# IBM KeyWorks Toolkit

## Service Provider Module Structure & Administration

June 11, 1999

Copyright© 1999 International Business Machines Corporation. All rights reserved.
Note to U.S. Government Users – Documentation related to restricted rights – Use, duplication, or disclosure is subject to restriction set forth in GSA ADP Schedule Contract with IBM Corp.
IBM is a registered trademark of International Business Machines Corporation, Armonk, N.Y.

Copyright© 1997 Intel Corporation. All rights reserved.
Intel Corporation, 5200 N. E. Elam Young Parkway, Hillsboro, OR 97124-6497.

Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owner's benefit, without intent to infringe.
**001.001.004**

# Table of Contents


**CHAPTER 1. INTRODUCTION ........ 1**
1.1 SERVICE PROVIDER MODULES ........ 1
1.2 INTENDED AUDIENCE ........ 2
1.3 DOCUMENTATION SET ........ 2
1.4 REFERENCES ........ 3

**CHAPTER 2. MODULE STRUCTURE AND ADMINISTRATION ........ 5**
2.1 SECURITY SERVICES ........ 5
*2.1.1 Module-to-Module Interaction ........ 5*
2.2 MODULE ADMINISTRATION COMPONENTS ........ 6
2.3 INSTALLING A SERVICE PROVIDER MODULE ........ 6
2.4 ATTACHING A SERVICE PROVIDER MODULE ........ 7
*2.4.1 Module Entry Point ........ 7*
*2.4.2 Module Function Table Registration ........ 7*
*2.4.3 Memory Management Upcalls ........ 8*
2.5 ERROR HANDLING ........ 8
2.6 INSTALL EXAMPLE ........ 9
*2.6.1 CL Module Install ........ 9*
2.7 ATTACH/DETACH EXAMPLE ........ 11
*2.7.1 AddInAuthenticate ........ 11*

**CHAPTER 3. SERVICE PROVIDER MODULE INTERFACE FUNCTIONS ........ 12**
3.1 DATA STRUCTURES ........ 12
*3.1.1 CSSM_HANDLEINFO ........ 12*
*3.1.2 CSSM_MODULE_FUNCS ........ 12*
*3.1.3 CSSM_REGISTRATION_INFO ........ 13*
3.2 SERVICE PROVIDER MODULE FUNCTIONS ........ 15
*3.2.1 CSSM_DeregisterServices ........ 15*
*3.2.2 CSSM_GetHandleInfo ........ 16*
*3.2.3 CSSM_RegisterServices ........ 17*
*3.2.4 EventNotify ........ 18*
*3.2.5 FreeModuleInfo ........ 20*
*3.2.6 GetModuleInfo ........ 21*
*3.2.7 Initialize ........ 23*
*3.2.8 Terminate ........ 24*

**CHAPTER 4. RELEVANT CSSM API FUNCTIONS ........ 25**
4.1 DATA STRUCTURES ........ 25
*4.1.1 Basic Data Types ........ 25*
*4.1.2 CSSM_ALL_SUBSERVICES ........ 25*
*4.1.3 CSSM_BOOL ........ 25*
*4.1.4 CSSM_CALLBACK ........ 26*
*4.1.5 CSSM_CRYPTO_DATA ........ 26*
*4.1.6 CSSM_DATA ........ 26*
*4.1.7 CSSM_GUID ........ 27*
*4.1.8 CSSM_HANDLE ........ 27*
*4.1.9 CSSM_INFO_LEVEL ........ 27*
*4.1.10 CSSM_MEMORY_FUNCS / CSSM_API_MEMORY_FUNCS ........ 28*
*4.1.11 CSSM_MODULE_FLAGS ........ 28*
*4.1.12 CSSM_MODULE_HANDLE ........ 28*
*4.1.13 CSSM_MODULE_INFO ........ 29*

*4.1.14 CSSM_NOTIFY_CALLBACK* ........ *30*
*4.1.15 CSSM_RETURN* ........ *30*
*4.1.16 CSSM_SERVICE_FLAGS* ........ *30*
*4.1.17 CSSM_SERVICE_INFO* ........ *31*
*4.1.18 CSSM_SERVICE_MASK* ........ *32*
*4.1.19 CSSM_SERVICE_TYPE* ........ *32*
*4.1.20 CSSM_SPI_FUNC_TBL* ........ *32*
*4.1.21 CSSM_USER_AUTHENTICATION* ........ *33*
*4.1.22 CSSM_USER_AUTHENTICATION_MECHANISM* ........ *33*
*4.1.23 CSSM_VERSION* ........ *33*
4.2 FUNCTION DEFINITIONS ........ 34
*4.2.1 CSSM_ModuleAttach* ........ *34*
*4.2.2 CSSM_ModuleDetach* ........ *36*
*4.2.3 CSSM_FreeModuleInfo* ........ *37*
*4.2.4 CSSM_GetCSSMRegistryPath* ........ *38*
*4.2.5 CSSM_GetGUIDUsage* ........ *39*
*4.2.6 CSSM_GetHandleUsage* ........ *40*
*4.2.7 CSSM_GetModuleGUIDFromHandle* ........ *41*
*4.2.8 CSSM_GetModuleInfo* ........ *42*
*4.2.9 CSSM_GetModuleLocation* ........ *44*
*4.2.10 CSSM_ListModules* ........ *45*
*4.2.11 CSSM_ModuleInstall* ........ *46*
*4.2.12 CSSM_SetModuleInfo* ........ *47*
*4.2.13 CSSM_ModuleUninstall* ........ *48*
*4.2.14 CSSM_SetModuleInfo* ........ *49*
*4.2.15 CSSM_FreeModuleInfo* ........ *50*
*4.2.16 CSSM_GetError* ........ *51*
*4.2.17 CSSM_SetError* ........ *52*
*4.2.18 CSSM_ClearError* ........ *53*

**APPENDIX A. KEYWORKS ERRORS ........ 54**

**APPENDIX B. LIST OF ACRONYMS ........ 55**

**APPENDIX C. GLOSSARY ........ 56**

# List of Figures

Figure 1. IBM KeyWorks Toolkit Architecture ........ 2

# List of Tables

Table 1. Service Access Table ........ 13
Table 2. Module Event Types ........ 18
Table 3. Module Event Parameters ........ 18
Table 4. Notification Reasons ........ 30
Table 5. Invalid Errors ........ 54

# Chapter 1. Introduction

The IBM KeyWorks Toolkit defines the infrastructure for a complete set of security services. It is an extensible architecture that provides mechanisms to manage service provider security modules, which use cryptography as a computational base to build security protocols and security systems. Figure 1 shows the four basic layers of the IBM KeyWorks Toolkit: Application Domains, System Security Services, IBM KeyWorks Framework, and Service Providers. The IBM KeyWorks Framework is the core of this architecture. It provides a means for applications to directly access security services through the KeyWorks security application programming interface (API), or to indirectly access security services via layered security services and tools implemented over the KeyWorks API. The IBM KeyWorks Framework manages the service provider security modules and directs application calls through the KeyWorks API to the selected service provider module that will service the request. The KeyWorks API defines the interface for accessing security services. The KeyWorks service provider interface defines the interface for service providers who develop plug-able security service products.

Service providers perform various aspects of security services, including:

- Cryptographic Services
- Key Recovery Services
- Trust Policy Libraries
- Certificate Libraries
- Data Storage Libraries

Cryptographic Service Providers (CSPs) are service provider modules that perform cryptographic operations including encryption, decryption, digital signing, key pair generation, random number generation, (RNG) and key exchange. Key Recovery Service Providers (KRSPs) generate and process Key Recovery Fields (KRFs) which can be used to retrieve the original session key if it is lost, or if an authorized party requires access to the decryption key. Trust Policy (TP) modules implement policies defined by authorities and institutions, such as VeriSign (as a Certificate Authority (CA)) or MasterCard (as an institution). Each TP module embodies the semantics of a trust model based on using digital certificates as credentials. Applications may use a digital certificate as an identity credential and/or an authorization credential. Certificate Library (CL) modules provide format-specific, syntactic manipulation of memory-resident digital certificates and Certificate Revocation Lists (CRLs). Data Storage Library (DL) modules provide persistent storage for certificates and CRLs.

## 1.1 Service Provider Modules

An IBM KeyWorks service provider module is a Dynamically Linked Library (DLL) composed of functions that implement some or all of the KeyWorks module interfaces. Applications directly or indirectly select the modules used to provide security services to the application. Independent Software Vendors (ISVs) and hardware vendors will provide these service providers. The functionality of the service providers may be extended beyond the services defined by the KeyWorks API, by exporting additional services to applications using a KeyWorks PassThrough mechanism.

The API calls defined for service provider modules are categorized as service operations, module management operations, and module-specific operations. Service operations include functions that perform a security operation such as encrypting data, inserting a CRL into a data source, or verifying that a certificate is trusted. Module management functions support module installation, registration of module features and attributes, and queries to retrieve information on module availability and features. Module-specific operations are enabled in the API through pass through functions whose behavior and use is defined by the service provider module developer.

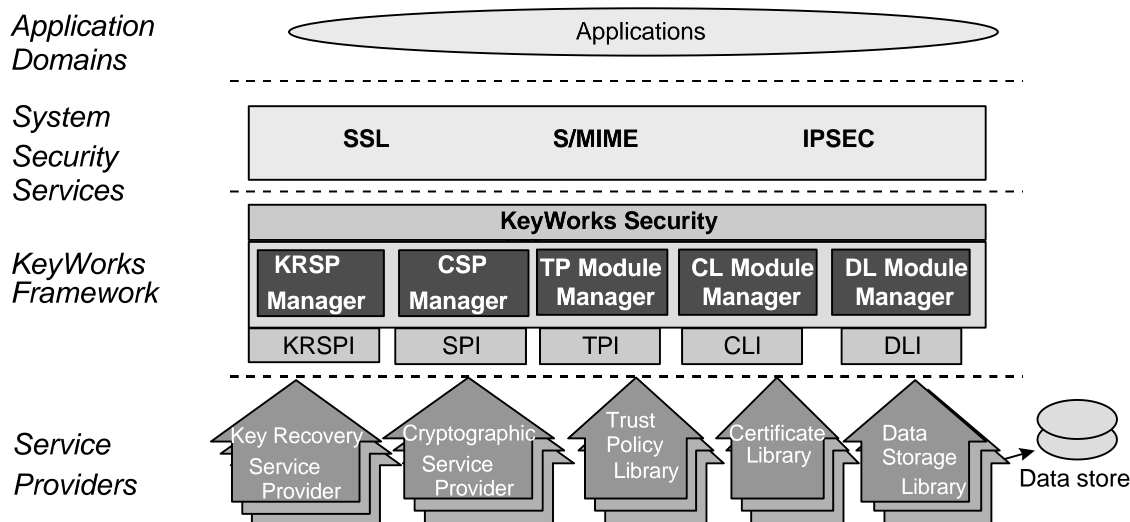


**Figure 1**. **IBM KeyWorks Toolkit Architecture**

Each module, regardless of the security services it offers, has the same set of module management responsibilities. Every module must expose functions that allow KeyWorks to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of KeyWorks, and register with KeyWorks.

## 1.2 Intended Audience

ISVs who want to develop their own TP service provider modules should use this document. These ISVs can be highly experienced software and security architects, advanced programmers, and sophisticated users. The intended audience of this document must be familiar with high-end cryptography and digital certificates. They must also be familiar with local and foreign government regulations on the use of cryptography and the implication of those regulations for their applications and products. We assume that this audience is familiar with the basic capabilities and features of the protocols they are considering.

## 1.3 Documentation Set

The IBM KeyWorks Toolkit documentation set consists of the following manuals. These manuals are provided in electronic format and can be viewed using the Adobe Acrobat Reader distributed with the IBM KeyWorks Toolkit. Both the electronic manuals and the Adobe Acrobat Reader are located in the IBM KeyWorks Toolkit `doc` subdirectory.

- *IBM KeyWorks Toolkit Developer's Guide*
  Document filename: kw_dev.pdf
  This document presents an overview of the IBM KeyWorks Toolkit. It explains how to integrate IBM KeyWorks into applications and contains a sample IBM KeyWorks application.

