THALES

www.thales-esecurity.com

Thales e-Security

nCore

Developer Tutorial

Version: 3.7

Date: 15 January 2016

Contents

Chapter 1: Introduction **11**
Read this guide if 11
Conventions 11
Typographical conventions 11
CLI command conventions 12
Model numbers 12
Document version numbers 12
Further information 12
Contacting Thales Support 13
Chapter 2: nCore architecture **14**
Architecture overview 14
Generating a key 14
Loading a key 15
Transacting a command 16
Chapter 3: C tutorial **18**
Overview 18
nCore API functionality used in this tutorial 19
Variables used in this tutorial 20
Before connecting to the hardserver 21
Declaring a call context 21
Declaring memory allocation upcalls 21
Declaring threading upcalls 22
Initializing the nFast application handle 22
Connecting to the hardserver 23
Getting Security World information 23
Setting up the authorization mechanism 23
Generating a symmetric key 24

Obtaining authorization and selecting a module 26
Preparing the key-generation command and ACL 27
Freeing memory 30
Generating an asymmetric key 31
Obtaining authorization and selecting a module 32
Preparing the key-generation command and ACL 34
Freeing memory 37
Using a key 38
Finding a key 39
Loading a key 40
Encrypting a file 41
Cleaning up resources 44
Chapter 4: Java tutorial 45
Overview 45
Creating a softcard 46
nCore classes used in this tutorial 46
Variables used in this tutorial 47
Before connecting to the hardserver 48
Connecting to the hardserver 48
Generating a key 48
Methods used in generate_key() 52
Using a key 53
Signing a file 53
Cleaning up resources 56
Appendix A: Java examples 57
Java key management example utilities (kmjava) 57
AppKeyGen.java 57
GenerateExport.java 57
KMJavaFloodTest.java 57
NFKMInfo.java 57

NVRamRTCUtil.java 57
SimpleCrypt.java 58
SlotPoller.java 58
Java JCE/CSP example utilities (jcecsp) 58
AsymmetricEncryptionExample.java 58
ECDHExample.java 58
JCEChanTest.java 58
JCEFloodTest.java 58
JCESigTest.java 59
KeyLoadTimer.java 59
KeyStorageExample.java 59
NCipherLibraryInteropExample.java 59
PrepareSslExamples.java 59
PrepareSSLServerExamples.sh 59
SignaturesExample.java 59
SslClientExample.java 59
SslServerExample.java 60
SymmetricEncryptionExample.java 60
SignatureTest.java 60
Java generic stub examples (nfjava) 60
BlobInfo.java 60
Channel.java 60
CheckMod.java 60
CrypTest.java 60
DesKat.java 61
DKTest.java 61
EasyConnection.java 61
Enquiry.java 61
FloodTest.java 61
GenCert.java 61

InitUnit.java 61
NFEnum.java 61
Option.java 61
ParseException.java 62
Parser.java 62
Reference.java 62
ReportVersion.java 62
ScoreKeeper.java 62
SigTest.java 62
Appendix B: Key structures 64
Mechanisms 64
Mech_Any 65
Key Types 66
Random 69
ArcFour 69
Blowfish 70
CAST 71
CAST256 71
DES 71
DES2 73
Triple DES 74
Rijndael 75
SEED 76
Serpent 77
SSLMasterSecret 77
Twofish 79
Diffie-Hellman and ElGamal 80
DSA 83
Elliptic Curve ECDH and ECDSA 86
KCDSA 88

RSA 93
DeriveKey 96
Hash functions 103
SHA-1 103
Tiger 103
SHA-224 104
SHA-256 104
SHA-384 104
SHA-512 105
MD2 105
MD5 105
RIPEMD 160 106
HAS160 106
HMAC signatures 107
ACLs 107
Use limits 110
Actions 113
Action types 113
OpPermissions 113
MakeBlob 114
MakeArchiveBlob 115
NSO 116
NVRAM 117
ReadShare 118
SendShare 118
FileCopy 118
UserAction 119
DeriveKey 119
Using DeriveKey — an example 120
Certificates 129

Using a certificate to authorize an action 130
Generating a certificate to authorize another operation 130
Appendix C: NKFM Functions 134
Debugging NFKM functions 134
Functions 134
NKFM_changepp 134
NFKM_checkconsistency 135
NFKM_checkpp 135
NFKM_cmd_generaterandom 135
NFKM_cmd_destroy 136
NFKM_cmd_loadblob 136
NFKM_cmd_getkeyplain 137
NFKM_erasecard 137
NFKM_erasemodule 137
NFKM_hashpp 137
NFKM_initworld_* 138
NFKM_loadadminkeys_* 141
NFKM_loadcardset_* 146
NFKM_loadworld_* 148
NFKM_makecardset_* 150
NFKM_newkey_* 154
NFKM_operatorcard_changepp 159
NFKM_operatorcard_checkpp 160
NFKM_recordkey 160
NFKM_recordkeys 160
NFKM_replaceacs_* 161
Appendix D: nCore API commands 165
Basic commands 165
ClearUnit 165
ClearUnitEx 166

ModExp 167
ModExpCrt 168
Key-management commands 168
ChangeSharePIN 168
ChannelOpen 169
ChannelUpdate 171
Decrypt 172
DeriveKey 173
Destroy 175
Duplicate 176
Encrypt 176
Export 177
FirmwareAuthenticate 178
FormatToken 178
GenerateKey and GenerateKeyPair 179
GenerateLogicalToken 184
GetChallenge 184
GetKML 185
GetTicket 185
Hash 187
ImpathKXBegin 188
ImpathKXFinish 190
ImpathReceive 190
ImpathSend 191
InitialiseUnit 192
LoadBlob 192
LoadLogicalToken 193
MakeBlob 194
MergeKeyIDs 197
ReadShare 198

RedeemTicket 199
RemoveKM 200
RSAImmedSignDecrypt 200
RSAImmedVerifyEncrypt 201
SetACL 202
SetKM 203
SetNSOPerms 204
SetRTC 206
Sign 206
SignModuleState 207
StaticFeatureEnable 209
UpdateMergedKey 210
Verify 211
WriteShare 212
Commands used by the generic stub only 213
ExistingClient 213
NewClient 214
Glossary 215
Internet addresses 222

Chapter 1: Introduction

This guide describes how to write applications using the nCore API, the native application programming interface for nShield modules. It also describes various programming libraries and utility functions that Thales supplies.

Read this guide in conjunction with the nCore API documentation located in:

- Windows: **`%NFAST_HOME%\document\ncore\html\index.html`** (C) and **`%NFAST_HOME%\java\docs\index.html`** (Java)
- Unix-based: **`/opt/nfast/document/ncore/html/index.html`** (C) and **`/opt/nfast//java/docs/index.html`** (Java).

Read this guide if ...

Read this guide if you are an application developer who is writing cryptographic applications using the nCore API. If you are writing an application using a standard API, such as Java JCE/JCA, MS CAPI, CAPI NG or PKCS #11, you should read the *Cryptographic API Integration Guide*.

The *nCore Developer Tutorial*:

- explains the nCore programming architecture
- presents a tutorial on using the nCore API in C
- presents a tutorial on using the nCore API in Java.

Conventions

Typographical conventions

Note: The word **Note** indicates important supplemental information.

If there is a danger of loss or exposure of key material (or any other security risk), this is indicated by a security triangle in the margin.

Keyboard keys that you must press are represented like this: `Enter`, `Ctrl-C`.

Examples of onscreen text from graphical user interfaces are represented by **boldface** text. Names of files, command-line utilities, and other system items are represented in **`monospace`** text. Variable text that you either see onscreen or that you must enter is represented in *italic*.

Examples of onscreen terminal display, both of data returned and of your input, are represented in a form similar to the following:

```
install
```

CLI command conventions

The basic syntax for a CLI command is:

```
command object <object_name> [parameter] [option] [modifier]
```

In this syntax, user-defined values are shown in *italics* and enclosed within the **< >** characters. Optional elements are shown enclosed within the **[ ]** characters. Mutually exclusive elements are separated by the **|** character.

Many system objects require the inclusion of a user-defined keyword value. For example, the **user** object is executed against a user-supplied *user_name*. Throughout this guide, all user-defined keyword values are shown in *italics*.

Each CLI command that you run performs an operation against the internal configuration of the appliance. The specific type of operation is specified by the first user-defined keyword value in the command string.

Model numbers

Model numbering conventions are used to distinguish different Thales hardware security devices. In the table below, *n* represents any single-digit integer.

Model number	Used for
NH2047	nShield Connect 6000.
NH2040	nShield Connect 1500.
NH2033	nShield Connect 500.
NH2068	nShield Connect 6000+.
NH2061	nShield Connect 1500+.
NH2054	nShield Connect 500+.
nC2021E-000, NCE2023E-000	An nToken (PCI express interface).
nC3*nnn*E-*nnn*, nC4*nnn*E-*nnn*	Thales nShield Solo HSM with a PCI Express (PCIe) interface.
nC30*nn*U-10, nC40*nn*U-10.	An nShield Edge module.

