "SslSession" on page 74, `SslContextGet_SslSession()`

## SslContextSet_VerifyCallback Macro

**Purpose** Specify the callback function to assist with certificate verification.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_VerifyCallback (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
A pointer to a callback function of type `SslCallbackFunc()`.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** See "VerifyCallback" on page 65 and `SslCallbackFunc()` (documented on page 373) for more on how this callback function is used to verify a certificate.

**See Also** "VerifyCallback" on page 65, `SslContextGet_VerifyCallback()`, `SslLibSet_VerifyCallback()`

## SslContextSet_WbufSize Macro

**Purpose** Specify the size, in bytes, of the `SslContext`'s write buffer.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslContextSet_WbufSize (ssl, v)`

**Parameters**

→ *ssl*
The `SslContext` on which to operate.

→ *v*
The write buffer size. This is a value that ranges from 0 to 16384.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "WbufSize" on page 66, `SslContextGet_WbufSize()`, `SslLibSet_WbufSize()`

## SslLibGet_AppInt32 Macro

**Purpose** Obtain the value of the AppInt32 attribute, an arbitrary 32-bit value that an application can attach to an SslLib.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_AppInt32 (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** Returns the 32-bit value.

**Comments** `SslLibDestroy()` does not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

**See Also** "AppInt32" on page 67, `SslLibSet_AppInt32()`, `SslContextGet_AppInt32()`

## SslLibGet_AppPtr Macro

**Purpose** Obtain the value of the AppPtr attribute, an arbitrary pointer value that an application can attach to an SslLib.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_AppPtr (lib, v)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

→ *v*
The address of a pointer variable into which the `AppPtr` attribute value is written.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** `SslLibDestroy()` does not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

**See Also** "AppPtr" on page 67, `SslLibSet_AppPtr()`, `SslContextGet_AppPtr()`

## SslLibGet_AutoFlush Macro

**Purpose** Determine whether `SslSend()` and `SslWrite()` attempt to immediately send the supplied data bytes to the network.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_AutoFlush (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** Returns 0 if data is buffered, or 1 if the data is sent immediately.

**Comments** It is very important to remember to use `SslFlush()` when AutoFlush is disabled.

**See Also** "AutoFlush" on page 61, `SslLibSet_AutoFlush()`, `SslContextGet_AutoFlush()`

## SslLibGet_BufferedReuse Macro

**Purpose** Determine if the last message in an SslSession-reused handshake should be buffered instead of being sent over the network.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_BufferedReuse (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** Returns 0 if the last message is *not* buffered, or a non-zero value if it is.

**See Also** "BufferedReuse" on page 75, `SslLibSet_BufferedReuse()`, `SslContextGet_BufferedReuse()`

## SslLibGet_CipherSuites Macro

**Purpose** Obtain the list of cipher suites that the SSL protocol is attempting to use. This list is inherited from the SslLib when an `SslContext` is created.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_CipherSuites (lib, v)`

**Parameters**

→ *lib*
The SslLib from which the value is to be retrieved.

→ *v*
Supply the address of a pointer variable; upon return the pointer variable will point to a series of bytes, where the first two bytes indicate the number of bytes that follow, and each pair of bytes after that is one of the values listed under "Cipher Suites" on page 344. See "CipherSuites" on page 61 for more details on this attribute.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "CipherSuites" on page 61, `SslLibSet_CipherSuites()`, `SslContextGet_CipherSuites()`

## SslLibGet_Compat Macro

**Purpose** Determine which SSL protocol compatibility flags are set. These flags enable compatibility with certain incorrect SSL protocol implementations.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_Compat (lib)`

**Parameters**

→ *lib*
The SslLib from which the value is to be retrieved.

**Returns** Returns a 32-bit integer value that is the logical OR of those compatibility flags that have been set. See "Compatibility Flags" on page 342 for the defined constants that correspond to the compatibility flags.

**See Also** "Compat" on page 69, `SslLibSet_Compat()`, `SslContextGet_Compat()`

## SslLibGet_DelayReadServerFinished Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_DelayReadServerFinished (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns**

## SslLibGet_DontSendShutdown Macro

**Purpose** Determine whether or not an `SslClose()` will send a shutdown message to the server.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_DontSendShutdown (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** Returns zero if `SslClose()` does send a shutdown message to the server, or a non-zero value if it doesn't.

**See Also** "DontSendShutdown" on page 76, `SslLibSet_DontSendShutdown()`, `SslContextGet_DontSendShutdown()`, `SslLibGet_DontWaitForShutdown()`

## SslLibGet_DontWaitForShutdown Macro

**Purpose** Determine whether or not the SslLib will wait for a shutdown message in `SslClose()`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_DontWaitForShutdown (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** Returns zero if the SslLib waits for a shutdown message, or a non-zero value if it doesn't.