- *IBM KeyWorks Toolkit Application Programming Interface Specification*
  Document filename: kw_api.pdf
  This document defines the interface that application developers employ to access security services provided by IBM KeyWorks and service provider modules.

- *IBM KeyWorks Toolkit Service Provider Module Structure & Administration Specification*.
  Document filename: kw_mod.pdf
  This document describes the features common to all IBM KeyWorks service provider modules. It should be used in conjunction with the IBM KeyWorks service provider interface specifications in order to build a security service provider module.

- *IBM KeyWorks Toolkit Cryptographic Service Provider Interface Specification*
  Document filename: kw_spi.pdf
  This document defines the interface to which cryptographic service providers must conform in order to be accessible through IBM KeyWorks.

- *Key Recovery Service Provider Interface Specification*
  Document filename: kr_spi.pdf
  This document defines the interface to which key recovery service providers must conform in order to be accessible through IBM KeyWorks.

- *Key Recovery Server Installation and Usage Guide*
  Document filename: krs_gd.pdf
  This document describes how to install and use key recovery solutions using the components in the IBM Key Recovery Server.

- *IBM KeyWorks Toolkit Trust Policy Interface Specification*
  Document filename: kw_tp_spi.pdf
  This document defines the interface to which policy makers, such as certificate authorities, certificate issuers, and policy-making application developers, must conform in order to extend IBM KeyWorks with model or application-specific policies.

- *IBM KeyWorks Toolkit Certificate Library Interface Specification*
  Document filename: kw_cl_spi.pdf
  This document defines the interface to which library developers must conform to provide format-specific certificate manipulation services to numerous IBM KeyWorks applications and trust policy modules.

- *IBM KeyWorks Toolkit Data Storage Library Interface Specification*
  Document filename: kw_dl_spi.pdf
  This document defines the interface to which library developers must conform to provide format-specific or format-independent persistent storage of certificates.

## 1.4 References


| | |
|---|---|
| Cryptography | *Applied Cryptography*, Schneier, Bruce, 2nd Edition, John Wiley and Sons, Inc., 1996. |
| | *Handbook of Applied Cryptography,* Menezes, A., Van Oorschot, P., and Vanstone, S., CRC Press, Inc., 1997. |
| | *SDSI - A Simple Distributed Security Infrastructure,* R. Rivest and B. Lampson, 1996. |
| | *Microsoft CryptoAPI, Version 0.9*, Microsoft Corporation, January 17, 1996. |
| CDSA Spec | *Common Data Security Architecture Specification,* Intel Architecture Labs, 1997. |

| | |
|---|---|
| CSSM API | *Common Security Services Manager Application Programming Interface Specification,* Intel Architecture Labs, 1997. |
| Key Escrow | *A Taxonomy for Key Escrow Encryption Systems,* Denning, Dorothy E. and Branstad, Dennis, Communications of the ACM, Vol. 39, No. 3, March 1996. |
| PKCS | *The Public-Key Cryptography Standards*, RSA Laboratories, Redwood City, CA: RSA Data Security, Inc. |
| IBM KeyWorks CLI | *Certificate Library Interface Specification,* Intel Architecture Labs, 1997. |
| IBM KeyWorks DLI | *Data Storage Library Interface Specification,* Intel Architecture Labs, 1997. |
| IBM KeyWorks KRI | *Key Recovery Service Provider Interface Specification,* Intel Architecture Labs, 1997. |
| IBM KeyWorks SPI | *Cryptographic Service Provider Interface Specification,* Intel Architecture Labs, 1997. |
| IBM KeyWorks TPI | *Trust Policy Interface Specification,* Intel Architecture Labs, 1997. |
| X.509 | *CCITT. Recommendation X.509: The Directory – Authentication Framework,* 1988. CCITT stands for Comite Consultatif Internationale Telegraphique et Telephonique (International Telegraph and Telephone Consultative Committee) |

# Chapter 2. Module Structure and Administration

Service provider modules are composed of module administration components and implementation of security service interfaces in one or more categories of service. Module administration components include the tasks required during module installation, attach, and detach. The module developer determines the number, categories, and contents of the service implementation. Both the administration components and service interfaces are discussed in the following sections.

## 2.1 Security Services

The primary components of a service provider module are the security services that it offers. A service provider module may provide one to four categories of service, with each service having one or more available subservices. The service categories are Cryptographic Service Provider (CSP) services, key recovery services, Trust Policy (TP) services, Certificate Library (CL) services, and Data Storage Library (DL) services. A subservice consists of a unique set of capabilities within a certain service. For example, in a CSP service providing access to hardware tokens, each subservice would represent a slot. A TP service may have one subservice that supports the Secure Electronic Transfer (SET) Merchant TP and a second subservice that supports the SET Cardholder TP. A CL service may have different subservices for different encoding formats. A DL service could use subservices to represent different types of persistent storage. In all cases, the subservice implements the basic service functions for its category of service.

Each service category contains a number of basic service functions. A library developer may choose to implement some or all of the functions specified in the service interface. A module developer may also choose to extend the basic interface functionality by exposing pass through operations. Details of the functions and their expected behavior can be found in the IBM KeyWorks Toolkit service provider interface (SPI) documents for the individual service provider modules.

### 2.1.1 Module-to-Module Interaction

Modules may make use of other IBM KeyWorks service provider modules to implement their functionality. For example, a module implementing a CL may use the capabilities of a CSP module to perform the cryptographic operations of sign and verify. In that case, the CL module could package the certificate or Certificate Revocation List (CRL) fields to be signed or verified, attach to the appropriate CSP module, and call CSSM_SignData or CSSM_VerifyData to perform the operation.

A second form of module-to-module interaction is subservice collaboration. For example, a Public-Key Cryptographic Standard (PKCS#11) module may require collaborating CSP and DL subservices. Collaborating subservices are assumed to share state. A module indicates that two or more subservices collaborate by assigning them the same subservice ID. When an application attaches one of the collaborating subservices, it will receive a handle that may be used to access any of the subservices having the same subservice ID. This mechanism may be used for collaboration across categories of services, but is not available within a single category of service.

Subservices may make use of other products or services as part of their implementation. For example, an Open Database Connectivity (ODBC) DL subservice may make use of a commercial database product such as Microsoft Access. A CL subservice may make use of a Certificate Authority (CA) service, such as the VeriSign DigitalID Center, for filling certification requests. The encapsulation of these products and services is exposed to applications in the CSSM_XX_WRAPPEDPRODUCT_INFO data structure, which is available by querying the KeyWorks registry.

A module developer may provide additional utility libraries for use by other module developers. Utility libraries are software components that contain functions that may be useful to several modules. For example, a utility library that performs DER encoding might be useful to several modules providing CL services. The utility library developer is responsible for making the definition, interpretation, and usage of their library available to other module developers.

## 2.2 Module Administration Components

Every module implementation shares certain administrative tasks that must be performed during module installation, attach, and detach. As part of module installation, the module developer must register information about the module's services with IBM KeyWorks. This information is stored in the KeyWorks registry and may be queried by applications using the CSSM_GetModuleInfo function.

On attach, the module's administrative responsibilities include authentication, module registration, and module initialization. Authentication is a protocol whereby KeyWorks ensures the integrity of the service provider module prior to attaching into the system. Because service provider modules are dynamic components in the system, KeyWorks requires the use of a strong verification mechanism to screen all components as they are added to the KeyWorks environment. This aids in KeyWorks's detection and protection against classic forms of attack, such as stealth and man-in-the-middle attacks.

Following authentication, the module registers its functions with KeyWorks and performs any initialization operations. The module uses CSSM_RegisterServices to register a function table with KeyWorks for each subservice that it supports. The function tables consist of pointers to the subservice functions supported by the module. During future function calls from the application, KeyWorks will use these function pointers to direct calls to the appropriate module subservice. When the module is detached, it performs any necessary cleanup actions.

## 2.3 Installing a Service Provider Module

Every module must include functions for module initialization and cleanup. The first time the module is attached, KeyWorks calls the module's Initialize function to allow the module to perform any necessary initialization operations. The last time the module is detached, KeyWorks calls the Terminate function that allows the module to perform any necessary cleanup actions. KeyWorks will call the module's EventNotify function as part of every attach and detach operation.

Before an application can use a module, the module's name, location, and description must be registered with KeyWorks by an installation application. The name given to a module includes both a logical name and a Globally Unique ID (GUID). The logical name is a string chosen by the module developer to describe the module. The GUID is a structure used to differentiate between service provider modules in the KeyWorks registry. GUIDs are discussed in more detail later in this section. The location of the module is required at installation time so the KeyWorks can locate the module and its credentials when an application requests an attach. The module description indicates to KeyWorks the security services available within this module. The module description is clarified below.

Each module must have a GUID that the KeyWorks, applications, and the module itself use to uniquely identify a given module. The GUID is used by the KeyWorks registry to expose service provider module availability and capabilities to applications. A module uses its GUID to identify itself when it sets an error. When attaching the library, the application uses the GUID to identify the requested module.

A GUID is defined below.  GUID generators are publicly available for Windows 95, Windows NT, and on many UNIX-based platforms.

```
typedef struct cssm_guid {
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8 Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR;
```

At install time, the installation program must inform KeyWorks of the ways in which this module can be used. The module usage information includes indicators of the overall module capabilities and descriptions of the security services available from this module.  The overall module capabilities include indicators such as the module’s threading properties or exportability.  The security service descriptions include information on each service, its subservices, and any embedded products or services.   For example, a module description might indicate that this is an exportable module containing a DL service and a CSP service, where the CSP service provides one subservice to access a software token and a second subservice to access a hardware token.  The module description is made available to applications via queries to the KeyWorks registry.

## 2.4 Attaching a Service Provider Module

Before an application can use the functions of a specific module subservice, it must use the CSSM_ModuleAttach function to request that KeyWorks attach to the module’s subservice.  On the first attach, KeyWorks verifies the integrity of the service provider module prior to loading the module. Loading the module initiates a call to an operating system (OS-specific) entry point in the module.  On registration, the service provider module registers its tables of service function pointers with KeyWorks and receives the application’s memory management upcalls.  KeyWorks then uses the module function table to call the module’s Initialize function to confirm version compatibility and calls the module’s EventNotify function to indicate that an attach operation is occurring.  Once these steps have successfully completed, KeyWorks returns a module handle to the application that uniquely identifies the pairing of the application thread to the module subservice instance.  The application uses this handle to identify the module subservice in future function calls.  The module subservice uses the handle to identify the calling application.  KeyWorks notifies the module of subsequent attach requests from the application by using the module’s EventNotify function.  Subsequent attach operations do not require integrity verification.

### 2.4.1 Module Entry Point

When KeyWorks first attaches to or last detaches from a module, it initiates an OS-specific entry point. For the Windows NT OS, DLLMain is the entry point.  For AIX,  _init and  _fini are the entry points.  On attach, this function is responsible for authenticating KeyWorks and then calling CSSM_RegisterServices. On detach, it is responsible for calling CSSM_DeregisterServices.  To avoid OS-related conflicts, any setup or cleanup operations should be performed in the module’s Initialize and Terminate functions.

### 2.4.2 Module Function Table Registration

On attach, a module must register its function tables with KeyWorks by calling CSSM_RegisterServices. Its function tables consist of a table of module management function pointers, plus one table of  SPI function pointers for each (service, subservice) pair contained in the module.  The module management functions include Initialize, EventNotify, and Terminate.  The interface functions reflect the KeyWorks API for each security service.  The function prototypes and their descriptions provide the IBM KeyWorks Toolkit SPI specifications.  See Section 1.3 for a complete list of these documents.  If a subservice does not

support a given function in its SPI, the pointer to that function should be set to NULL. These structures are specified in the KeyWorks header files, cssmspi.h, cssmcspi.h, cssmtpi.h, cssmcli.h, and cssmdli.h.