Document version numbers

The version number of this document is shown on the copyright page of this guide. Quote the version number and the date on the copyright page if you need to contact Support about this document.

Further information

This guide forms one part of the information and support provided by Thales. You can find additional documentation in the **document** directory of the installation media for your product.

The *nCore API Documentation* is supplied as HTML files installed in the following locations:

- Windows:
  - API reference for host:`%NFAST_HOME%\document\ncore\html\index.html`
  - API reference for SEE:`%NFAST_HOME%\document\csddoc\html\index.html`
- Unix-based:
  - API reference for host:`/opt/nfast/document/ncore/html/index.html`
  - API reference for SEE:`/opt/nfast/document/csddoc/html/index.html`

The Java Generic Stub classes, nCipherKM JCA/JCE provider classes, and Java Key Management classes are supplied with HTML documentation in standard **Javadoc** format, which is installed in the appropriate `nfast\java` or `nfast/java` directory when you install these classes.

Release notes containing the latest information about your product are available in the `release` directory of your installation media.

Note: We strongly recommend familiarizing yourself with the information provided in the release notes before using any hardware and software related to your product.

If you would like to receive security advisories from Thales, you can subscribe to the low volume security-announce mailing list by emailing the word `subscribe` in the message body to **nShield-securityadvisories@thales-esecurity.com**.

Contacting Thales Support

To obtain support for your product, visit: http://www.thales-esecurity.com/support-landing-page.

Chapter 2: nCore architecture

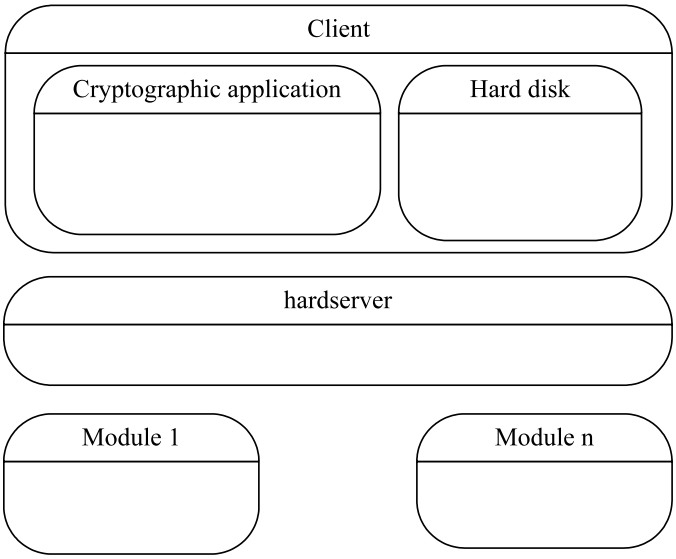
This section describes the interaction between your application and an nShield module that occurs when performing the following cryptographic tasks:

- generating a key
- loading a key
- transacting a command on a module

Architecture overview

Figure 1 illustrates a typical architecture in which one would use the nCore API:

Figure 1. Programming environment architecture



In Figure 1:

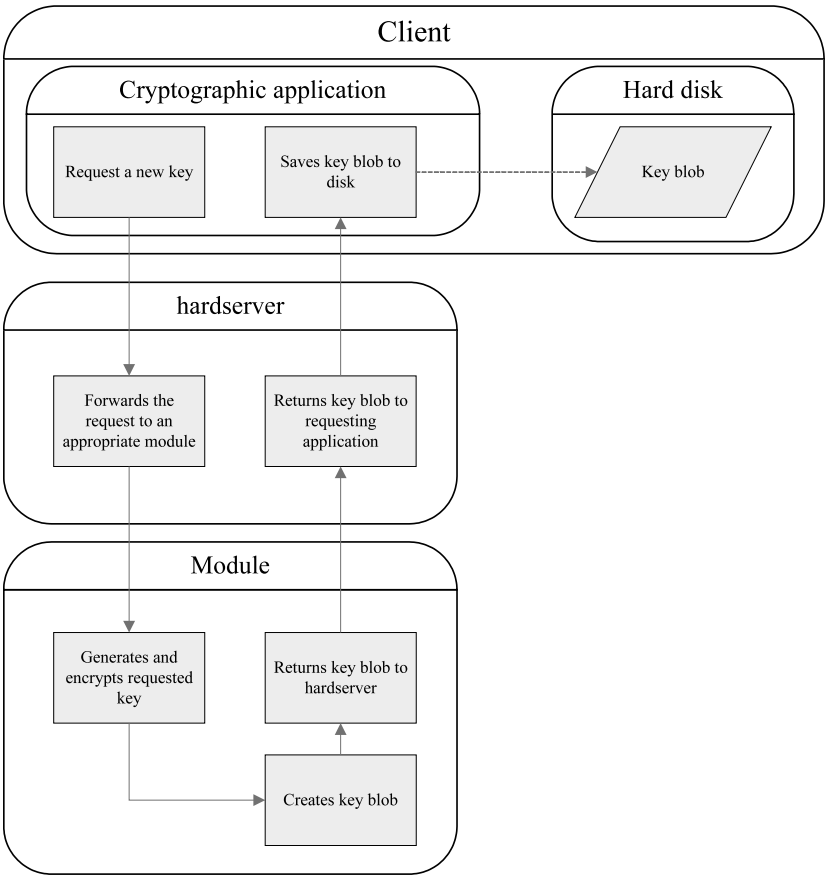
- **Client**: The computer on which your cryptographic application runs.
- **hardserver**: An intermediary between applications and module. The hardserver is responsible for routing commands to modules, and returning the reply from the module to the calling application.
- **module**: The hardware that performs cryptographic tasks.

Generating a key

Keys generated using the nCore API are generally stored in encrypted form on the hard disk of the computer running the cryptographic application. The key blob that contains the encrypted key information is generated by a module when an application uses the module to generate a key.

Figure 2 illustrates the interaction between your cryptographic application and the Security World that occurs during the key-generation process:

Figure 2. Key-generation process



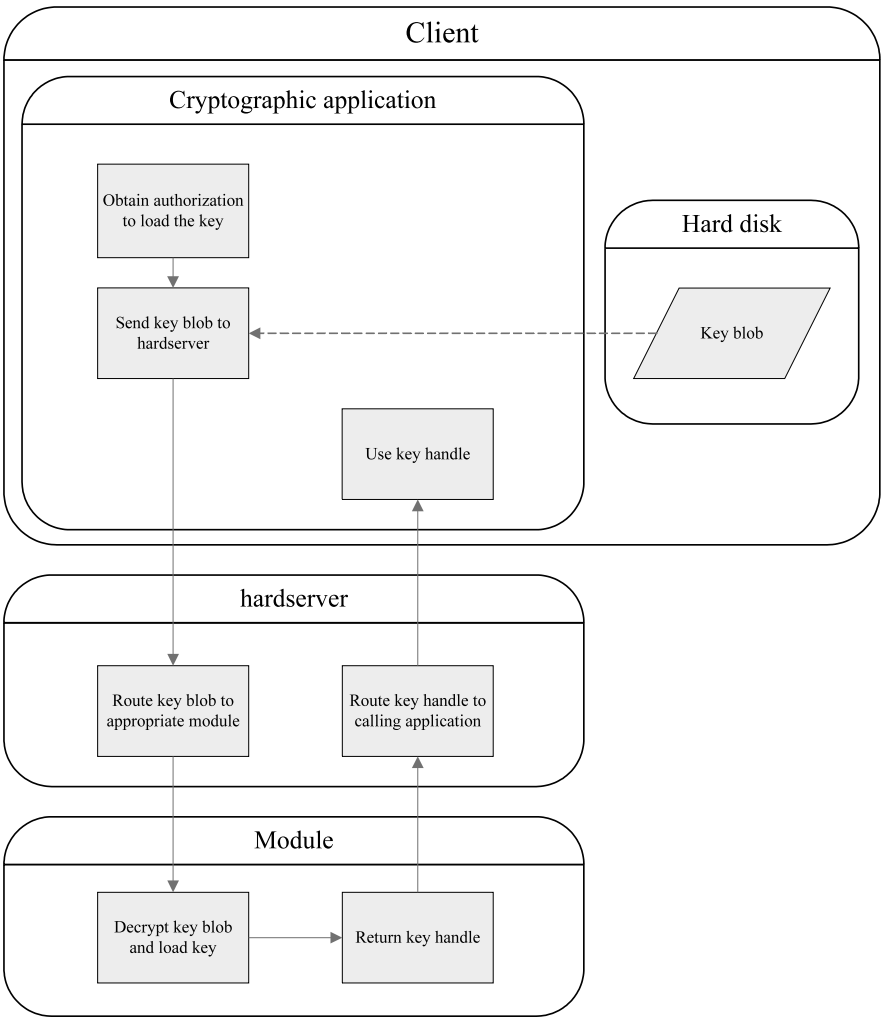
A key blob can only be decrypted by a module that has a record of the key that was used to encrypt the information in the key blob. A key blob contains key information and an Access Control List (ACL) which defines who can use the key and what operations the key can be used for.