**See Also** "DontWaitForShutdown" on page 76, `SslContextGet_DontWaitForShutdown()`, `SslLibGet_DontSendShutdown()`, `SslLibSet_DontWaitForShutdown()`

## SslLibGet_HelloVersion Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_HelloVersion (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns**

## SslLibGet_InfoCallback Macro

**Purpose** Obtain a pointer to the callback function called when various situations occur during the usage of an SslLib. This callback is primarily intended for debugging and feedback purposes.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_InfoCallback (lib, v)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

→ *v*
Pass the address of a pointer variable; upon return this variable will point to the callback function. The callback function is of type `SslCallbackFunc()`.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** See "InfoCallback" on page 69 and `SslCallbackFunc()` (documented on page 373) for more on how this callback function is used.

**See Also** "InfoCallback" on page 69, `SslLibSet_InfoCallback()`, `SslContextGet_InfoCallback()`

## SslLibGet_InfoInterest Macro

**Purpose** Obtain the flags that specify the events for which the Info callback will be called.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_InfoInterest (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** The logical OR of the `sslFlgInfo`*xxx* values listed under "InfoInterest Values" on page 347.

**See Also** "InfoInterest" on page 70, `SslLibSet_InfoInterest()`, `SslContextGet_InfoInterest()`

## SslLibGet_Mode Macro

**Purpose** Obtain the value of the Mode attribute, which controls whether the SSL protocol is on or off.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_Mode (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** Returns one of the values listed under "Mode Attribute Values" on page 339.

**See Also** "Mode" on page 63, `SslLibSet_Mode()`, `SslContextGet_Mode()`

## SslLibGet_ProtocolSupport Macro

**Purpose** Determine which variants of the SSL protocol the library supports.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_ProtocolSupport (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** One or more of the values listed under "Protocol Variants" on page 340, OR'd together.

**See Also** `SslContextGet_ProtocolSupport()`, `SslLibGet_ProtocolVersion()`, `SslLibSet_ProtocolSupport()`

## SslLibGet_ProtocolVersion Macro

**Purpose** Determine which version of the SSL protocol is being used.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_ProtocolVersion (lib)`

**Parameters** → *lib*
The SslLib from which the value is to be retrieved.

**Returns** One of the values listed under "Protocol Versions" on page 340.

**See Also** "ProtocolVersion" on page 73, `SslLibSet_ProtocolVersion()`, `SslContextGet_ProtocolVersion()`

## SslLibGet_RbufSize Macro

**Purpose** Obtain the size, in bytes, of the read buffer. Read and write buffers are associated with an `SslContext`; `SslContexts` created against this SslLib will inherit the SslLib's buffer values.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_RbufSize (lib)`

**Parameters** → *`lib`*
The SslLib from which the value is to be retrieved.

**Returns** The read buffer size. This is a value that ranges from 0 to 16384.

**See Also** "RbufSize" on page 64, `SslLibSet_RbufSize()`, `SslContextGet_RbufSize()`

## SslLibGet_ReadStreaming Macro

**Purpose** Determine whether data can be returned to the application from the SSL connection before the full record has been downloaded.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_ReadStreaming (lib)`

**Parameters** → *`lib`*
The SslLib from which the value is to be retrieved.

**Returns** Returns a zero value if data is only returned when the full record has been downloaded, or a non-zero value otherwise.

**See Also** "ReadStreaming" on page 77, `SslContextGet_ReadStreaming()`, `SslLibSet_ReadStreaming()`

## SslLibGet_VerifyCallback Macro

**Purpose** Obtain a pointer to the callback function used to assist with certificate verification.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_VerifyCallback (lib, v)`

**Parameters**

→ *lib*
The SslLib from which the value is to be retrieved.

→ *v*
Pass the address of a pointer variable; upon return this variable will point to the callback function. The callback function is of type `SslCallbackFunc()`.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** See "VerifyCallback" on page 65 and `SslCallbackFunc()` (documented on page 373) for more on how this callback function is used to verify a certificate.

**See Also** "VerifyCallback" on page 65, `SslLibSet_VerifyCallback()`, `SslContextGet_VerifyCallback()`

## SslLibGet_WbufSize Macro

**Purpose** Obtain the size, in bytes, of the write buffer. Read and write buffers are associated with an `SslContext`; `SslContexts` created against this SslLib will inherit the SslLib's buffer values.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibGet_WbufSize (lib)`

**Parameters**

→ *lib*
The SslLib from which the value is to be retrieved.