### 2.4.3 Memory Management Upcalls

All memory allocation and deallocation for data passed between the application and a module via KeyWorks is ultimately the responsibility of the calling application. Since a module needs to allocate memory to return data to the application, the application must provide the module with a means of allocating memory that the application has the ability to free. It does this by providing the module with memory management upcalls.

Memory management upcalls are pointers to the memory management functions used by the calling application. They are provided to a module via KeyWorks as a structure of function pointers and are passed to the module when it calls the CSSM_RegisterServices function. The functions will be the calling application's equivalent of malloc, free, calloc, and re-alloc, and will be expected to have the same behavior as those functions. The function parameters will consist of the normal parameters for that function. The function return values should be interpreted in the standard manner. A module is responsible for making the memory management functions available to all of its internal functions.

## 2.5 Error Handling

When an error occurs inside a module, the function should call CSSM_SetError. The CSSM_SetError function takes the module's GUID and an error number as inputs. The module's GUID is used to identify where the error occurred. The error number is used to describe the error.

The error number set by a module subservice should fall into one of two ranges. The first range of error numbers is predefined by KeyWorks. These are errors that are common to all modules implementing a given subservice function. They are defined in the header file, cssmerr.h, which is distributed as part of KeyWorks. The second range of error numbers is used to define module-specific error codes. These module-specific error codes should be in the range of CSSM_XX_PRIVATE_ERROR to CSSM_XX_END_ERROR, where XX stands for the service category abbreviation (CSP, KRSP, TP, CL, DL). CSSM_XX_PRIVATE_ERROR and CSSM_XX_END_ERROR are also defined in the header file cssmerr.h. A module developer is responsible for making the definition and interpretation of their module-specific error codes available to applications.

When no error has occurred, but the appropriate return value from a function is CSSM_FALSE, that function should call CSSM_ClearError before returning. When the application receives a CSSM_FALSE return value, it is responsible for checking whether an error has occurred by calling CSSM_GetError. If the module function has called CSSM_ClearError, the calling application receives a CSSM_OK response from the CSSM_GetError function, indicating no error has occurred.

## 2.6 Install Example

An installation program is responsible for registering a module's capabilities with KeyWorks. A sample code segment for the installation of a CL Module is shown in the example below. This example runs on Windows-based systems.

### 2.6.1 CL Module Install

```
#include "cssm.h"
CSSM_GUID clm_guid =
{ 0x5fc43dc1, 0x732, 0x11d0, { 0xbb, 0x14, 0x0, 0xaa, 0x0, 0x36, 0x67, 0x2d }
};
CSSM_BOOL CLModuleInstall()
{
    CSSM_VERSION        cssm_version = { CSSM_MAJOR, CSSM_MINOR };
    CSSM_VERSION        cl_version = { CLM_MAJOR_VER, CLM_MINOR_VER };
    CSSM_GUID           cl_guid = clm_guid;
    CSSM_CLSUBSERVICE   sub_service;
    CSSM_SERVICE_INFO   service_info;
    CSSM_MODULE_INFO    module_info;
    char                SysDir[_MAX_PATH];

    /* fill subservice information */
    sub_service.SubServiceId = 0;
    strcpy(sub_service.Description, "X509v3 SubService");
    sub_service.CertType = CSSM_CERT_X_509v3;
    sub_service.CertEncoding = CSSM_CERT_ENCODING_DER;
    sub_service.AuthenticationMechanism = CSSM_AUTHENTICATION_NONE;
    sub_service.NumberOfTemplateFields = NUMBER_X509_CERT_OIDS;
    sub_service.CertTemplates = X509_CERT_OIDS_ARRAY;
    sub_service.NumberOfTranslationTypes = 0;
    sub_service.CertTranslationTypes = NULL;
    sub_service.WrappedProduct.EmbeddedEncoderProducts = NULL;
    sub_service.WrappedProduct.NumberOfEncoderProducts = 0;
    sub_service.WrappedProduct.AccessibleCAProducts = NULL;
    sub_service.WrappedProduct.NumberOfCAProducts = 0;

    /* fill service information */
    strcpy(service_info.Description, "CL Service");
    service_info.Type = CSSM_SERVICE_CL;
    service_info.Flags = 0;
    service_info.NumberOfSubServices = 1;
    service_info.ClSubServiceList = &sub_service;
    service_info.Reserved = NULL;

    /* fill module information */
    module_info.Version = cl_version;
    module_info.CompatibleCSSMVersion = cssm_version;
    strcpy(module_info.Description, "Vendor Module");
    strcpy(module_info.Vendor, "Vendor Name");
    module_info.Flags = 0;
    module_info.ServiceMask = CSSM_SERVICE_CL;
    module_info.NumberOfServices = 1;
    module_info.ServiceList = &service_info;
    module_info.Reserved = NULL;

    /* get system dir path */
    GetSystemDirectory(SysDir, _MAX_PATH);
```

```
    /* Install the module */
    if (CSSM_ModuleInstall(clm_fullname_string,
                           clm_filename_string,
                           SysDir,
                           &clm_guid,
                           &module_info,
                           NULL,
                           NULL) == CSSM_FAIL)
    {
        return CSSM_FALSE;
    }

    return CSSM_TRUE;
}
```

## 2.7 Attach/Detach Example

A module is responsible for performing certain operations when KeyWorks attaches to and detaches from it. These operations should be performed in a function called AddInAuthenticate, which must be exported by the module. The AddInAuthenticate function will be called by the framework when the module is loaded. The steps shown in Section 2.7.1 must be performed in order for the attach process to work properly.

In the code example in Section 2.7.1, it is assumed that the CSSM entry points, such as CSSM_RegisterServices, have been resolved at link time. If not, the module may call GetProcAddress to resolve the entry points. The module is a CSP in this example, and the functions registered with the framework are CSP functions, although the unimplemented functions in the function table are initialized to NULL, and not reassigned.

### 2.7.1 AddInAuthenticate

```
#include "cssm.h"

CSSM_SPI_MEMORY_FUNCS     CssmMemFuncs;
CSSM_GUID CspGuid =
{ 0x83badc39, 0xfac1, 0x11cf, { 0x81, 0x72, 0x0, 0xaa, 0x0, 0xb1, 0x99, 0xdd }
};

CSSM_RETURN CSSMAPI AddInAuthenticate(char* cssmCredentialPath, char*
cssmSection)
{
      CSSM_SPI_CSP_FUNCS         CssmCSPFuncs;
      CSSM_REGISTRATION_INFO     CssmRegInfo;
      CSSM_MODULE_FUNCS          CssmModuleFuncs[1];
      CSSM_RETURN                retcode;

      // initialize tables
      memset(&CssmCSPFuncs, 0, sizeof(CSSM_SPI_CSP_FUNCS));
      memset(&CssmRegInfo, 0, sizeof(CSSM_REGISTRATION_INFO));

      // Now register services
      CssmCSPFuncs.DecryptData         = DecryptData;
      CssmCSPFuncs.EncryptData         = EncryptData;

      CssmRegInfo.Initialize           = Initialize;
      CssmRegInfo.Terminate            = Uninitialize;
      CssmRegInfo.EventNotify          = EventNotify;
      CssmRegInfo.ThreadSafe           = CSSM_TRUE;
      CssmRegInfo.ServiceSummary       = CSSM_SERVICE_CSP;
      CssmRegInfo.NumberOfServiceTables = 1;
      CssmRegInfo.Services                    = CssmModuleFuncs;

      CssmModuleFuncs[0].ServiceType = CSSM_SERVICE_CSP;
      CssmModuleFuncs[0].CspFuncs = &CssmCSPFuncs;

      retcode = CSSM_RegisterServices(&CspGuid, &CssmRegInfo, &CssmMemFuncs,
      NULL);

      return retcode;
}
```

# Chapter 3. Service Provider Module Interface Functions

## 3.1 Data Structures

This section describes the data structures that may be passed to or returned from a service provider module function. They will be used by modules to prepare data to be passed to and from the calling application via the IBM KeyWorks Framework. These data structures are defined in the header file, cssmspi.h, which is distributed with the IBM KeyWorks Toolkit. Data structures that are specific to a particular type of service provider module, such as a Cryptographic Service Provider (CSP) or a Key Recovery Service Provider (KRSP), are described in the individual IBM KeyWorks service provider interface (SPI) specification documents.

### 3.1.1 CSSM_HANDLEINFO

This structure is used by service provider modules to obtain information about a CSSM_HANDLE.

```
typedef struct cssm_handleinfo {
    uint32 SubServiceID;
    uint32 SessionFlags;
    CSSM_NOTIFY_CALLBACK Callback;
    uint32 ApplicationContext;
} CSSM_HANDLEINFO, *CSSM_HANDLEINFO_PTR;
```

Definitions:

*SubServiceID* - An identifier for this subservice.

*SessionFlags* – A bit-mask of service options defined by a particular subservice of the module. Legal values are described in the module-specific documentation. A default set of flags is specified in the CSSM_MODULE_INFO structure for use by the caller.

*Callback* - A callback function registered by the application as part of the module attach operation. This function should be used to notify the application of certain events.

*ApplicationContext* - An identifier which should be passed back to the application as part of the Callback function.

### 3.1.2 CSSM_MODULE_FUNCS

This structure is used by service provider modules to pass a table of function pointers for a single service to KeyWorks.

```
typedef struct cssm_module_funcs {
    CSSM_SERVICE_TYPE ServiceType;
    union {
        void *ServiceFuncs;
        CSSM_SPI_CSP_FUNCS_PTR CspFuncs;
        CSSM_SPI_DL_FUNCS_PTR  DlFuncs;
        CSSM_SPI_CL_FUNCS_PTR  ClFuncs;
        CSSM_SPI_TP_FUNCS_PTR  TpFuncs;
        CSSM_SPI_KRSP_FUNCS_PTR KrspFuncs;
        };
} CSSM_MODULE_FUNCS, *CSSM_MODULE_FUNCS_PTR;
```

Definitions:

*ServiceType* - The type of service provider module services accessible via the XXFuncs function table.

*XXFuncs* - A pointer to a function table of the type described by ServiceType These function pointers are used by KeyWorks to direct function calls from an application to the appropriate service in the service provider module. These function pointer tables are described in the KeyWorks header files cssmcspi.h, cssmkrspi.h, cssmdli.h, cssmcli.h, and cssmtpi.h. Table 1 provides the service access tables.

**Table 1. Service Access Table**

| **Value** | **Description** |
|---|---|
| CSSM_SPI_CSP_FUNCS_PTR CspFuncs | Function pointers to CSP services |
| CSSM_SPI_KRSP_FUNCS_PTR KrspFuncs | Function pointers to KR services |
| CSSM_SPI_DL_FUNCS_PTR DlFuncs | Function pointers to DL services |
| CSSM_SPI_CL_FUNCS_PTR ClFuncs | Function pointers to CL services |
| CSSM_SPI_TP_FUNCS_PTR TpFuncs | Function pointers to TP services |

### 3.1.3 CSSM_REGISTRATION_INFO

This structure is used by service provider modules to pass tables of function pointers and module information to KeyWorks.

```
typedef struct cssm_registration_info {
    /* Loading, Unloading and Event Notifications */
    CSSM_RETURN (CSSMAPI *Initialize) (CSSM_MODULE_HANDLE Handle,
                                       uint32 VerMajor,
                                       uint32 VerMinor);
    CSSM_RETURN (CSSMAPI *Terminate)  (CSSM_MODULE_HANDLE Handle);
    CSSM_RETURN (CSSMAPI *EventNotify)(CSSM_MODULE_HANDLE Handle,
                                        const CSSM_EVENT_TYPE Event,
                                        const uint32 Param);
    CSSM_MODULE_INFO_PTR (CSSMAPI *GetModuleInfo)
                                      (CSSM_MODULE_HANDLE ModuleHandle,
                                       CSSM_SERVICE_MASK ServiceMask,
                                       uint32 SubserviceID,
                                       CSSM_INFO_LEVEL InfoLevel);
    CSSM_RETURN (CSSMAPI *FreeModuleInfo) (CSSM_MODULE_HANDLE ModuleHandle,
                                           CSSM_MODULE_INFO_PTR ModuleInfo);
    CSSM_BOOL ThreadSafe;
    uint32 ServiceSummary;
    uint32 NumberOfServiceTables;
    CSSM_MODULE_FUNCS_PTR Services;
} CSSM_REGISTRATION_INFO, *CSSM_REGISTRATION_INFO_PTR;
```

Definitions:

*Initialize* - Pointer to function that verifies compatibility of the requested module version with the actual module version, and which performs module setup operations.