Loading a key

Because key information is encrypted in a key blob, the key itself cannot be used to perform a cryptographic operation until it is decrypted. To use a key, you first need to load the encrypted key blob into a module, as illustrated in Figure 3. The key blob is decrypted using a key stored on the module, and a handle or object reference to the key is returned to your application .

In most cases it is necessary to provide authentication in the form of a smart card and/or a pass phrase before using a key. The user interaction that prompts for authentication to be provided is handled by the nCore API.

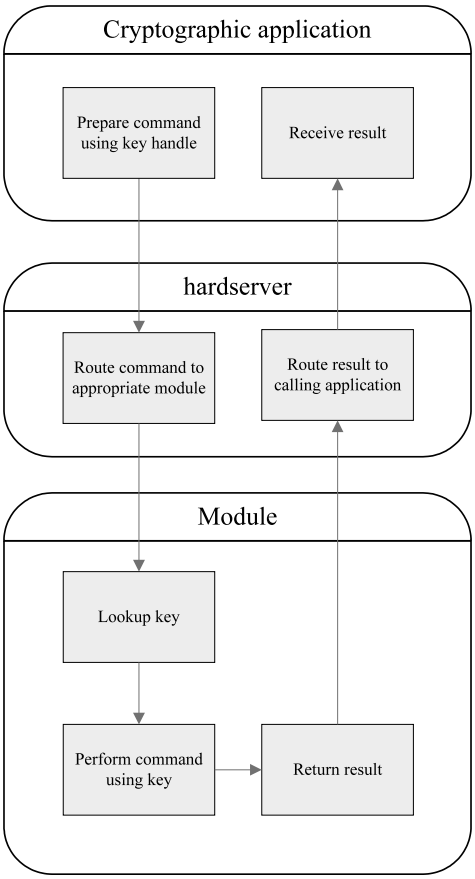
Figure 3. Key-loading process



Transacting a command

After an application has loaded a key, it can instruct a module to use the key to perform cryptographic operations such as encryption, decryption, signing and verification. Figure 4 illustrates the process of transacting a command.

Figure 4. Command transaction process



C tutorial on page 18 explains how to write a C application that:

- creates a connection to the hardserver
- generates a key
- loads a key onto a module
- transacts a command with the module to use the key to encrypt a file.

Java tutorial on page 45 explains how to write a similar Java application which signs a file.

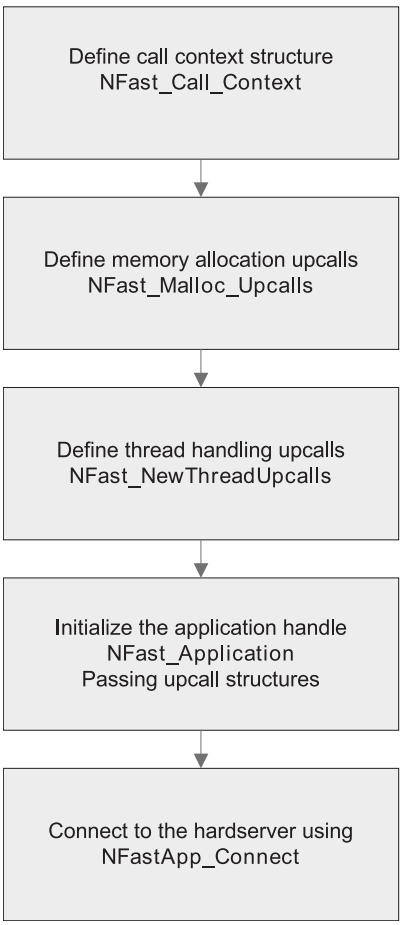
Chapter 3: C tutorial

Overview

This overview section provides a description of how to achieve two fundamental nCore API programming tasks: connecting to the hardserver and transacting a command. These two tasks are common to almost all cryptographic applications. The rest of this chapter works through a simple example of a basic cryptographic application.

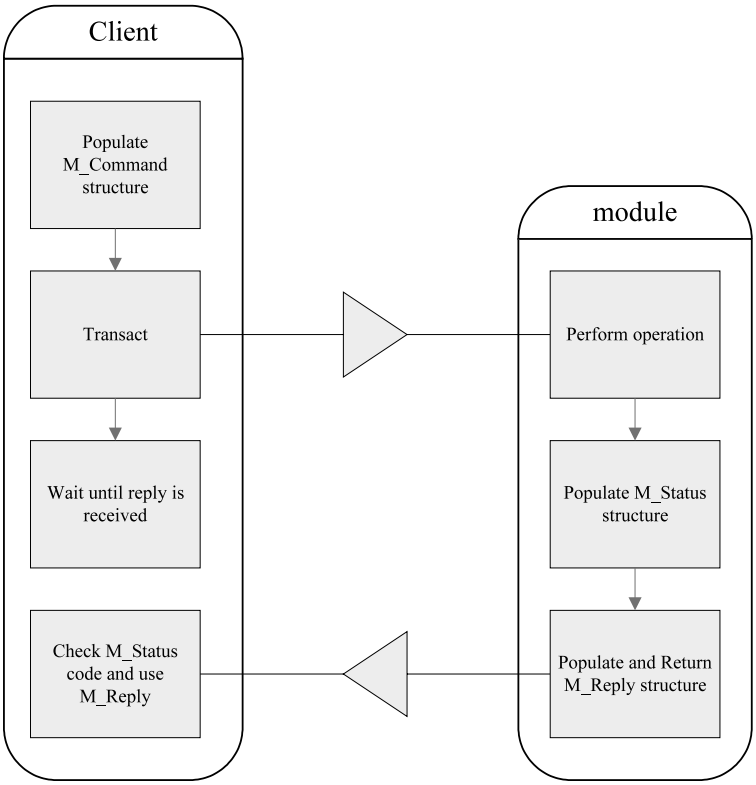
All applications that require nCore functionality first need to create a connection to a hardserver running on an nShield module. Figure 5 illustrates the steps required to create a connection to a hardserver running on Thales hardware:

Figure 5. Connecting to the hardserver



When connected to the hardserver, an application can send an **`M_Command`** to a module. The module processes the command and then returns the results along with any relevant error and status codes. Figure 6 illustrates the process of transacting a cryptographic operation with the module.

Figure 6. Transacting a command



The **M_Reply** structure contains the results of the operation and an **M_Status** message that indicates the outcome of the operation. If a problem was encountered, the **M_Status** value gives an indication of what went wrong. The **M_Reply** contains the results of the command, for example, a key handle or the bytes of an encrypted file.

nCore API functionality used in this tutorial

This tutorial uses the following libraries from the nCore API. You may find it useful to familiarize yourself with these libraries by reading the API documentation, which is located at ***nfast_dir*/document/ncore/html/index.html**.

nfkm.h

This library provides Security World functionality, for example, card-loading libraries, key-generation, and key-loading.

nfinttypes.h

This library is a utility library that provides standard integer types.

nffile.h

This library is a utility library that provides file manipulation functionality.

simplebignum.h

This library is a utility implementation of bignum functionality.

ncthread-upcalls.h

This library is a thread-handling library.

rqcard-applic.h

This library is a card-loading library.

rqcard-fips.h

This library is a card-loading library for use in a FIPS 140-2 level 3 (Federal Information Processing Standards) environment.

Variables used in this tutorial

The following table lists and describes the variables used in this tutorial. Throughout this tutorial you may wish to refer to this table. You may also find it useful to consult the API documentation of the listed types.

Variable Name	Variable Type	Description
rc	M_Status	Status code returned by operations
worldinfo	NFKM_WorldInfo	Information about a Security World
app	NFast_AppHandle	The application handle
app_init_args	NFastAppInitArgs	Used to initialize the application
conn	NFastApp_Connection	The connection to a hardserver
moduleinfo	NFKM_ModuleInfo	Contains information about the module being used
keyident	NFKM_KeyIdent	The name of the key
keyinfo	NFKM_Key	Information about the key
keyid	M_KeyID	The key loaded into the module
ltid	M_KeyID	The card set loaded into the module
keytype	M_KeyType	The cryptographic key type, for example, KeyType_DSA
mech	M_Mech	The encryption mechanism used, for example, Mech_DSA
sigbytes	M_ByteBlock	The marshaled signature
iv	M_IV	The initialization vector
command	M_Command	The command sent to module
reply	M_Reply	The reply returned by the module
idch	M_KeyID	The ID of the channel used for streaming
rqcard	RQCard	The card-loader handle
rqcard_fips	RQCard_FIPS	The card-loader handle used in a FIPS 140-2 level 3 environment

Note: When specifying **KeyType_ECDSAPublic** or **KeyType_ECDHPublic**, the curve parameters for Brainpool curves must be supplied as custom values. Named Brainpool curves are not currently supported. See the nCore API documentation for further information on elliptic curve key generation and the use of custom curve parameters.