**Returns** The write buffer size. This is a value that ranges from 0 to 16384.

**See Also** "WbufSize" on page 66, `SslLibSet_WbufSize()`, `SslContextGet_WbufSize()`

## SslLibSet_AppInt32 Macro

**Purpose** Set the value of the `AppInt32` attribute, an arbitrary 32-bit value that an application can attach to an SslLib.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_AppInt32 (lib, v)`

**Parameters** → *lib*
The SslLib on which to operate.

→ *v*
The 32-bit value.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** `SslLibDestroy()` does not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

**See Also** "AppInt32" on page 67, `SslLibGet_AppInt32()`, `SslContextSet_AppInt32()`

## SslLibSet_AppPtr Macro

**Purpose** Set the value of the `AppPtr` attribute, an arbitrary pointer value that an application can attach to an SslLib.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_AppPtr (lib, v)`

**Parameters** → *lib*
The SslLib on which to operate.

→ *v*
The `AppPtr` attribute value.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** `SslLibDestroy()` does not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

**See Also** "AppPtr" on page 67, `SslLibGet_AppPtr()`, `SslContextSet_AppPtr()`

## SslLibSet_AutoFlush Macro

**Purpose** Specify whether `SslSend()` and `SslWrite()` should attempt to immediately send the supplied data bytes to the network.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_AutoFlush (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*
Pass 0 to have the data buffered, or 1 to have the data sent immediately.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** It is very important to remember to use `SslFlush()` when AutoFlush is disabled.

**See Also** "AutoFlush" on page 61, `SslContextGet_AutoFlush()`, `SslLibSet_AutoFlush()`

## SslLibSet_BufferedReuse Macro

**Purpose** Specify whether the last message in an SslSession-reused handshake should be buffered instead of being sent over the network.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_BufferedReuse (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*

Supply a value of 0 if the last message should *not* be buffered, or a non-zero value if it should.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "BufferedReuse" on page 75, `SslLibGet_BufferedReuse()`, `SslContextSet_BufferedReuse()`

## SslLibSet_CipherSuites Macro

**Purpose** Specify the list of cipher suites that the SSL protocol should attempt to use. When an `SslContext` is created, it inherits this list of cipher suites.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_CipherSuites (lib, v)`

**Parameters**

→ *lib*

The SslLib on which to operate.

→ *v*

Pointer to an array of byte pairs. The first two bytes contain the number of bytes that follow, and each successive pair is one of the values listed under "Cipher Suites" on page 344.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "CipherSuites" on page 61, `SslLibGet_CipherSuites()`, `SslContextSet_CipherSuites()`

## SslLibSet_Compat Macro

**Purpose** Determine how compatible the SSL protocol should be with certain incorrect SSL protocol implementations.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_Compat (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*
The logical OR of those compatibility flags that correspond to the SSL protocol incompatibilities that should be accommodated. See "Compatibility Flags" on page 342 for the defined constants that correspond to the compatibility flags.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "Compat" on page 69, `SslLibGet_Compat()`, `SslContextSet_Compat()`

## SslLibSet_DelayReadServerFinished Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_DelayReadServerFinished (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

## SslLibSet_DontSendShutdown Macro

**Purpose** Specify whether or not an `SslClose()` should send a shutdown message to the server.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_DontSendShutdown (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*
Zero if `SslClose()` should send a shutdown message, or a non-zero value if it shouldn't.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "DontSendShutdown" on page 76, `SslLibGet_DontSendShutdown()`, `SslContextSet_DontSendShutdown()`, `SslLibSet_DontWaitForShutdown()`

## SslLibSet_DontWaitForShutdown Macro

**Purpose** Specify whether or not the SSlLib should wait for a shutdown message in `SslClose()`.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_DontWaitForShutdown (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*
Zero if the SslLib waits for a shutdown message, or a non-zero value if it doesn't.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "DontWaitForShutdown" on page 76, `SslLibGet_DontWaitForShutdown()`, `SslLibSet_DontSendShutdown()`, `SslContextSet_DontWaitForShutdown()`

## SslLibSet_HelloVersion Macro

**Purpose**

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_HelloVersion (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

## SslLibSet_InfoCallback Macro

**Purpose** Specify the callback function called when various situations occur during the usage of an SslLib. This callback is primarily intended for debugging and feedback purposes.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_InfoCallback (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*
A pointer to a callback function of type `SslCallbackFunc()`.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** See "InfoCallback" on page 69 and `SslCallbackFunc()` (documented on page 373) for more on how this callback function is used.