*Terminate* - Pointer to function that performs module cleanup operations.

*EventNotify* - Pointer to function that accepts event notification from IBM KeyWorks.

*GetModuleInfo*- Pointer to function that obtains and returns dynamic information about the module.

*FreeModuleInfo* - Pointer to function that frees the module information structure.

*ThreadSafe* - A flag that indicates to KeyWorks whether or not the module is capable of handling multithreaded access.

*ServiceSummary* - A bit-mask indicating the types of services offered by this module. It is the bitwise-OR of the service types described in Table 1.

*NumberOfServiceTables* - The number of distinct services provided by this module. This is also the length of the *Services* array.

*Services* - An array of CSSM_MODULE_FUNCS structures that provide the mechanism for accessing the module's services.

## 3.2 Service Provider Module Functions

A service provider module interfaces with KeyWorks using the functions described in this section.

### 3.2.1 CSSM_DeregisterServices

**CSSM_RETURN CSSMAPI CSSM_DeregisterServices** (const CSSM_GUID_PTR GUID)

This function is used by a service provider module to deregister its function table with KeyWorks.

**Parameters**

*GUID (input)*
A pointer to the CSSM_GUID structure containing the Globally Unique ID (GUID) for this module.

**Return Value**

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

**See Also**

CSSM_RegisterServices

### 3.2.2 CSSM_GetHandleInfo

**CSSM_HANDLEINFO_PTR CSSMAPI CSSM_GetHandleInfo** (CSSM_HANDLE hModule)

This function retrieves a CSSM_HANDLEINFO structure which describes the attributes of the service provider module referenced by *hModule*.

**Parameters**

*hModule (input)*
Handle of the service provider module.

**Return Value**

A pointer to a CSSM_HANDLEINFO data structure. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

### 3.2.3 CSSM_RegisterServices

**CSSM_RETURN CSSMAPI CSSM_RegisterServices** (const CSSM_GUID_PTR GUID,
const CSSM_REGISTRATION_INFO_PTR
FunctionTable,
CSSM_SPI_MEMORY_FUNCS_PTR
UpcallTable,
void *Reserved)

This function is used by a service provider module to register its function table with KeyWorks and to receive a memory management upcall table from KeyWorks.

**Parameters**

*GUID (input)*
A pointer to the CSSM_GUID structure containing the GUID for the calling module.

*FunctionTable (input)*
A structure containing pointers to the interface functions implemented by this module, organized by interface type.

*UpcallTable (output)*
A pointer to the CSSM_SPI_MEMORY_FUNCS structure containing the memory management function pointers to be used by this module.

*Reserved (input)*
A reserved input.

**Return Value**

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

**See Also**

CSSM_DeregisterServices

### 3.2.4 EventNotify

**CSSM_RETURN CSSMAPI EventNotify** (CSSM_MODULE_HANDLE Handle,
const CSSM_EVENT_TYPE Event,
const uint32 Param)

This function is used by KeyWorks to notify the module of certain events such as module attach and detach operations.

**Parameters**

*Handle (input)*
The handle that identifies the module to application thread pairing.

*Event (input)*
The event that is occurring. The possible events are described in Table 2.

**Table 2. Module Event Types**

| Event | Description |
|---|---|
| CSSM_EVENT_ATTACH | The application has requested an attach operation. |
| CSSM_EVENT_DETACH | The application has requested a detach operation. |
| CSSM_EVENT_INFOATTACH | An application has requested module info and KeyWorks wants to obtain the module's dynamic capabilities. The service provider module cannot assume that Initialize or Terminate has been called. |
| CSSM_EVENT_INFODETACH | KeyWorks has finished obtaining the module's dynamic capabilities. |
| CSSM_EVENT_CREATE_CONTEXT | A context has been created. |
| CSSM_EVENT_DELETE_CONTEXT | A context has been deleted. |

*Param (input)*
An event-specific parameter (see Table 3).

**Table 3. Module Event Parameters**

| Event | Parameter |
|---|---|
| CSSM_EVENT_ATTACH | None |
| CSSM_EVENT_DETACH | None |
| CSSM_EVENT_INFOATTACH | None |
| CSSM_EVENT_INFODETACH | None |
| CSSM_EVENT_CREATE_CONTEXT | Context handle |
| CSSM_EVENT_DELETE_CONTEXT | Context handle |

**Return Value**

A CSSM_OK return value signifies that the module's event-specific operations were successfully performed. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

Initialize, Terminate

### 3.2.5 FreeModuleInfo

**CSSM_RETURN CSSMAPI FreeModuleInfo** (CSSM_MODULE_HANDLE ModuleHandle,
CSSM_MODULE_INFO_PTR ModuleInfo)

This function frees the memory allocated to hold all of the info structures returned by GetModuleInfo. All substructures within the info structure are freed by this function.

**Parameters**

*ModuleHandle (input)*
The handle of the attached service provider module.

*ModuleInfo (input)*
A pointer to the CSSM_MODULE_INFO structures to be freed.

**Return Value**

This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

**See Also**

GetModuleInfo

### 3.2.6 GetModuleInfo

**CSSM_MODULE_INFO_PTR CSSMAPI GetModuleInfo**

(CSSM_MODULE_HANDLE ModuleHandle,
CSSM_SERVICE_MASK ServiceMask,
uint32 SubserviceID,
CSSM_INFO_LEVEL InfoLevel)

This function returns descriptive information about the module identified by the ModuleHandle. The information returned can include all of the capability information for each subservices, and for each of the service types implemented by the selected module. The request for information can be limited to a particular set of services, as specified by the service bit-mask. The request may be further limited to one or all of the subservices implemented in one or all of the service categories. Finally, the detail level of the information returned can be controlled by the InfoLevel input parameter. This is particularly important for the module with dynamic capabilities. InfoLevel can be used to request static attribute values only or dynamic values.

**Parameters**

*ModuleHandle (input)*
The handle of the attached service provider module.

*ServiceMask (input)*
A bit-mask specifying the module service types used to restrict the capabilities information returned by this function. An input value of zero specifies all services for the specified module.

*SubserviceID (input)*
A single subservice ID or the value CSSM_ALL_SUBSERVICES must be provided. If a subservice ID is provided the get operation is limited to the specified subservice. Note that the operation may already be limited by a service mask. If so, the subservice ID applies to all service categories selected by the service mask. If CSSM_ALL_SUBSERVICES is specified, information for all subservices (as limited by the service mask) is returned by this function.

*InfoLevel (input)*
Indicates the level of detail returned by this function. Information retrieval can be restricted as follows:

- CSSM_INFO_LEVEL_MODULE - Returns only the information contained in the cssm_moduleinfo structure.
- CSSM_INFO_LEVEL_SUBSERVICE - Returns the information returned by CSSM_INFO_LEVEL_MODULE and the information contained in the cssm_XXsubservice structure, where XX corresponds to the module type, such as cssm_tpsubservice.
- CSSM_INFO_LEVEL_STATIC_ATTR - Returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically defined for the module.
- CSSM_INFO_LEVEL_ALL_ATTR - Returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically or dynamically defined for the module. Dynamic modules, whose capabilities change over time, support a query function used by KeyWorks to interrogate the module's current capability status.

**Return Value**

A pointer to a module info structure containing a pointer to an array of zero or more service information structures. Each structure contains type information identifying the service description as representing Certificate Library services (CL), Data Storage Library (DL) services, etc. The service descriptions are subclassed into subservice descriptions that describe the attributes and capabilities of a subservice.

**See Also**

CSSM_SetModuleInfo, CSSM_FreeModuleInfo

### 3.2.7 Initialize

```
CSSM_RETURN CSSMAPI Initialize (CSSM_MODULE_HANDLE Handle,
                                uint32 VerMajor,
                                uint32 VerMinor)
```

This function checks whether the current version of the module is compatible with the input version, and performs any module-specific setup activities.

**Parameters**

*Handle (input)*
The handle that identifies the module to application thread pairing.

*VerMajor (input)*
The major version number of the module expected by the calling application.

*VerMinor (input)*
The minor version number of the module expected by the calling application.

**Return Value**

A CSSM_OK return value signifies that the current version of the module is compatible with the input version numbers, and all setup operations were successfully performed. When CSSM_FAIL is returned, either the current module is incompatible with the requested module version or an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

Terminate, EventNotify

### 3.2.8 Terminate

**CSSM_RETURN CSSMAPI Terminate** (CSSM_MODULE_HANDLE Handle)

This function performs any module-specific cleanup activities.

**Parameters**

*Handle (input)*
The handle that identifies the module to application thread pairing.

**Return Value**

A CSSM_OK return value signifies that all cleanup operations were successfully performed. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

Initialize, EventNotify

# Chapter 4. Relevant CSSM API functions

Several API functions are particularly relevant to module developers because they are used either by the application to access a module, or by a module to access IBM KeyWorks services such as the KeyWorks registry or the error-handling routines. This subset of API functions is included in this chapter for quick reference by module developers. For additional information, module developers are encouraged to reference the *IBM KeyWorks Toolkit Application Programming Interface Specification*.

## 4.1 Data Structures

### 4.1.1 Basic Data Types

```
typedef unsigned char uint8;
typedef unsigned short uint16;
typedef short sint16;
typedef unsigned int uint32;
typedef int sint32;
```

The following is used by KeyWorks data structures to represent a character string inside of a fixed-length buffer. The character string is expected to be NULL-terminated. The string size was chosen to accommodate current security standards, such as Public-Key Cryptography Standard (PKCS#11).

```
#define CSSM_MODULE_STRING_SIZE  64
typedef char CSSM_STRING [CSSM_MODULE_STRING_SIZE + 4];
```

### 4.1.2 CSSM_ALL_SUBSERVICES

This data type is used to identify that information on all of the subservices is being requested or returned.

```
#define CSSM_ALL_SUBSERVICES (-1)
```

### 4.1.3 CSSM_BOOL

This data type is used to indicate a true or false condition.

```
typedef uint32 CSSM_BOOL;
#define CSSM_TRUE 1
#define CSSM_FALSE 0
```

Definitions:

*CSSM_TRUE* - Indicates a true result or a true value.

*CSSM_FALSE* - Indicates a false result or a false value.

### 4.1.4 CSSM_CALLBACK

An application uses this data type to request that a service provider module call back into the application for certain cryptographic information.

```
typedef CSSM_DATA_PTR (CSSMAPI *CSSM_CALLBACK) (void *allocRef, uint32 ID);
```

Definitions:

*allocRef* - Memory heap reference specifying which heap to use for memory allocation.

*ID* - Input data to identify the callback.

### 4.1.5 CSSM_CRYPTO_DATA

This data structure is used to encapsulate cryptographic information, such as the passphrase to use when accessing a private key.

```
typedef struct cssm_crypto_data {
       CSSM_DATA_PTR Param;
       CSSM_CALLBACK Callback;
       uint32 CallbackID;
}CSSM_CRYPTO_DATA, *CSSM_CRYPTO_DATA_PTR
```

Definitions:

*Param* - A pointer to the parameter data and its size in bytes.

*Callback* - An optional callback routine for the service provider modules to obtain the parameter.

*CallbackID* - A tag that identifies the callback.

### 4.1.6 CSSM_DATA

The CSSM_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application via KeyWorks. Trust Policy (TP) modules and Certificate Libraries (CLs) use this structure to hold certificates and Certificate Revocation Lists (CRLs). Other service provider modules, such as Cryptographic Service Providers (CSPs), use this same structure to hold general data buffers. Data Storage Library (DL) modules use this structure to hold persistent security-related objects.

```
typedef struct cssm_data{
    uint32 Length;
    uint8 *Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definitions:

*Length* - Length of the data buffer in bytes.

*Data* - Points to the start of an arbitrary length data buffer.

### 4.1.7 CSSM_GUID

This structure designates a Globally Unique ID (GUID) that distinguishes one service provider module from another. All GUID values should be computer-generated to guarantee uniqueness. (The GUID generator in Microsoft Developer Studio and the RPC UUIDGEN/uuid_gen program can be used on a number of UNIX-based platforms.)