Before connecting to the hardserver

The nCore API provides mechanisms that allow you to control how threading, memory allocation, and numbers larger than the available C data types are handled, through an upcall mechanism. Specifying these upcalls is optional. Also optional is the call context structure, which can contain any contextual information that your application might require to keep track of. If you define your own upcalls and call context they must be supplied as arguments when initializing a handle to the hardserver.

Declaring a call context

Many nCore functions take a call context argument, **cctx** or **ctx**, which is passed on to upcalls. The call context structure can be used for any purpose required by an application. For example, the call context could identify an application thread.

The following code shows an example declaration of a call context structure:

```
struct NFast_Call_Context {
  int notused;
};
```

Declaring memory allocation upcalls

By default the nCore API manages memory by using the standard C library functions **malloc**, **realloc**, and **free**. To customize memory management, define a collection of memory allocation upcalls and pass this collection when initializing the application handle. For example, a heavily threaded application may allocate memory per thread, and have separate application handles per thread, to avoid contention. In this code example the memory allocation upcalls re-direct back to the default memory application functions. The call context **cctx** and the transaction context **tctx** can contain any context information required by your application.

```
const NFast_MallocUpcalls mallocupcalls = {
  local_malloc,
  local_realloc,
  local_free
};
static void *local_malloc(size_t nbytes,
    struct NFast_Call_Context *cctx,
    struct NFast_Transaction_Context *tctx) {
  return malloc(nbytes);
}
static void *local_realloc(void *ptr,
    size_t nbytes,
    struct NFast_Call_Context *cctx,
    struct NFast_Transaction_Context *tctx) {
  return realloc(ptr, nbytes);
}
static void local_free(void *ptr,
    struct NFast_Call_Context *cctx,
    struct NFast_Transaction_Context *tctx) {
  free(ptr);
}
```

Declaring threading upcalls

ncthread_upcalls provides a mechanism to specify how threads are implemented on the target platform. If an application needs to use a non-native thread model then the application can either:

- fill in an **nf_thread_upcalls** structure with suitable upcalls and optionally write a translation function **xlate_cctx_to_ncthread()**
- or fill in an **NFast_ThreadUpcalls** structure, and use **NFAPP_IF_THREAD** in the code example below instead of **NFAPP_IF_NEWTHREAD**.

```
const NFast_NewThreadUpcalls newthreadupcalls = {
  &ncthread_upcalls,
  xlate_cctx_to_ncthread
};
static void xlate_cctx_to_ncthread(NFast_AppHandle app,
   struct NFast_Call_Context *cc,
   struct nf_lock_cctx **lcc_r) {
  *lcc_r = 0;
}
```

Initializing the nFast application handle

The hardserver application handle is the main access point to nCore functionality. The following code specifies the application initialization arguments and initializes the application handle. The flags sent to the application initialization function in the following code example are:

- **NFAPP_IF_MALLOC** indicates that an application is setting its own memory allocation upcalls
- **NFAPP_IF_BIGNUM** is necessary for any **bignum** operations to work. The following code example uses **simplebignum** upcalls
- One of **NFAPP_IF_NEWTHREAD** or **NFAST_IF_THREAD** is required in threaded applications. This code example does not perform any multi-threaded operations but the setting are included anyway for the purposes of the example.

```
memset(&app_init_args, 0, sizeof app_init_args);
app_init_args.flags = NFAPP_IF_MALLOC|NFAPP_IF_BIGNUM|NFAPP_IF_NEWTHREAD;
app_init_args.mallocupcalls = &mallocupcalls;
app_init_args.bignumupcalls = &sbn_upcalls;
app_init_args.newthreadupcalls = &newthreadupcalls;

rc = NFastApp_InitEx(&app, &app_init_args, cctx);
```

Connecting to the hardserver

Now that application handle is initialized, create a connection to the hardserver, as shown in the following code example. The **`NFastApp_Connect()`** automatically determines whether to use pipes, local sockets, or TCP sockets, as appropriate.

```
rc = NFastApp_Connect(app, &conn, 0, cctx);
if(rc) {
  NFast_Perror("error calling NFastApp_Connect", rc);
  goto cleanup;
}
```

Getting Security World information

The following code reads in the Security World information that is associated with the application handle. An application handle will only ever be associated with a single Security World, which consists of one or more modules.

```
rc = NFKM_getinfo(app, &worldinfo, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_getinfo", rc);
  goto cleanup;
}
```

Setting up the authorization mechanism

The nCore API supports three types of key protection:

- module protection
- pass phrase protection
- card set protection.

The following three code examples demonstrate how to set up an application to use card set protection.

Initializing the card-loading libraries

The following code initializes the card-loading libraries, which are used later in the example. Card-loading libraries are bound to a single connection and to a single Security World.

```
rc = RQCard_init(&rqcard, app, conn, worldinfo, cctx);
if(rc) {
  NFast_Perror("error calling RQCard_init", rc);
  goto cleanup;
}
rqcard_initialized = 1;
```

Obtaining additional FIPS authorization

Strict FIPS 140 mode requires authorization for key-generation, which can be obtained from either an Operator Card or an Administrator Card. The following code initializes the strict FIPS code library, which seeks strict FIPS 140 authorization when this is required.

```
rc = RQCard_fips_init(&rqcard, &rqcard_fips);
if(rc) {
  NFast_Perror("error calling RQCard_fips_init", rc);
  goto cleanup;
}
rqcard_fips_initialized = 1;
```

Selecting a user interface

The following code selects the default user interface for the platform on which the example is running. The user interface will be displayed to the user when authorization is required to perform an operation.

```
rc = RQCard_ui_default(&rqcard);
if(rc) {
  NFast_Perror("error calling RQCard_ui_default", rc);
  goto cleanup;
}
```

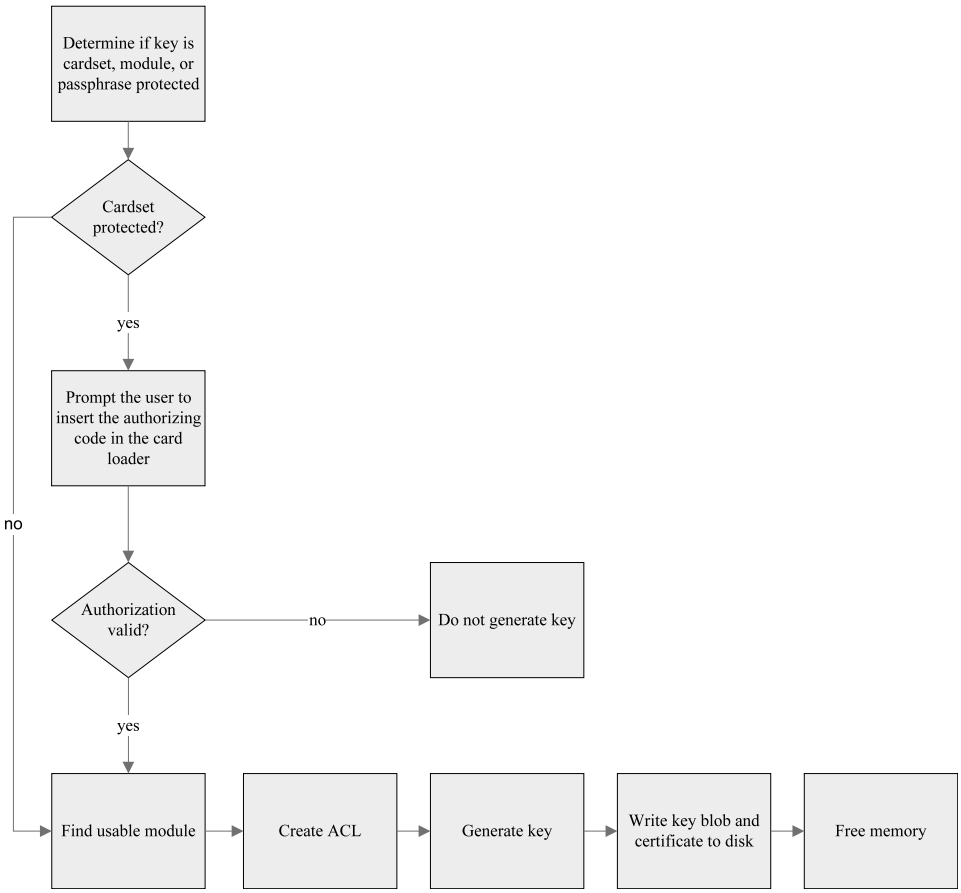
Generating a symmetric key

This section describes the key-generation process in detail. The process of generating a symmetric key differs slightly from the process of generating an asymmetric key, so each is described in a separate section. There is some repetition in the two sections.