**See Also** "InfoCallback" on page 69, `SslLibGet_InfoCallback()`, `SslContextSet_InfoCallback()`

## SslLibSet_InfoInterest Macro

**Purpose** Specify the events for which the Info callback will be called.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_InfoInterest (`*`lib`*`, `*`v`*`)`

**Parameters**

→ *`lib`*
The SslLib on which to operate.

→ *`v`*
The logical OR of the `sslFlgInfo`*xxx* values listed under "InfoInterest Values" on page 347.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "InfoInterest" on page 70, `SslLibGet_InfoInterest()`, `SslContextSet_InfoInterest()`

## SslLibSet_Mode Macro

**Purpose** Specify the value of the Mode attribute, which controls whether the SSL protocol is on or off.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_Mode (`*`lib`*`, `*`v`*`)`

**Parameters**

→ *`lib`*
The SslLib on which to operate.

→ *v*
One of the values listed under "Mode Attribute Values" on page 339.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "Mode" on page 63, `SslLibGet_Mode()`, `SslContextSet_Mode()`

## SslLibSet_ProtocolSupport Macro

**Purpose** Sets the protocol variants supported by the library.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_ProtocolSupport (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*
One or more of the values listed under "Protocol Variants" on page 340, OR'd together.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** Use `sslSupport_SSLv3`, `sslSupport_TLSv1`, or `sslSupport_Both` to enable SSLv3, TLSv1, or both protocols. The default is `sslSupport_Both`.

**WARNING!** Do not disable things that you don't know anything about. Also, do not turn off the `Ex512/Ex1024` bits without also removing the relevant ciphers from the cipher suite list.

**See Also** `SslContextSet_ProtocolSupport()`, `SslLibGet_ProtocolSupport()`, `SslLibSet_ProtocolVersion()`

## SslLibSet_ProtocolVersion Macro

**Purpose** Specify the version of the SSL protocol to use.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_ProtocolVersion (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*
One of the values listed under "Protocol Versions" on page 340.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** SslLib sends a TLSv1 ClientHello message by default. Note that in Palm OS Cobalt version 6.0 an attempt to change this protocol version to SSLv3 has no effect—SslLib continues to send a TLSv1 ClientHello message.

**See Also** "ProtocolVersion" on page 73, `SslLibGet_ProtocolVersion()`, `SslContextSet_ProtocolVersion()`

## SslLibSet_RbufSize Macro

**Purpose** Specify the size, in bytes, of the read buffer. Read and write buffers are associated with an `SslContext`; `SslContexts` created against this SslLib will inherit the SslLib's buffer values.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_RbufSize (lib, v)`

**Parameters**

→ *lib*
The SslLib on which to operate.

→ *v*
The read buffer size. This is a value that ranges from 0 to 16384.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "RbufSize" on page 64, `SslLibGet_RbufSize()`, `SslContextSet_RbufSize()`

## SslLibSet_ReadStreaming Macro

**Purpose** Specify whether data can be returned to the application from the SSL connection before the full record has been downloaded.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_ReadStreaming (`*`lib`*`, `*`v`*`)`

**Parameters**

→ *`lib`*
The SslLib on which to operate.

→ *`v`*
A zero value if data is only to be returned when the full record has been downloaded, or a non-zero value otherwise.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "ReadStreaming" on page 77, `SslContextSet_ReadStreaming()`, `SslLibGet_ReadStreaming()`

## SslLibSet_VerifyCallback Macro

**Purpose** Specify the callback function to assist with certificate verification.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_VerifyCallback (`*`lib`*`, `*`v`*`)`

**Parameters**

→ *`lib`*
The SslLib on which to operate.

→ *`v`*
A pointer to a callback function of type `SslCallbackFunc()`.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**Comments** See "VerifyCallback" on page 65 and `SslCallbackFunc()` (documented on page 373) for more on how this callback function is used to verify a certificate.

**See Also** "VerifyCallback" on page 65, `SslLibGet_VerifyCallback()`, `SslContextSet_VerifyCallback()`

## SslLibSet_WbufSize Macro

**Purpose** Specify the size, in bytes, of the write buffer. Read and write buffers are associated with an `SslContext`; `SslContexts` created against this SslLib will inherit the SslLib's buffer values.

**Declared In** `SslLibMac.h`

**Prototype** `#define SslLibSet_WbufSize (lib, v)`

**Parameters**

→ *`lib`*
The SslLib on which to operate.

→ *`v`*
The write buffer size. This is a value that ranges from 0 to 16384.

**Returns** Returns `errNone` if the operation completed successfully. Otherwise, the function that this macro evaluates to returns one of the error codes listed under "SSL Library Errors" on page 351.

**See Also** "WbufSize" on page 66, `SslLibGet_WbufSize()`, `SslContextSet_WbufSize()`

# Index

## A

## V