```
typedef struct cssm_guid{
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8  Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR
```

Definitions:

*Data1* - Specifies the first 8 hexadecimal digits of the GUID.

*Data2* - Specifies the first group of 4 hexadecimal digits of the GUID.

*Data3* - Specifies the second group of 4 hexadecimal digits of the GUID.

*Data4* - Specifies an array of 8 elements that contains the third and final group of 8 hexadecimal digits of the GUID in elements 0 and 1, and the final 12 hexadecimal digits of the GUID in elements 2 through 7.

### 4.1.8 CSSM_HANDLE

A unique identifier for an object managed by KeyWorks or by a service provider module.

```
typedef uint32 CSSM_HANDLE, *CSSM_HANDLE_PTR
```

### 4.1.9 CSSM_INFO_LEVEL

This enumerated list defines the levels of information detail that can be retrieved about the services and capabilities implemented by a particular module. Modules can implement multiple KeyWorks service types. Each service may provide one or more subservices. Modules also can have dynamically available services and features.

```
typedef enum cssm_info_level {
    CSSM_INFO_LEVEL_MODULE= 0,
                /* values from CSSM_SERVICE_INFO struct */
    CSSM_INFO_LEVEL_SUBSERVICE   = 1,
                /* values from CSSM_SERVICE_INFO and XXsubservice struct */
    CSSM_INFO_LEVEL_STATIC_ATTR  = 2,
                /* values from CSSM_SERVICE_INFO and XXsubservice and
                   all static-valued attributes of a subservice */
    CSSM_INFO_LEVEL_ALL_ATTR     = 3,
                /* values from CSSM_SERVICE_INFO and XXsubservice and
                   all attributes, static and dynamic, of a subservice */
} CSSM_INFO_LEVEL;
```

### 4.1.10 CSSM_MEMORY_FUNCS / CSSM_API_MEMORY_FUNCS

This structure is used by applications to supply memory functions for the KeyWorks and the service provider modules. The functions are used when memory needs to be allocated by the KeyWorks or service providers for returning data structures to the applications.

```
typedef struct cssm_memory_funcs {
    void *(*malloc_func) (uint32 Size, void *AllocRef);
    void  (*free_func)   (void *MemPtr, void *AllocRef);
    void *(*realloc_func)(void *MemPtr, uint32 Size, void *AllocRef);
    void *(*calloc_func) (uint32 Num, uint32 Size, void *AllocRef);
    void *AllocRef;
} CSSM_MEMORY_FUNCS, *CSSM_MEMORY_FUNCS_PTR;

typedef CSSM_MEMORY_FUNCS CSSM_API_MEMORY_FUNCS;
typedef CSSM_API_MEMORY_FUNCS *CSSM_API_MEMORY_FUNCS_PTR;
```

Definitions:

*malloc_func* - Pointer to a function that returns a void pointer to the allocated memory block of at least *Size* bytes from heap *AllocRef.*

*free_func* - Pointer to a function that deallocates a previously allocated memory block ( *MemPtr* ) from heap *AllocRef.*

*realloc_func* - Pointer to a function that returns a void pointer to the reallocated memory block (*MemPtr* ) of at least *Size* bytes from heap *AllocRef.*

*calloc_func* - Pointer to a function that returns a void pointer to an array of *Num* elements of length *Size* initialized to zero from heap *AllocRef.*

*AllocRef* - Indicates which memory heap the function operates on.

### 4.1.11 CSSM_MODULE_FLAGS

This bit-mask is used to identify characteristics of the module, such as whether or not it is threadsafe.

```
typedef uint32 CSSM_MODULE_FLAGS;

#define CSSM_MODULE_THREADSAFE  0x1  /* Module is threadsafe */
#define CSSM_MODULE_EXPORTABLE  0x2  /* Module can be exported outside the USA */
```

### 4.1.12 CSSM_MODULE_HANDLE

A unique identifier for an attached service provider module.

```
typedef uint32 CSSM_MODULE_HANDLE
```

### 4.1.13 CSSM_MODULE_INFO

This structure aggregates all service descriptions about all service types of a module implementation.

```
typedef struct cssm_moduleinfo {
    CSSM_VERSION Version;
    CSSM_VERSION CompatibleCSSMVersion;
    CSSM_STRING Description;
    CSSM_STRING Vendor;
    CSSM_MODULE_FLAGS Flags;
    CSSM_SERVICE_MASK ServiceMask;
    uint32 NumberOfServices;
    CSSM_SERVICE_INFO_PTR ServiceList;
    void *Reserved;
} CSSM_MODULE_INFO, *CSSM_MODULE_INFO_PTR;
```

Definitions:

*Version*- The major and minor version numbers of this service provider module.

*CompatibleCSSMVersion* - The version of KeyWorks that this module was written to.

*Description* - A text description of this module and its functionality.

*Vendor*- The name and description of the module vendor.

*Flags*- Characteristics of this module, such as whether or not it is threadsafe.

*ServiceMask* - A bit-mask identifying the types of services available in this module.

*NumberOfServices* - The number of services for which information is provided. Multiple descriptions (as subservices) can be provided for a single service category.

*ServiceList* - An array of pointers to the service information structures. This array contains *NumberOfServices* entries.

*Reserved* - This field is reserved for future use. It should always be set to NULL.

### 4.1.14 CSSM_NOTIFY_CALLBACK

An application uses this data type to request that a service provider module call back into the application to notify it of certain events.

```
typedef CSSM_RETURN (CSSMAPI *CSSM_NOTIFY_CALLBACK)(CSSM_MODULE_HANDLE
                                                     ModuleHandle,
                                                     uint32 Application,
                                                     uint32 Reason,
                                                     uint32 Param);
```

Definitions:

*ModuleHandle* - The handle of the attached service provider module.

*Application* - Input data to identify the callback.

*Reason* - The reason for the notification (see Table 4).

**Table 4. Notification Reasons**

| Reason | Description |
|---|---|
| CSSM_NOTIFY_SURRENDER | The service provider module is temporarily surrendering control of the process. |
| CSSM_NOTIFY_COMPLETE | An asynchronous operation has completed. |
| CSSM_NOTIFY_DEVICE_REMOVED | A device, such as a token, has been removed. |
| CSSM_NOTIFY_DEVICE_INSERTED | A device, such as a token, has been inserted. |

*Param* - Any additional information about the event.

### 4.1.15 CSSM_RETURN

This data type is used to indicate whether a function was successful.

```
typedef enum cssm_return {
       CSSM_OK = 0,
       CSSM_FAIL = -1
} CSSM_RETURN
```

Definitions:

*CSSM_OK* - Indicates operation was successful

*CSSM_FAIL* - Indicates operation was unsuccessful.

### 4.1.16 CSSM_SERVICE_FLAGS

This bit-mask is used to identify characteristics of the service, such as whether it contains any embedded products.

```
typedef uint32 CSSM_SERVICE_FLAGS

#define CSSM_SERVICE_ISWRAPPEDPRODUCT 0x1
                                  /* On  = Contains one or more embedded
products
                                     Off = Contains no embedded products */
```

### 4.1.17 CSSM_SERVICE_INFO

This structure holds a description of a module service. The service described is of the KeyWorks service type specified by the module type.

```
typedef struct cssm_serviceinfo {
    CSSM_STRING Description;
    CSSM_SERVICE_TYPE Type;
    CSSM_SERVICE_FLAGS Flags;
    uint32 NumberOfSubServices;
    union {
        void *SubServiceList;
        CSSM_CSPSUBSERVICE_PTR CspSubServiceList;
        CSSM_DLSUBSERVICE_PTR  DlSubServiceList;
        CSSM_CLSUBSERVICE_PTR  ClSubServiceList;
        CSSM_TPSUBSERVICE_PTR  TpSubServiceList;
        CSSM_KRSUBSERVICE_PTR  KrSubServiceList;
    };
    void *Reserved;
} CSSM_SERVICE_INFO, *CSSM_SERVICE_INFO_PTR;
```

Definitions:

*Description*- A text description of the service.

*Type* - Specifies exactly one type of service structure, such as CSSM_SERVICE_CSP, CSSM_SERVICE_CL, etc.

*Flags*- Characteristics of this service, such as whether it contains any embedded products.

*NumberOfSubServices* - The number of elements in the module *SubServiceList*.

*SubServiceList* - A list of descriptions of the encapsulated subservices (not of the basic service types).

*CspSubServiceList* - A list of descriptions of the encapsulated CSP subservices.

*DlSubServiceList* - A list of descriptions of the encapsulated DL subservices.

*ClSubServiceList* - A list of descriptions of the encapsulated CL subservices.

*TpSubServiceList* - A list of descriptions of the encapsulated TP subservices.

*KrSubServiceList* - A list of descriptions of the encapsulated key recovery subservices.

*Reserved* - This field is reserved for future use. It should always be set to NULL.

### 4.1.18 CSSM_SERVICE_MASK

This defines a bit-mask of all the types of KeyWorks services a single module can implement.

```
typedef uint32 CSSM_SERVICE_MASK;

#define CSSM_SERVICE_CSSM  0x1
#define CSSM_SERVICE_CSP   0x2
#define CSSM_SERVICE_DL    0x4
#define CSSM_SERVICE_CL    0x8
#define CSSM_SERVICE_TP    0x10
#define CSSM_SERVICE_KR    0x20
#define CSSM_SERVICE_LAST  CSSM_SERVICE_TP
```

### 4.1.19 CSSM_SERVICE_TYPE

This data type is used to identify a single service from the CSSM_SERVICE_MASK options defined above.

```
typedef CSSM_SERVICE_MASK CSSM_SERVICE_TYPE
```

### 4.1.20 CSSM_SPI_FUNC_TBL

This structure is used by service provider modules to reference an application's memory management functions. The functions are used when a service provider module needs to allocate memory for returning data structures to the application, or needs to deallocate memory for a data structure that is passed to it from an application.

```
typedef struct cssm_spi_func_tbl {
    void *(*malloc_func) (CSSM_HANDLE AddInHandle, uint32 Size);
    void  (*free_func)   (CSSM_HANDLE AddInHandle, void *MemPtr);
    void *(*realloc_func)(CSSM_HANDLE AddInHandle, void *MemPtr, uint32 Size);
    void *(*calloc_func) (CSSM_HANDLE AddInHandle, uint32 Num, uint32 Size);
} CSSM_SPI_MEMORY_FUNCS, *CSSM_SPI_MEMORY_FUNCS_PTR;
```

Definitions:

*malloc_func* - Pointer to a function that returns a void pointer to the allocated memory block of at least *Size* bytes from the heap of the application associated with *AddInHandle.*

*free_func* - Pointer to a function that deallocates a previously allocated memory block ( *MemPtr* ) from the heap of the application associated with *AddInHandle.*

*realloc_func* - Pointer to a function that returns a void pointer to the reallocated memory block (*MemPtr* ) of at least *Size* bytes from the heap of the application associated with *AddInHandle.*

*calloc_func* - Pointer to function that returns a void pointer to an array of *Num* elements of length *Size* initialized to zero from the heap of the application associated with *AddInHandle.*

### 4.1.21 CSSM_USER_AUTHENTICATION

This structure holds the user's credentials for authenticating to a module. The type of credentials required is defined by the module and specified as a CSSM_USER_AUTHENTICATION_MECHANISM.