Note: This section does not explain how to use softcards to protect keys. Softcards can be listed with **`NFKM_listsoftcards()`** and loaded with **`NFKM_loadsoftcard()`**. For more information about using softcards, see the information about **`nkfm.h`** in the nCore API documentation.

Figure 7 illustrates the key-generation process.

Figure 7. The key-generation process



The code in this section makes use of the following variables:

Variable Name	Variable Type	Description
`acl_params`	`NFKM_MakeACL_Params`	Used to construct ACLs
`blob_params`	`NFKM_MakeBlobs_Params`	Used when making blobs
`keyinfo`	`NFKM_Key`	Information about a key
`moduleinfo`	`NFKM_ModuleInfo`	The module to use
`mc`	`M_ModuleCert`	A certificate from a module
`fips140authhandle`	`NFKM_FIPS140AuthHandle`	FIPS authorization
`ltid`	`M_KeyID`	A loaded card set
`cardset`	`NFKM_CardSet`	Information about a card set
`moduleid`	`M_ModuleID`	The ID of a module
`cardhash`	`NFKM_CardSetIdent`	A hash of a card set
`rc`	`M_Status`	A command return code
`command`	`M_Command`	A command structure
`reply`	`M_Reply`	A command reply

Obtaining authorization and selecting a module

Keys are generated on a specific module and protected by some form of authorization. When a key is generated the type of authorization that is required to use the key is defined, as well as the purposes for which the key is allowed to be used, for example, only for encryption and decryption, or only for signing and verification.

Using card set protection

The following code prompts the user to provide a card to protect the key that will be generated. The card set hash populates **`cardhash`** when the card-loader completes.

```
rc = RQCard_logic_ocs_anyone(rqcard, &cardhash,
                             "Insert a card set to protect the new key");
if(rc) {
  NFast_Perror("error calling RQCard_logic_ocs_anyone", rc);
  goto cleanup;
}
```

Selecting a Security World module

Now that authorization has been obtained, prompt the user to select a module in the Security World on which to generate the key. Alternatively you could use the **`RQCard_whichmodule_specific()`** function to dictate which module will be used, or the **`NFKM_getusablemodule()`** function to use the first available module.

The module ID and a key ID for the desired card set on that module are assigned to the **`moduleid`** and **`ltid`** variables when the card-loader completes.

```
rc = RQCard_whichmodule_anyone(rqcard, &moduleid, &ltid);
if(rc) {
  NFast_Perror("error calling RQCard_whichmodule_anyone", rc);
  goto cleanup;
}

rc = rqcard->uf->eventloop(rqcard);
if(rc) {
  NFast_Perror("error running card loader", rc);
  goto cleanup;
}
```

The **`moduleid`**, **`id`**, and **`ltid`** variables are now populated. Next, populate the **`moduleinfo`** variable for the chosen module, and create a card set handle.

```
for(n = 0; n < worldinfo->n_modules; ++n)
  if(worldinfo->modules[n]->module == moduleid)
    break;
assert(n < worldinfo->n_modules);
moduleinfo = worldinfo->modules[n];

rc = NFKM_findcardset(app, &cardhash, &cardset, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_findcardset", rc);
  goto cleanup;
}
```

Note: Up to now in this example the application has performed actions common to generating either a symmetric key or an asymmetric key. The process from here on differs depending on which key type is generated.

Preparing the key-generation command and ACL

Start by setting up some command parameters based on the information we have already gathered.

```
command.cmd = Cmd_GenerateKeyPair;
command.args.generatekey.params.type = keytype;
command.args.generatekey.flags = Cmd_GenerateKey_Args_flags_Certify;
command.args.generatekey.module = moduleinfo->module;
```

Keys are stored with an ACL, which defines which entities can perform operations with the key. The next step is to populate the **`acl_params`** variable with the information needed to create the ACL that will be stored in the key blob along with the key we generate. In this example the application sets the **`acl_params.f`** flags parameter to enable key recovery and specify the type of key protection to use. There are three options:

- card set protection
- module protection
- pass phrase protection.

This following code demonstrates how to indicate that a key should be protected by a card set. In this case, the card set is the one selected earlier by the user in *Selecting a Security World module* on page 26.

```
acl_params.f = NFKM_NKF_RecoveryEnabled|protection;
acl_params.cs = cardset;
```

The make ACL blob flags (**`acl_params.f`**) parameter must be same as the make blob flags parameter (**`blob_params.f`**), so is set accordingly.

```
acl_params.f = blob_params.f;
```

The next step is to define in the ACL for which operations the key is allowed to be used. In this example, the application specifies that the key can be used to sign, verify, encrypt, or decrypt.

```
acl_params.op_base = (NFKM_DEFOPPERMS_SIGN
              |NFKM_DEFOPPERMS_VERIFY
              |NFKM_DEFOPPERMS_ENCRYPT
              |NFKM_DEFOPPERMS_DECRYPT);
```

The application is now ready to generate the ACL:

```
rc = NFKM_newkey_makeaclx(app, conn, worldinfo, &acl_params,
                          &command.args.generatekey.acl, cctx);
```

The following code sets up further generate key command parameters. The parameters that are required differ according to key type. For example, if an application is generating a Rijndael key, you need to specify the length of the key required, in bytes:

```
command.args.generatekey.params.params.random.lenbytes = 128/8;
```

Generating a key in a strict FIPS environment requires that an application obtains authorization (in this case, card set authorization) before attempting to generate a key. It is possible that the card loader has already obtained the necessary authorization from a prior card-loading operation. In this case, the following call will retrieve this authorization:

```
rc = RQCard_fips_get(rqcard_fips, moduleinfo->module, &fips140authhandle,
                     0);
```

If this call returns **`Status_RQCardMustContinue`**, an application must explicitly attempt to obtain the correct authorization as follows:

```
rc = RQCard_fips_logic(rqcard);
if(rc) {
  NFast_Perror("error calling RQCard_fips_logic", rc);
  goto cleanup;
}
rc = RQCard_whichmodule_specific(rqcard, moduleinfo->module, 0);
if(rc) {
  NFast_Perror("error calling RQCard_whichmodule_anyone", rc);
  goto cleanup;
}
rc = rqcard->uf->eventloop(rqcard);
if(rc) {
  NFast_Perror("error running card loader", rc);
  goto cleanup;
}
rc = RQCard_fips_get(rqcard_fips, moduleinfo->module, &fips140authhandle,
                     0);
```

Now that the application has obtained the necessary strict FIPS authorization (or cancelled the operation if the correct authorization could not be obtained), it can use the authorization to authorize the creation of the key.

```
rc = NFKM_newkey_makeauth(app, worldinfo, &command.flags, &command.certs,
                          fips140authhandle, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_newkey_makeauth", rc);
  goto cleanup;
}
```

With or without FIPS authorization, the application has now obtained all the information necessary to transact a key-generation operation, so is now ready to send the key-generation command to the selected module. The reply is checked using the reply checking utility function mentioned at the beginning of the chapter.

```
rc = NFastApp_Transact(conn, cctx, &command, &reply, 0);
rc = check_reply(rc, &reply, "error generating new key");
if(rc)
  goto cleanup;
```

The application has now generated a new key, but as yet the key exists only in the module's memory. Next, construct an **`NFKM_Key`** key information structure (**`keyinfo`**) and then save it to disk.

```
keyinfo->v = 8;
keyinfo->appname = keyident.appname;
keyinfo->ident = keyident.ident;
time(&keyinfo->gentime);
```

The next step is to populate the parameters of the **`blob_params`** structure, which contains the information that is to be written to the key blob. The following code also checks that a key-generation certificate was included in the reply. The **`NFKM_MakeBlobsParams`** flags **`blob_params.f`** must be the same as the flags passed to **`NFKM_newkey_makeaclx()`** when the application created the private ACL.

```
mc = 0;
blob_params.kpriv = reply.reply.generatekey.key;
if(reply.reply.generatekey.flags & Cmd_GenerateKey_Reply_flags_cert_present)
  mc = reply.reply.generatekey.cert;
if(cardset) {

  blob_params.lt = ltid;
  blob_params.cs = cardset;
}
blob_params.fips = fips140authhandle;
```