```
typedef struct cssm_user_authentication {
    CSSM_DATA_PTR Credential;
    CSSM_CRYPTO_DATA_PTR MoreAuthenticationData;
} CSSM_USER_AUTHENTICATION, *CSSM_USER_AUTHENTICATION_PTR;
```

Definitions:

*Credential* - A certificate, a shared secret, a magic token, or whatever is required by a service provider module for user authentication. The required credential type is specified as a CSSM_USER_AUTHENTICATION_MECHANISM.

*MoreAuthenticationData* - A passphrase or other data that can be provided as immediate data within this structure or via a callback function to the user/caller.

### 4.1.22 CSSM_USER_AUTHENTICATION_MECHANISM

The enumerated list of CSSM_User_Authentication_Mechanism defines different methods a service provider module can require when authenticating a caller. The module specifies which mechanism the caller must use for each subservice type provided by the module. KeyWorks-defined authentication methods include password-based authentication, a login sequence, or a certificate and passphrase. It is anticipated that new mechanisms will be added to this list as required.

```
typedef enum cssm_user_authentication_mechanism {
    CSSM_AUTHENTICATION_NONE = 0,
    CSSM_AUTHENTICATION_CUSTOM = 1,
    CSSM_AUTHENTICATION_PASSWORD = 2,
    CSSM_AUTHENTICATION_USERID_AND_PASSWORD = 3,
    CSSM_AUTHENTICATION_CERTIFICATE_AND_PASSPHRASE = 4,
    CSSM_AUTHENTICATION_LOGIN_AND_WRAP = 5,
} CSSM_USER_AUTHENTICATION_MECHANISM;
```

### 4.1.23 CSSM_VERSION

This structure is used to represent the version of KeyWorks components.

```
typedef struct cssm_version {
    uint32 Major;
    uint32 Minor;
} CSSM_VERSION, *CSSM_VERSION_PTR;
```

Definitions:

*Major* - The major version number of the component.

*Minor* - The minor version number of the component.

## 4.2 Function Definitions

### 4.2.1 CSSM_ModuleAttach

```
CSSM_MODULE_HANDLE CSSMAPI CSSM_ModuleAttach
                    (const CSSM_GUID_PTR GUID,
                     const CSSM_VERSION_PTR Version,
                     const CSSM_API_MEMORY_FUNCS_PTR MemoryFuncs,
                     uint32 SubserviceID,
                     uint32 SubserviceFlags,
                     uint32 Application,
                     const CSSM_NOTIFY_CALLBACK Notification,
                     const void * Reserved)
```

This function attaches the service provider module and verifies that the version of the module expected by the application is compatible with the version on the system. The module can implement subservices (as described in the IBM KeyWorks Toolkit SPI documentation in Section 1.3). The caller can specify a specific subservice provided by the module. Subservice flags may be required to set parameters for the service.

**Parameters**

*GUID (input)*
A pointer to the CSSM_GUID structure containing the GUID for the CSP module.

*Version (input)*
The major and minor version number of the service provider module that the application is compatible with.

*MemoryFuncs (input)*
A structure containing pointers to the memory routines.

*SubserviceID (input)*
The number of a subservice provided by the module. This value should always be taken from the CSSM_MODULE_INFO structure to ensure that a compatible identifier is used. (Service provider modules that implement only one service can use zero as the subservice identifier.)

*SubserviceFlags (input)*
Bit-mask of service options defined by a particular subservice of the module. Legal values are described in module-specific documentation. A default set of flags is specified in the CSSM_MODULE_INFO structure for use by the caller.

*Application (input/optional)*
Nonce passed to the application when its callback is invoked, allowing the application to determine the proper context of operation.

*Notification (input/optional)*
Callback provided by the application that is used by the service provider module to notify the application of certain events. For example, a CSP may use this callback in the following situations: a parallel operation completes, a token running in serial mode surrenders control to the application, or the token is removed (hardware-specific).

*Reserved (input)*
A reserved input.

**Return Value**

A handle is returned for the attached service provider module. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

CSSM_ModuleDetach

### 4.2.2 CSSM_ModuleDetach

**CSSM_RETURN CSSMAPI CSSM_ModuleDetach** (CSSM_MODULE_HANDLE ModuleHandle)

This function detaches the application from the service provider module.

**Parameters**

*ModuleHandle (input)*
The handle that describes the service provider module.

**Return Value**

A CSSM_OK return value signifies that the application has been detached from the module.
If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

CSSM_ModuleAttach

### 4.2.3 CSSM_FreeModuleInfo

**CSSM_RETURN CSSMAPI CSSM_FreeModuleInfo** (CSSM_MODULE_INFO_PTR ModuleInfo)

This function frees the memory allocated by CSSM_GetModuleInfo to hold the module info structures. All substructures within the info structure are freed by this function.

**Parameters**

*ModuleInfo (input)*
A pointer to the CSSM_MODULE_INFO structures to be freed.

**Return Value**

This function returns CSSM_OK if successful, and returns CSSM_FAIL if an error has occurred. Use CSSM_GetError to determine the exact error.

**See Also**

CSSM_GetModuleInfo, CSSM_SetModuleInfo

### 4.2.4 CSSM_GetCSSMRegistryPath

**CSSM_DATA_PTR CSSMAPI CSSM_GetCSSMRegistryPath** (void)

This function gets the directory path of the KeyWorks registry.

**Parameters**

*None*

**Return Value**

A pointer to a CSSM_DATA structure containing the registry path information, or a NULL if an error occurred in getting the information. Use CSSM_GetError to determine the exact error.

### 4.2.5 CSSM_GetGUIDUsage

**CSSM_SERVICE_MASK CSSMAPI CSSM_GetGUIDUsage**
(const CSSM_GUID_PTR ModuleGUID)

Returns a bit-mask describing the KeyWorks function categories of service provided by the module identified by GUID.

**Parameters**

*ModuleGUID (input)*
GUID for the module of interest.

**Return Value**

A CSSM_SERVICE_MASK from the info structure describing the services provided by the module referenced by the GUID.

**See Also**

CSSM_GetHandleUsage

### 4.2.6 CSSM_GetHandleUsage

**CSSM_SERVICE_MASK CSSMAPI CSSM_GetHandleUsage**
(CSSM_HANDLE ModuleHandle)

Returns a bit-mask describing the KeyWorks function categories of services provided by the module, and identified by the specified handle for an attached module.

**Parameters**

*ModuleHandle (input)*
Handle of the module for which information should be returned.

**Return Value**

A CSSM_SERVICE_MASK from the info structure describing the services provided by the module referenced by the handle.

**See Also**

CSSM_GetGUIDUsage

### 4.2.7 CSSM_GetModuleGUIDFromHandle

**CSSM_GUID_PTR CSSMAPI CSSM_GetModuleGUIDFromHandle**
(CSSM_HANDLE ModuleHandle)

This function determines the GUID associated with a specific module handle.

**Parameters**

*ModuleHandle (input)*
The handle that describes the service provider module.

**Return Value**

A CSSM_GUID_PTR to a data structure containing the GUID associated with *ModuleHandle*. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

### 4.2.8 CSSM_GetModuleInfo

**CSSM_MODULE_INFO_PTR CSSMAPI CSSM_GetModuleInfo**

```
(const CSSM_GUID_PTR ModuleGUID,
 CSSM_SERVICE_MASK ServiceMask,
 uint32 SubserviceID,
 CSSM_INFO_LEVEL InfoLevel)
```

This function returns descriptive information about the module identified by the ModuleGUID. The information returned can include all of the capability information, information for each subservice, or information for each of the service types implemented by the selected module. The request for information can be limited to a particular set of services, as specified by the ServiceMask bit-mask. The request may be further limited to one or all of the subservices implemented in one or all of the service categories. Finally, the detail level of the information returned can be controlled by the InfoLevel input parameter. This is particularly important for module with dynamic capabilities. InfoLevel can be used to request static attribute values only or dynamic values.

**Parameters**

*ModuleGUID (input)*
A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

*ServiceMask (input)*
A bit-mask specifying the module service types used to restrict the capabilities information returned by this function. An input value of zero specifies all services for the specified module.

*SubserviceID (input)*
A single subservice ID or the value CSSM_ALL_SUBSERVICES must be provided. If a subservice ID is provided, the get operation is limited to the specified subservice. Note that the operation may already be limited by a service mask. If so, the subservice ID applies to all service categories selected by the service mask. If CSSM_ALL_SUBSERVICES is specified, information for all subservices (as limited by the service mask) is returned by this function.

*InfoLevel (input)*
Indicates the level of detail returned by this function. Information retrieval can be restricted as follows. Note that not all service provider modules support all of the following values.

- CSSM_INFO_LEVEL_MODULE - Returns only the information contained in the cssm_moduleinfo structure.

- CSSM_INFO_LEVEL_SUBSERVICE - Returns the information returned by CSSM_INFO_LEVEL_MODULE and the information contained in the cssm_XXsubservice structure, where XX corresponds to the module type, such as cssm_tpsubservice.

- CSSM_INFO_LEVEL_STATIC_ATTR - Returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically defined for the module.

- CSSM_INFO_LEVEL_ALL_ATTR - Returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically or dynamically defined for the module. Dynamic modules, whose capabilities change over time, support a query function used by KeyWorks to interrogate the module's current capability status.

**Return Value**

A CSSM_MODULE_INFO_PTR to an array of one or more info structures. Each structure contains type information identifying the capability description as representing CL capabilities, DL capabilities, etc. The capability descriptions can also be subclassed into subservices.

**See Also**

CSSM_GetModuleInfo, CSSM_FreeModuleInfo

### 4.2.9 CSSM_GetModuleLocation

**CSSM_DATA_PTR CSSMAPI CSSM_GetModuleLocation** (const CSSM_GUID_PTR GUID)

This function returns the directory path of the service provider module specified by the GUID input parameter.

**Parameters**

*GUID (input)*
A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

**Return Value**

A pointer to a CSSM_DATA data structure containing the directory path of the module associated with GUID. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

### 4.2.10 CSSM_ListModules

**CSSM_LIST_PTR CSSMAPI CSSM_ListModules** (CSSM_SERVICE_MASK ServiceMask,
CSSM_BOOL MatchAll)

This function returns a list containing the GUID/name pair for each of the currently installed service provider modules that provide services in any of the KeyWorks functional categories selected in the service mask.

**Parameters**

*ServiceMask (input)*
A bit-mask selecting the KeyWorks functional categories. This information can be used to select information about potential service provider modules.

*MatchAll (input)*
A Boolean value defining how the multiple bits in the service mask are interpreted. CSSM_TRUE means the service modules selected must match all service areas specified by the service mask. CSSM_FALSE means the service module selected must specify one or more of the service areas specified by the service mask.

**Return Value**

A pointer to the CSSM_LIST structure containing the GUID/name pair for each of the modules. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

CSSM_GetModuleInfo, CSSM_FreeModuleInfo

### 4.2.11 CSSM_ModuleInstall

**CSSM_RETURN CSSMAPI CSSM_ModuleInstall**

```
(const char *ModuleName,
 const char *ModuleFileName,
 const char *ModulePathName,
 const CSSM_GUID_PTR GUID,
 const CSSM_MODULE_INFO_PTR ModuleDescription,
 const void * Reserved1,
 const CSSM_DATA_PTR Reserved2)
```

This function registers the module with KeyWorks. KeyWorks adds the module's descriptive information to its persistent registry. This makes the service module available for use on the local system. The function accepts as input the name and unique identifier for the module, the location executable code for the module, and a digitally signed list of capabilities supported by the module. The module name and description are added to the KeyWorks registry, making the module available for use by applications.

**Parameters**

*ModuleName (input)*
The name of the module.

*ModuleFileName (input)*
The name of the file that implements the module.

*ModulePathName (input)*
The path to the file that implements the module.

*GUID (input)*
A pointer to the CSSM_GUID structure containing the GUID for the module.

*ModuleDescription (input)*
A pointer to the CSSM_MODULE_INFO structure containing a description of the module.

*Reserved1 (input)*
Reserve data for the function.

*Reserved2 (input)*
Reserve data for the function.

**Return Value**

A CSSM_OK return value signifies that information has been updated. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

CSSM_ModuleUninstall

### 4.2.12 CSSM_SetModuleInfo

```
CSSM_RETURN CSSMAPI CSSM_SetModuleInfo
        (const CSSM_GUID_PTR ModuleGUID,
         const CSSM_MODULE_INFO_PTR ModuleInfo)
```

This function replaces all of the currently registered descriptive information about the module identified by GUID with the new specified information. CSSM_SetModuleInfo replaces all information for all service categories and all subservices.

To retain any of the module information, use the CSSM_GetModuleInfo function to retrieve the current module information from the KeyWorks registry, make a private copy, and then use the CSSM_SetModuleInfo function to update the KeyWorks registry.

This function should be used to incrementally update descriptive information that is unspecified at installation time.

**Parameters**

*ModuleGUID (input)*
A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

*ModuleInfo (input)*
A pointer to the complete structured set of descriptive information about the module.

**Return Value**

A CSSM_OK return value signifies that the application has been detached from the service provider module. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

CSSM_GetModuleInfo, CSSM_FreeModuleInfo

### 4.2.13 CSSM_ModuleUninstall

**CSSM_RETURN CSSMAPI CSSM_ModuleUninstall** (const CSSM_GUID_PTR GUID)

This function deletes the persistent KeyWorks internal information about the module and removes it from the name space of available modules in the KeyWorks system.

**Parameters**

*GUID (input)*

A pointer to the CSSM_GUID structure containing the GUID for the module.

**Return Value**

A CSSM_OK return value means the module has been successfully uninstalled. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

**See Also**

CSSM_ModuleInstall

### 4.2.14 CSSM_SetModuleInfo