The parameters required for the **`NFKM_newkey_makeblobsx()`** are now populated, and the application is ready to create the key blob. As this is a symmetric key type the application need only save a private key blob.

```
rc = NFKM_newkey_makeblobsx(app, conn, worldinfo, &blob_params, keyinfo, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_newkey_makeblobsx", rc);
  goto cleanup;
}
if(mc) {
  rc = NFKM_newkey_writecert(app, conn, moduleinfo, blob_params.kpriv, mc,
                             keyinfo, cctx);
  if(rc) {
    NFast_Perror("error calling NFKM_newkey_writecert", rc);
    goto cleanup;
  }
}
```

The **keyinfo** structure is now ready to be saved to disk.

```
rc = NFKM_recordkey(app, keyinfo, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_recordkey", rc);
  goto cleanup;
}
rc = Status_OK;
```

Freeing memory

The final part of the key-generation process is the important step of unloading the key information in the module.

```
NFastApp_FreeACL(app, cctx, 0, &command.args.generatekey.acl);
NFKM_cmd_destroy(app, conn, 0, reply.reply.generatekey.key,
                 "generatekey.key", cctx);
if(ltid) NFKM_cmd_destroy(app, conn, 0, ltid, "ltid", cctx);
```

If you are running your application in strict FIPS mode, **NFKM_newkey_makeauth()** creates a certificate list, which also needs to be freed:

```
if(command.flags & Command_flags_certs_present)
   NFastApp_Free_CertificateList(app, cctx, 0, command.certs);

NFastApp_Free_Reply(app, cctx, 0, &reply);
keyinfo->appname = 0;
keyinfo->ident = 0;
NFKM_freekey(app, keyinfo, cctx);
NFKM_freecardset(app, cardset, cctx);
```

This concludes the explanation of symmetric key-generation. The next section describes the process of generating asymmetric keys.

Generating an asymmetric key

This section describes the asymmetric key-generation process in detail. The process of generating a symmetric key differs slightly from the process of generating an asymmetric key, so each is described in a separate section. There is some repetition in the two sections.

Note: This section does not explain how to use softcards to protect keys. Softcards can be listed with **`NFKM_listsoftcards()`** and loaded with **`NFKM_loadsoftcard()`**. See the nCore API documentation of **`nkfm.h`** for more information about using softcards.

Figure 8 illustrates the key-generation process.

Figure 8. The key-generation process

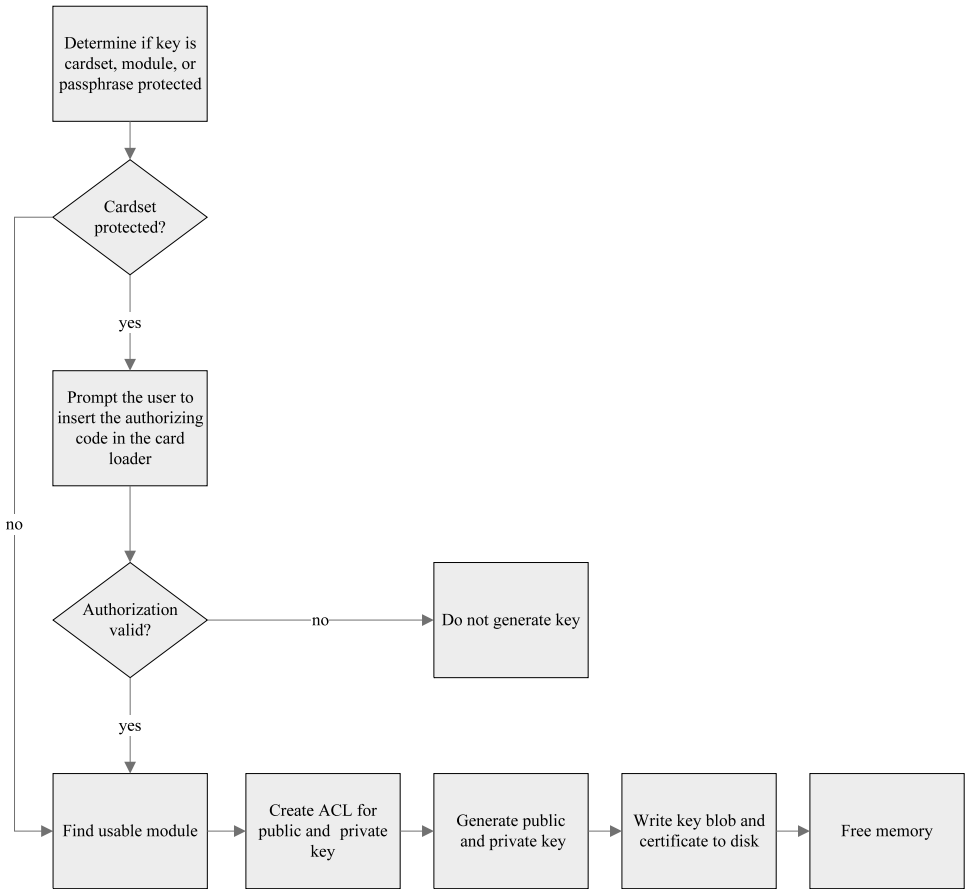
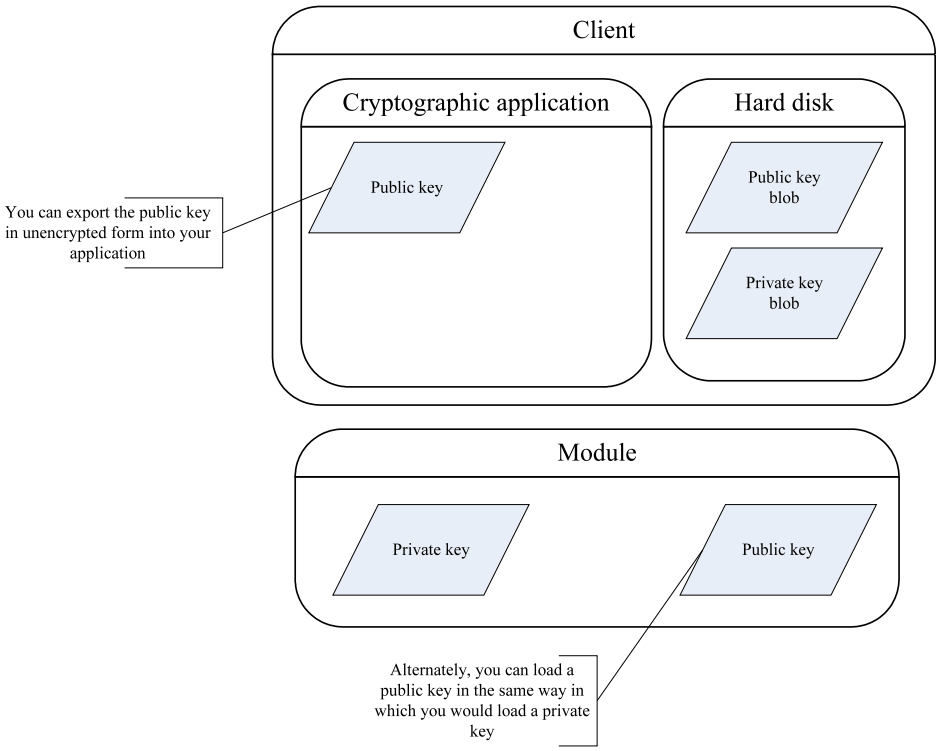


Figure 9 shows how generated asymmetric keys are stored in the programming environment architecture. See *nCore architecture* on page 14 for more information about the programming environment architecture.

Figure 9. Asymmetric key storage



The code in this section makes use of the following variables:

Variable Name	Variable Type	Description
`acl_params`	`NFKM_MakeACL_Params`	Used to construct ACLs
`blob_params`	`NFKM_MakeBlobs_Params`	Used when making blobs
`keyinfo`	`NFKM_Key`	Information about a key
`moduleinfo`	`NFKM_ModuleInfo`	The module to use
`mc`	`M_ModuleCert`	A certificate from a module
`fips140authhandle`	`NFKM_FIPS140AuthHandle`	FIPS authorization
`ltid`	`M_KeyID`	A loaded card set
`cardset`	`NFKM_CardSet`	Information about a card set
`moduleid`	`M_ModuleID`	The ID of a module
`cardhash`	`NFKM_CardSetIdent`	A hash of a card set
`rc`	`M_Status`	A command return code
`command`	`M_Command`	A command structure
`reply`	`M_Reply`	A command reply

Obtaining authorization and selecting a module

Keys are generated on a specific module and protected by some form of authorization. When a key is generated the type of authorization that is required to use the key is defined, as well as the purposes

for which the key is allowed to be used, for example, only for encryption and decryption, or only for signing and verification.

Using card set protection

Proper authorization is required to generate a key. This example handles card set authorization. The following code prompts the user to provide a card to protect the key that is to be generated. The card set hash populates **cardhash** when the card-loader completes.

```
rc = RQCard_logic_ocs_anyone(rqcard, &cardhash,
                             "Insert a cardset to protect the new key");
if(rc) {
  NFast_Perror("error calling RQCard_logic_ocs_anyone", rc);
  goto cleanup;
}
```

Selecting a Security World module

Now that authorization has been obtained, prompt the user to select a module in the Security World on which to generate the key. Alternatively you could use the **RQCard_whichmodule_specific()** function to dictate which module to use or the **NFKM_getusablemodule()** function to use the first available module.

The module ID and a key ID for the desired card set on that module are assigned to the **moduleid** and **ltid** variables when the card-loader completes.

```
rc = RQCard_whichmodule_anyone(rqcard, &moduleid, &ltid);
if(rc) {
  NFast_Perror("error calling RQCard_whichmodule_anyone", rc);
  goto cleanup;
}

rc = rqcard->uf->eventloop(rqcard);
if(rc) {
  NFast_Perror("error running card loader", rc);
  goto cleanup;
}
```

The **moduleid**, **id** and **ltid** are now populated. The next step is to populate the **moduleinfo** variable for the chosen module, and create a card set handle.

```
for(n = 0; n < worldinfo->n_modules; ++n)
  if(worldinfo->modules[n]->module == moduleid)
    break;
assert(n < worldinfo->n_modules);
moduleinfo = worldinfo->modules[n];

rc = NFKM_findcardset(app, &cardhash, &cardset, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_findcardset", rc);
  goto cleanup;
}
```

Note: Up to now in this example the application has performed actions common to generating either a symmetric key or an asymmetric key. The process from here on differs depending on which key type is generated.

Preparing the key-generation command and ACL

Start by setting up some command parameters based on the information we have already gathered.

```
command.cmd = Cmd_GenerateKeyPair;
command.args.generatekeypair.params.type = keytype;
command.args.generatekeypair.flags = Cmd_GenerateKeyPair_Args_flags_Certify;
command.args.generatekeypair.module = moduleinfo->module;
```

Keys are stored with an ACL which defines which entities can perform operations with the key. The next step is to populate the **`acl_params`** variable with the information needed to create the ACL that is stored in the key blob along with the key we generate. The application sets the **`acl_params.f`** flags parameter to enable key recovery, and specify the type of key protection to use. There are three options:

- card set protection
- module protection
- pass phrase protection.

This following code demonstrates how to indicate that a key should be protected by a card set. In this case the card set is the one selected earlier by the user in *Selecting a Security World module* on page 26.

```
acl_params.f = NFKM_NKF_RecoveryEnabled|protection;
acl_params.cs = cardset;
```

The make ACL blob flags (**`acl_params.f`**) must be same as the make blob flags (**`blob_params.f`**), so it is set accordingly.

```
blob_params.f = acl_params.f;
```

The next step is to define in the ACL which operations the key is allowed to be used for. Firstly the application defines the allowed uses for the private key ACL. The **`is_signing_only_keytype()`** function is not an nCore function:

```
if(is_signing_only_keytype(keytype))
  acl_params.op_base = NFKM_DEFOPPERMS_SIGN;
else if(is_encryption_only_keytype(keytype))
  acl_params.op_base = NFKM_DEFOPPERMS_DECRYPT;
else
  acl_params.op_base = (NFKM_DEFOPPERMS_SIGN
                |NFKM_DEFOPPERMS_DECRYPT);
```

The application is now ready to generate the private key ACL:

```
rc = NFKM_newkey_makeaclx(app, conn, worldinfo, &acl_params,
                          &command.args.generatekeypair.aclpriv, cctx);
```

For asymmetric keys the application also defines a public key ACL.

```
acl_params.f = NFKM_NKF_PublicKey;
if(is_signing_only_keytype(keytype))
  acl_params.op_base = NFKM_DEFOPPERMS_VERIFY;
else if(is_encryption_only_keytype(keytype))
  acl_params.op_base = NFKM_DEFOPPERMS_ENCRYPT;
else
  acl_params.op_base = (NFKM_DEFOPPERMS_VERIFY
                 |NFKM_DEFOPPERMS_ENCRYPT);
```

The public key ACL is created in the same manner as the private key ACL:

```
rc = NFKM_newkey_makeaclx(app, conn, worldinfo, &acl_params,
                          &command.args.generatekeypair.aclpub, cctx);
```

The following code sets up further key generation command parameters. The parameters that are required differ according to key type. For example, an application might use the following code when generating a 1024 bit DSA key using strict key verification. For details of the parameters required for the types of key you want to generate, see the relevant nCore API documentation.

```
command.args.generatekeypair.params.params.dsaprivate.flags =
  KeyType_DSAPrivate_GenParams_flags_Strict;
command.args.generatekeypair.params.params.dsaprivate.lenbits = 1024;
```

Generating a key in a strict FIPS environment requires that an application obtains authorization (in this case, card set authorization) before attempting to generate a key. It is possible that the card loader has already obtained the necessary authorization from a prior card-loading operation. In this case, the following call retrieves this authorization:

```
rc = RQCard_fips_get(rqcard_fips, moduleinfo->module, &fips140authhandle,
                     0);
```

If this call returns **`Status_RQCardMustContinue`**, an application must explicitly attempt to obtain the correct authorization as follows:

```
rc = RQCard_fips_logic(rqcard);
if(rc) {
  NFast_Perror("error calling RQCard_fips_logic", rc);
  goto cleanup;
}
rc = RQCard_whichmodule_specific(rqcard, moduleinfo->module, 0);
if(rc) {
  NFast_Perror("error calling RQCard_whichmodule_anyone", rc);
  goto cleanup;
}
rc = rqcard->uf->eventloop(rqcard);
if(rc) {
  NFast_Perror("error running card loader", rc);
  goto cleanup;
}
rc = RQCard_fips_get(rqcard_fips, moduleinfo->module, &fips140authhandle,
                     0);
```

Now that the application has obtained the necessary strict FIPS authorization (or cancelled the operation if the correct authorization could not be obtained), it can use the authorization to authorize the creation of the key.

```
rc = NFKM_newkey_makeauth(app, worldinfo, &command.flags, &command.certs,
                          fips140authhandle, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_newkey_makeauth", rc);
  goto cleanup;
}
```

With or without FIPS authorization, the application has now obtained all the information necessary to transact a key-generation operation, so is now ready to send the key-generation command to the selected module. The reply is checked using the reply checking utility function mentioned at the beginning of the chapter.

```
rc = NFastApp_Transact(conn, cctx, &command, &reply, 0);
rc = check_reply(rc, &reply, "error generating new key");
if(rc)
  goto cleanup;
```

The application has now generated a new key, but as yet the key exists only in the module's memory. Next, construct an **`NFKM_Key`** key information structure (**`keyinfo`**) and then save it to disk.

```
keyinfo->v = 8;
keyinfo->appname = keyident.appname;
keyinfo->ident = keyident.ident;
time(&keyinfo->gentime);
```

The next step is to populate the parameters of the **`blob_params`** structure, which contains the information that will be written to the key blob. The following code also checks that a key-generation

certificate was included in the reply. The **NFKM_MakeBlobsParams** flags **blob_params.f** must be the same as the flags passed to **NFKM_newkey_makeaclx()** when the application created the private ACL.

```
mc = 0;
blob_params.kpriv = reply.reply.generatekeypair.keypriv;
blob_params.kpub = reply.reply.generatekeypair.keypub;
if(reply.reply.generatekeypair.flags & Cmd_GenerateKeyPair_Reply_flags_certpriv_present)
  mc = reply.reply.generatekeypair.certpriv;
if(cardset) {
  blob_params.lt = ltid;
  blob_params.cs = cardset;
}
blob_params.fips = fips140authhandle;
```

The parameters required for the **NFKM_newkey_makeblobsx()** are now populated and the application can now create the key blob.

```
rc = NFKM_newkey_makeblobsx(app, conn, worldinfo, &blob_params, keyinfo, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_newkey_makeblobsx", rc);
  goto cleanup;
}
if(mc) {
  rc = NFKM_newkey_writecert(app, conn, moduleinfo, blob_params.kpriv, mc,
                             keyinfo, cctx);
  if(rc) {
    NFast_Perror("error calling NFKM_newkey_writecert", rc);
    goto cleanup;
  }
}
```

The **keyinfo** structure is now ready to be saved to disk.

```
rc = NFKM_recordkey(app, keyinfo, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_recordkey", rc);
  goto cleanup;
}
rc = Status_OK;
```