```
CSSM_RETURN CSSMAPI CSSM_SetModuleInfo
        (const CSSM_GUID_PTR ModuleGUID,
         const CSSM_MODULE_INFO_PTR ModuleInfo)
```

This function replaces all of the currently registered descriptive information about the module, identified by the ModuleGUID, with the newly specified information. The operation is a total replacement of all information for all service categories and all subservices.

If the caller wants to retain any of the information registered prior to execution of this call, the caller must use the CSSM_GetModuleInfo function to retrieve the current information, update a private copy, and then use the CSSM_SetModuleInfo function to return the updated copy back to the KeyWorks registry.

This function should be used to incrementally update descriptive information that is unspecified at installation time.

**Parameters**

*ModuleGUID (input)*
A pointer to the CSSM_GUID structure containing the GUID for the service provider module.

*ModuleInfo (input)*
A pointer to the complete structured set of descriptive information about the module.

**Return Value**

A CSSM_RETURN value indicating pass or fail. CSSM_OK indicates success; otherwise use CSSM_GetError to determine the type of error that has occurred.

**See Also**

CSSM_GetModuleInfo, CSSM_FreeModuleInfo

### 4.2.15 CSSM_FreeModuleInfo

**CSSM_RETURN CSSMAPI CSSM_FreeModuleInfo** (CSSM_MODULE_INFO_PTR ModuleInfo)

This function frees the memory allocated to hold all of the info structures returned by CSSM_GetModuleInfo. All substructures within the info structure are freed by this function.

**Parameters**

*ModuleInfo (input)*
A pointer to the CSSM_MODULE_INFO structures to be freed.

**Return Value**

A KeyWorks return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

**See Also**

CSSM_GetModuleInfo, CSSM_SetModuleInfo

### 4.2.16 CSSM_GetError

**CSSM_ERROR_PTR CSSMAPI CSSM_GetError** (void)

This function returns the current error information.

**Parameters**

*None*

**Return Value**

Returns the current error information. If there is currently no valid error, the error number will be CSSM_OK. A NULL pointer indicates that the CSSM_InitError was not called by the KeyWorks Core or that a call to CSSM_DestoryError has been made by the KeyWorks Core. No error information is available.

**See Also**

CSSM_ClearError, CSSM_SetError

### 4.2.17 CSSM_SetError

**CSSM_RETURN CSSMAPI CSSM_SetError** (CSSM_GUID_PTR guid,
uint32 error_number)

This function sets the current error information to error_number and guid.

**Parameters**

*guid (input)*
Pointer to the GUID of the service provider module.

*error_number (input)*
An error number. It should fall within one of the valid KeyWorks, CL, TP, DL, or CSP error ranges.

**Return Value**

CSSM_OK if error was successfully set. A return value of CSSM_FAIL indicates that the error number passed is not within a valid range, the GUID passed is invalid, CSSM_InitError was not called by the KeyWorks Core, or CSSM_DestroyError has been called by the KeyWorks Core. No error information is available.

**See Also**

CSSM_ClearError, CSSM_GetError

### 4.2.18 CSSM_ClearError

**void CSSMAPI CSSM_ClearError** (void)

This function sets the current error value to CSSM_OK. This can be called if the current error value has been handled and therefore is no longer a valid error.

**Parameters**

*None*

**See Also**

CSSM_SetError, CSSM_GetError

# Appendix A. KeyWorks Errors

## A.1 Service Provider Module Structure and Administration Errors

The following table provides Service Provider Module Structure and Administration errors.

**Table 5. Invalid Errors**

| Error Code | Error Name |
|---|---|
| 10501 | CSSM_INVALID_GUID |
| 10301 | CSSM_INVALID_POINTER |
| 10341 | CSSM_INVALID_SUBSERVICED |
| 10342 | CSSM_INVALID_INFO_LEVEL |
| 10303 | CSSM_MEMORY_ERROR |

# Appendix B. List of Acronyms

| | |
|---|---|
| API | Application Programming Interface |
| CA | Certificate Authority |
| CL | Certificate Library |
| CRL | Certificate Revocation List |
| CSP | Cryptographic Service Provider |
| DL | Data Storage Library |
| DLL | Dynamically Linked Library |
| GUID | Globally Unique ID |
| ISV | Independent Software Vendor |
| KRF | Key Recovery Field |
| KRSP | Key Recovery Service Provider |
| ODBC | Open Database Connectivity |
| PKCS | Public-Key Cryptographic Standard |
| RNG | Random Number Generation |
| SET | Secure Electronic Transaction |
| SPI | Service Provider Interface |
| TP | Trust Policy |

# Appendix C. Glossary

| | |
|---|---|
| Asymmetric algorithms | Cryptographic algorithms, where one key is used to encrypt and a second key is used to decrypt. They are often called public-key algorithms. One key is called the public key, and the other is called the private key or secret key. RSA (Rivest-Shamir-Adelman) is the most commonly used public-key algorithm. It can be used for encryption and for signing. |
| Authentication Information | Information that is verified for authentication. For example, a Key Recovery Officer (KRO) selects a password which will be used for authentication with the Key Recovery Coordinator (KRC). A KRO operator who has identification information must search the Authentication Information (AI) database to locate an AI value that corresponds to the individual who generated the information. |
| Certificate | See Digital certificate. |
| Certificate Authority | An entity that guarantees or sponsors a certificate. For example, a credit card company signs a cardholder's certificate to assure that the cardholder is who he or she claims to be. The credit card company is a Certificate Authority (CA). CAs issue, verify, and revoke certificates. |
| Certificate chain | The hierarchical chain of all the other certificates used to sign the current certificate. This includes the CA who signs the certificate, the CA who signed that CA's certificate, and so on. There is no limit to the depth of the certificate chain. |
| Certificate signing | The CA can sign certificates it issues or co-sign certificates issued by another CA. In a general signing model, an object signs an arbitrary set of one or more objects. Hence, any number of signers can attest to an arbitrary set of objects. The arbitrary objects could be, for example, pieces of a document for libraries of executable code. |
| Certificate validity date | A start date and a stop date for the validity of the certificate. If a certificate expires, the CA may issue a new certificate. |
| Cryptographic algorithm | A method or defined mathematical process for implementing a cryptography operation. A cryptographic algorithm may specify the procedure for encrypting and decrypting a byte stream, digitally signing an object, computing the hash of an object, generating a random number, etc. IBM KeyWorks accommodates Data Encryption Standard (DES), RC2, RC4, International Data Encryption Algorithm (IDEA), and other encryption algorithms. |
| Cryptographic Service Provider | Cryptographic Service Providers (CSPs) are modules that provide secure key storage and cryptographic functions. The modules may be software only or hardware with software drivers. The cryptographic functions provided may include:<br>• Bulk encryption and decryption<br>• Digital signing |

- Cryptographic hash
- Random number generation
- Key exchange

Cryptography — The science for keeping data secure. Cryptography provides the ability to store information or to communicate between parties in such a way that prevents other non-involved parties from understanding the stored information or accessing and understanding the communication. The encryption process takes understandable text and transforms it into an unintelligible piece of data (called ciphertext); the decryption process restores the understandable text from the unintelligible data. Both involve a mathematical formula or algorithm and a secret sequence of data called a key. Cryptographic services provide confidentiality (keeping data secret), integrity (preventing data from being modified), authentication (proving the identity of a resource or a user), and non-repudiation (providing proof that a message or transaction was send and/or received).

There are two types of cryptography:

- In shared/secret key (symmetric) cryptography there is only one key that is a shared secret between the two communicating parties. The same key is used for encryption and decryption.
- In public key (asymmetric) cryptography different keys are used for encryption and decryption. A party has two keys: a public key and a private key. The two keys are mathematically related, but it is virtually impossible to derive the private key from the public key. A message that is encrypted with someone's public key (obtained from some public directory) can only be decrypted with the associated private key. Alternately, the private key can be used to "sign" a document; the public key can be used as verification of the source of the document.

Cryptoki — Short for cryptographic token interface. See Token.

Data Encryption Standard — In computer security, the National Institute of Standards and Technology (NIST) Data Encryption Standard (DES), adopted by the U.S. Government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data encryption algorithm.

Digital certificate — The binding of some identification to a public key in a particular domain, as attested to directly or indirectly by the digital signature of the owner of that domain. A digital certificate is an unforgettable credential in cyberspace. The certificate is issued by a trusted authority, covered by that party’s digital signature. The certificate may attest to the certificate holder’s identity, or may authorize certain actions by the certificate holder. A certificate may include multiple signatures and may attest to multiple objects or multiple actions.

Digital signature — A data block that was created by applying a cryptographic signing algorithm to some other data using a secret key. Digital signatures may be used to:

- Authenticate the source of a message, data, or document

- Verify that the contents of a message has not been modified since it was signed by the sender
- Verify that a public key belongs to a particular person

Typical digital signing algorithms include MD5 with RSA encryption, and DSS, the proposed Digital Signature Standard defined as part of the U.S. Government Capstone project.

Enterprise

A company or individual who is authorized to submit on-line requests to the Key Recovery Officer (KRO). In the enterprise key recovery scenario, a process at the enterprise called the KRO is responsible for preparing key recovery requests and communicating them to the KRC. The KRO, acting on behalf of an enterprise or individual, sends an on-line request to the Key Recovery Coordinator (KRC) to recover a key from a Key Recovery Block (KRB).

Graphical User Interface

A type of display format that enables the user to choose commands, start programs, and see lists of files and other options by pointing to pictorial representations (icons) and lists of menu items on the screen. Graphical User Interfaces (GUIs) are used by the Microsoft Windows program for IBM-compatible microcomputers and by other systems.

Hash algorithm

A cryptographic algorithm used to hash a variable-size input stream into a unique, fixed-sized output value. Hashing is typically used in digital signing algorithms. Example hash algorithms include MD and MD2 from RSA Data Security. MD5, also from RSA Data Security, hashes a variable-size input stream into a 128-bit output value. SHA, a Secure Hash Algorithm published by the U.S. Government, produces a 160-bit hash value from a variable-size input stream.

IBM KeyWorks Architecture

A set of layered security services that address communications and data security problems in the emerging PC business space.

IBM KeyWorks Framework

The IBM KeyWorks Framework defines five key service components:

- Cryptographic Module Manager
- Key Recovery Module Manager
- Trust Policy Module Manager
- Certificate Library Module Manager
- Data Storage Library Module Manager

IBM KeyWorks binds together all the security services required by PC applications. In particular, it facilitates linking digital certificates to cryptographic actions and trust protocols.

Key Escrow

The storing of a key (or parts of a key) with a trusted party or trusted parties in case of loss or destruction of the key.

| | |
|---|---|
| Key Recovery Agent | The Key Recovery Agent (KRA) acts as the back end for a key recovery operation. The KRA can only be accessed through an on-line communication protocol via the Key Recovery Coordinator (KRC). KRAs are considered outside parties involved in the key recovery process; they are analogous to the neighbors who each hold one digit of the combination of the lock box containing the key. The authorized parties (i.e., enterprise or law enforcement) have the freedom to choose the number of specific KRAs that they want to use. The authorized party requests that each KRA decrypt its section of the Key Recovery Fields (KRFs) that is associated with the transmission. Then those pieces of information are used in the process that derives the session key. The KRA will only be able to recover a portion of the key, and reading the original message will require searching the remaining key space in order to find the key that will decrypt the message. The number of KRAs on each end of the communication does not have to be equal. |
| Key Recovery Block | The Key Recovery Block (KRB) is a piece of encrypted information that is contained within a block. The KRS components (i.e., KRO, KRC, KRA) work collectively to recover a session key from a provided KRB. In the enterprise scenario, the KRO has both the KRB and the credentials that authenticate it to receive the recovered key. This information will be transmitted over the network to the KRC. In the law enforcement scenario, the KRB is presented on a 3.5-inch diskette, and the credentials are in the physical form of a legal warrant. This warrant will specify any information available to the law enforcement agents which can be used to tie the warrant to the identity of the user for whom KRBs were generated (i.e., username, hostname, IP address). The KRC has the ability to check credentials and derive the original encryption key from the KRB with the help of its KRAs. |
| Key Recovery Coordinator | The Key Recovery Coordinator (KRC) acts as the front end for the key recovery operation. The KRO, acting on behalf of an enterprise or individual, sends an on-line request to the KRC to recover a key from a KRB. The KRC receives the on-line request and services it by interacting with the appropriate set of KRAs as specified within the KRB. The recovered key is then sent back to the KRO by the KRC using an on-line protocol. The KRC consists of one main application which, when started, behaves as a server process. The system, which serves as the KRC, may be configured to start the KRC application as part of system services; alternatively, the KRC operator can start up the KRC application manually. The KRC application performs the following operations:<br>• Listens for on-line recovery requests from KRO<br>• Can be used to launch an embedded application that allows manual key recovery for law enforcement<br>• Monitors and displays the status of the recovery requests being serviced |
| Key Recovery Field | A Key Recovery Field (KRF) is a block of data that is created from a symmetric key and key recovery profile information. The Key Recovery Service Provider (KRSP) is invoked from the IBM KeyWorks framework to create KRFs. There are two major pieces of the KRFs: block 1 contains information that is unrelated to the session key of the transmitted message, and encrypted with the public keys of the selected key recovery agents; block 2 contains information that is related to the session key of the transmission. The KRSP generates the |

KRFs for the session key. This information is *not* the key or any portion of the key, but is information that can be used to recover the key. The KRSP has access to location-unique jurisdiction policy information that controls and modifies some of the steps in the generation of the KRFs. Only once the KRFs are generated, and both the client and server sides have access to them, can the encrypted message flow begin. KRFs are generated so that they can be used by a KRA to recover the original symmetric key, either because the user who generated the message has lost the key, or at the warranted request of law enforcement agents.

Key Recovery Module Manager

The Key Recovery Module Manager enables key recovery for cryptographic services obtained through IBM KeyWorks. It mediates all cryptographic services provided by the KeyWorks and applies the appropriate key recovery policy on all such operations. The Key Recovery Module Manager contains a Key Recovery Policy Table (KRPT) that defines the applicable key recovery policy for all cryptographic products. The Key Recovery Module Manager routes the KR-API function calls made by an application to the appropriate KR-SPI functions. The Key Recovery Module Manager also enforces the key recovery policy on all cryptographic operations that are obtained through the KeyWorks. It maintains key recovery state in the form of key recovery contexts.

Key Recovery Officer

An entity called the Key Recovery Officer (KRO) is the focal point of the key recovery process. In the enterprise key recovery scenario, the KRO is responsible for preparing key recovery requests and communicating them to the KRC. The KRO has both the KRB and the credentials that authenticate it to receive the recovered key. The KRO is the entity that acts on behalf of an enterprise to initiate a key recovery request operation. An employee within an enterprise who desires key recovery will send a request to the KRO with the KRB that is to be recovered. The actual key recovery phase begins when the KRO operator uses the KRO application to initiate a key recovery request to the appropriate KRC. At this time, the operator selects a KRB to be sent for recovery, enters the Authentication Information (AI) information that can be used to authenticate the request to the KRC, and submits the request.

Key Recovery Policy

Key recovery policies are mandatory policies that are typically derived from jurisdiction-based regulations on the use of cryptographic products for data confidentiality. Often, the jurisdictions for key recovery policies coincide with the political boundaries of countries in order to serve the law enforcement and intelligence needs of these political jurisdictions. Political jurisdictions may choose to define key recovery policies for cryptographic products based on export, import, or use controls. Enterprises may define internal and external jurisdictions, and may mandate key recovery policies on the cryptographic products within their own jurisdictions.

Key recovery policies come in two flavors: *key recovery enablement policies* and *key recovery interoperability policies.* Key recovery enablement policies specify the exact cryptographic protocol suites (e.g., algorithms, modes, key lengths, etc.) and perhaps usage scenarios, where key recovery enablement is mandated. Furthermore, these policies may also define the number of bits of the cryptographic key that may be left out of the key recovery enablement operation; this is typically referred to as the *workfactor*. Key recovery interoperability policies specify to what degree a key recovery enabled cryptographic product is allowed to interoperate with other cryptographic products.

Key Recovery Server

The Key Recovery Server (KRS) consists of three major entities: Key Recovery Coordinator (KRC), Key Recovery Agent (KRA), and Key Recovery Officer (KRO). The KRS is intended to be used by enterprise employees and security personnel, law enforcement personnel, and KRSF personnel. The KRS interacts with one or more local or remote KRAs to reconstruct the secret key that can be used to decrypt the ciphertext.

Key Recovery Server Facility

The Key Recovery Server Facility (KRSF) is a facility room that houses the KRS component facilities ensuring they operate within a secure environment that is highly resistant to penetration and compromise. Several physical and administrative security procedures must be followed at the KRSF such as a combination keyed lock, limited personnel, standalone system, operating system with security features (Microsoft NT Workstation 4.0), NTFS (Windows NT Filesystem), and account and auditing policies.

Key Recovery Service Provider

Key Recovery Service Providers (KRSPs) are modules that provide key recovery enablement functions. The cryptographic functions provided may include:

- Key recovery field generation
- Key recovery field processing

Law Enforcement

A type of scenario where key recovery is mandated by the jurisdictional law enforcement authorities in the interest of national security and law enforcement. In the law enforcement scenario, the KRB is presented on a 3.5-inch diskette, and the credentials are in the physical form of a legal warrant. This warrant will specify any information available to the law enforcement agents which can be used to tie the warrant to the identity of the user for whom KRBs were generated (i.e., username, hostname, IP address).

Leaf certificate

The certificate in a certificate chain that has not been used to sign another certificate in that chain. The leaf certificate is signed directly or transitively by all other certificates in the chain.

Message digest

The digital fingerprint of an input stream. A cryptographic hash function is applied to an input message arbitrary length and returns a fixed-size output, which is called the digest value.

| | |
|---|---|
| Owned certificate | A certificate whose associated secret or private key resides in a local Cryptographic Service Provider (CSP). Digital-signing algorithms require using owned certificates when signing data for purposes of authentication and non-repudiation. A system may use certificates it does not own for purposes other than signing. |
| Private key | The cryptographic key is used to decipher messages in public-key cryptography. This key is kept secret by its owner. |
| Public key | The cryptographic key is used to encrypt messages in public-key cryptography. The public key is available to multiple users (i.e., the public). |
| Random number generator | A function that generates cryptographically strong random numbers that cannot be easily guessed by an attacker. Random numbers are often used to generate session keys. |
| Root certificate | The prime certificate, such as the official certificate of a corporation or government entity. The root certificate is positioned at the top of the certificate hierarchy in its domain, and it guarantees the other certificates in its certificate chain. Each Certificate Authority (CA) has a self-signed root certificate. The root certificate's public key is the foundation of signature verification in its domain. |
| Secure Electronic Transaction | A mechanism for securely and automatically routing payment information among users, merchants, and their banks. Secure Electronic Transaction (SET) is a protocol for securing bankcard transactions on the Internet or other open networks using cryptographic services.<br><br>SET is a specification designed to utilize technology for authenticating parties involved in payment card purchases on any type of on-line network, including the Internet. SET was developed by Visa and MasterCard, with participation from leading technology companies, including Microsoft, IBM, Netscape, SAIC, GTE, RSA, Terisa Systems, and VeriSign. By using sophisticated cryptographic techniques, SET will make cyberspace a safer place for conducting business and is expected to boost consumer confidence in electronic commerce. SET focuses on maintaining confidentiality of information, ensuring message integrity, and authenticating the parties involved in a transaction. |
| Security Context | A control structure that retains state information shared between a CSP and the application agent requesting service from the CSP. Only one context can be active for an application at any given time, but the application is free to switch among contexts at will, or as required. A security context specifies CSP and application-specific values, such as required key length and desired hash functions. |
| Security-relevant event | An event where a CSP-provided function is performed, a security module is loaded, or a breach of system security is detected. |

| | |
|---|---|
| Session key | A cryptographic key used to encrypt and decrypt data. The key is shared by two or more communicating parties, who use the key to ensure privacy of the exchanged data. |
| Signature | See Digital signature. |
| Signature chain | The hierarchical chain of signers, from the root certificate to the leaf certificate, in a certificate chain. |
| Smart Card | A device (usually similar in size to a credit card) that contains an embedded microprocessor that could be used to store information. Smart cards can store credentials used to authenticate the holder. |
| S/MIME | Secure/Multipurpose Internet Mail Extensions (S/MIME) is a protocol that adds digital signatures and encryption to Internet MIME messages. MIME is the official proposed standard format for extended Internet electronic mail. Internet e-mail messages consist of two parts, the header and the body. The header forms a collection of field/value pairs structured to provide information essential for the transmission of the message. The body is normally unstructured unless the e-mail is in MIME format. MIME defines how the body of an e-mail message is structured. The MIME format permits e-mail to include enhanced text, graphics, audio, and more in a standardized manner via MIME-compliant mail systems. However, MIME itself does not provide any security services.<br><br>The purpose of S/MIME is to define such services, following the syntax given in PKCS #7 for digital signatures and encryption. The MIME body carries a PKCS #7 message, which itself is the result of cryptographic processing on other MIME body parts. |
| Symmetric algorithms | Cryptographic algorithms that use a single secret key for encryption and decryption. Both the sender and receiver must know the secret key. Well-known symmetric functions include Data Encryption Standard (DES) and International Data Encryption Algorithm (IDEA). The U.S. Government endorsed DES as a standard in 1977. It is an encryption block cipher that operates on 64-bit blocks with a 56-bit key. It is designed to be implemented in hardware, and works well for bulk encryption. IDEA, one of the best known public algorithms, uses a 128-bit key. |
| Token | The logical view of a cryptographic device, as defined by a CSP’s interface. A token can be hardware, a physical object, or software. A token contains information about its owner in digital form, and about the services it provides for electronic-commerce and other communication applications. A token is a secure device. It may provide a limited or a broad range of cryptographic functions. Examples of hardware tokens are smart cards and Personal Computer Memory Card International Association (PCMCIA) cards. |

| | |
|---|---|
| Verification | The process of comparing two message digests. One message digest is generated by the message sender and included in the message. The message recipient computes the digest again. If the message digests are exactly the same, it shows or proves there was no tampering of the message contents by a third party (between the sender and the receiver). |
| Web of trust | A trust network among people who know and communicate with each other. Digital certificates are used to represent entities in the web of trust. Any pair of entities can determine the extent of trust between the two, based on their relationship in the web. Based on the trust level, secret keys may be shared and used to encrypt and decrypt all messages exchanged between the two parties. Encrypted exchanges are private, trusted communications. |