Freeing memory

The final part of the key-generation process is the important step of freeing the memory used by the application, so that no key information remains in memory, which would make the key vulnerable to attackers.

```
NFastApp_FreeACL(app, cctx, 0, &command.args.generatekeypair.aclpriv);
NFastApp_FreeACL(app, cctx, 0, &command.args.generatekeypair.aclpub);
NFKM_cmd_destroy(app, conn, 0, reply.reply.generatekeypair.keypriv,
                 "generatekeypair.keypriv", cctx);
NFKM_cmd_destroy(app, conn, 0, reply.reply.generatekeypair.keypub,
                 "generatekeypair.keypub", cctx);
if(ltid) NFKM_cmd_destroy(app, conn, 0, ltid, "ltid", cctx);
```

If you are running your application in strict FIPS mode, **`NFKM_newkey_makeauth()`** will have created a certificate list, which also needs to be freed:

```
if(command.flags & Command_flags_certs_present)
   NFastApp_Free_CertificateList(app, cctx, 0, command.certs);

NFastApp_Free_Reply(app, cctx, 0, &reply);
keyinfo->appname = 0;
keyinfo->ident = 0;
NFKM_freekey(app, keyinfo, cctx);
NFKM_freecardset(app, cardset, cctx);
```

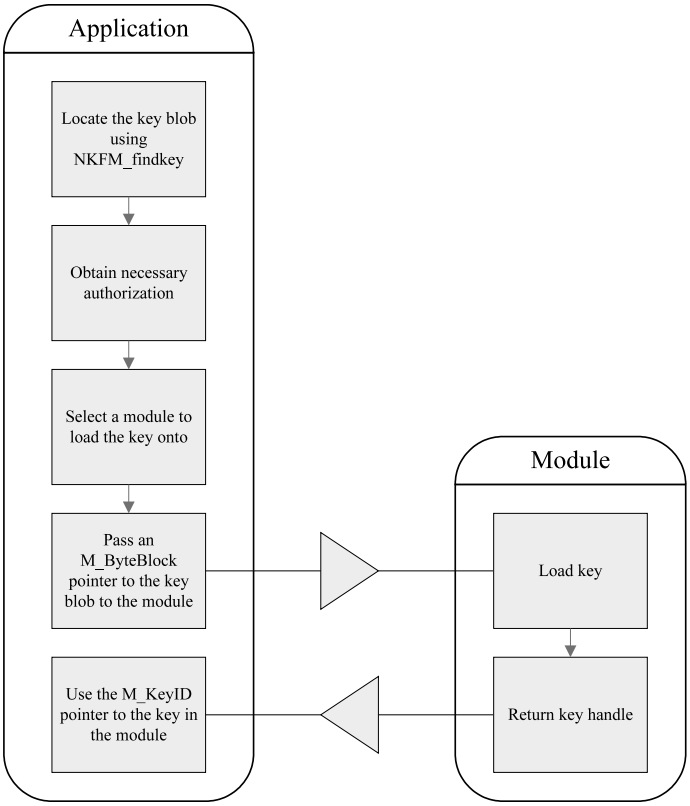
This concludes the explanation of asymmetric key-generation.

Using a key

Once a key has been generated on a module the encrypted key information, or key blob, is stored on the hard disk of the application that requested it. For your application to use a key, you first need to pass the information contained in the key blob to the hardserver, which will use a module to decrypt the key and return a key handle to your application.

The following diagram shows the process of loading a key:

Figure 10. Loading a key



Finding a key

To load a key, first locate the key blob. A key is identified by the name of the application that created it and the key identifier. The following code tries to find an existing key blob of the requested type. If a key of this type cannot be found, the code generates a new key.

The following code uses a function called **`generate_key()`** to generate a key if a key cannot be found.

```
rc = NFKM_findkey(app, keyident, &keyinfo, cctx);
if(rc) {
  NFast_Perror("error calling NFKM_findkey", rc);
  goto cleanup;
}
if(!keyinfo) {
  rc = generate_key(app, conn, worldinfo, &rqcard, &rqcard_fips, opt_protect,
                    keyident, keytype, cctx);
  if(rc)
    goto cleanup;
  rc = NFKM_findkey(app, keyident, &keyinfo, cctx);
  if(rc) {
    NFast_Perror("error calling NFKM_findkey", rc);
    goto cleanup;
  }
  if(keyinfo == 0) {
    fprintf(stderr,
            "NFKM_findkey could not find key even after generating it.\n");
    rc = -1;
    goto cleanup;
  }
}
```

Loading a key

Before a key can be loaded into a module, an application must obtain the appropriate authorization. In this example the authorization required comes from a card in a card set, so the application must first initialize the card-loading libraries:

```
if(keyinfo->flags & Key_flags_ProtectionCardSet) {
  M_ModuleID moduleid;
  int n;
  rc = RQCard_logic_ocs_specific(&rqcard, &keyinfo->cardset,
                                 "Load cardset");
  if(rc) {
    NFast_Perror("error calling RQCard_logic_ocs_specific", rc);
    goto cleanup;
  }
}
```

A Security World often contains multiple modules, many of which may have the key that is needed to decrypt the key blob an application wants to load. For this example the user is prompted to choose a module that contains the necessary key, and then prompted to provide the card that authorizes the use of the key:

```
rc = RQCard_whichmodule_anyone(&rqcard, &moduleid, &ltid);
if(rc) {
  NFast_Perror("error calling RQCard_whichmodule_anyone", rc);
  goto cleanup;
}
rc = rqcard.uf->eventloop(&rqcard);
if(rc) {
  NFast_Perror("error running card loader", rc);
  goto cleanup;
}
```

It is also possible for an application to ask the Security World to nominate a usable module by using the **`NFKM_getusablemodule()`** function:

```
rc = NFKM_getusablemodule(worldinfo, 0, &moduleinfo);
if(rc) {
  NFast_Perror("error calling NFKM_getusablemodule", rc);
  goto cleanup;
}
```

Now that the user has selected a module, an application can populate the **`moduleinfo`** variable, which is later used as a parameter to the **`NFKM_cmd_loadblob()`** function.

```
for(n = 0; n < worldinfo->n_modules; ++n)
  if(worldinfo->modules[n]->module == moduleid)
    break;
  assert(n < worldinfo->n_modules);
  moduleinfo = worldinfo->modules[n];
```

The application has now gathered all the information it needs to load the key onto a module using the **NFKM_cmd_loadblob()** function. The next step is to prepare a pointer to the key that will be loaded into the module. The following code loads the public key blob. An application can load the private key blob in similar fashion using **&keyinfo->privblob**.

```
const M_ByteBlock *blobptr;
blobptr = &keyinfo->pubblob;
```

The following code attempts to load the key blob. **NFKM_cmd_loadblob()** fills in the command structure and handles the reply. Assuming that the command executes successfully, you will now have a handle on the key loaded onto the selected module.

Note: It is possible to construct an **M_Command** structure by using **Cmd_LoadBlob()** directly instead.

```
rc = NFKM_cmd_loadblob(app,
                       conn,
                       moduleinfo->module,
                       blobptr,
                       ltid,
                       &keyid,
                       "loading key blob",
                       cctx);
if(rc) {
  NFast_Perror("error calling NFKM_cmd_loadblob", rc);
  goto cleanup;
}
```

Encrypting a file

This section demonstrates how to encrypt the contents of a text file by using a secure channel. For the sake of simplicity, this example has no error handling.

First, generate an appropriate initialization vector:

```
iv.mech = Mech_RijndaelmCBCi128pPKCS5;
for (i=0; i<sizeof iv->generic128.iv.bytes; i++)
  iv.iv->generic128.iv.bytes[i]=(unsigned char)((i*19) ^ iv.mech);
```

Next, open a channel to use to encrypt the file. The mechanism that the channel uses to encrypt the file is specified when the channel is opened:

```
M_Command channel_open_command;
M_Reply channel_open_reply;
M_Status channel_open_rc;
channel_open_command.cmd = Cmd_ChannelOpen;
channel_open_command.args.channelopen.type = ChannelType_Any;
channel_open_command.args.channelopen.mode = ChannelMode_Encrypt;
channel_open_command.args.channelopen.mech = mech;
```

Some **M_Command** arguments are optional. In this example, the application specifies both the key to be used to encrypt the file and the initialization vector and indicates which optional arguments have been specified by setting the appropriate flags:

```
channel_open_command.args.channelopen.flags |= Cmd_ChannelOpen_Args_flags_key_present;
channel_open_command.args.channelopen.key = &keyid;
channel_open_command.args.channelopen.flags |= Cmd_ChannelOpen_Args_flags_given_iv_present;
channel_open_command.args.channelopen.given_iv = iv;
```

To open the channel, transact the **M_Command** in the usual way and then set the channel ID pointer **idch**:

```
channel_open_rc = NFastApp_Transact(conn, cctx, &channel_open_command, &channel_open_reply,
0);
idch = channel_open_reply.reply.channelopen.idch;
```

The next step is to load the input file (the file to be encrypted) into a file stream (**inputstream**) and prepare the output file stream (**outputstream**) to which the encrypted file is going to be written.

```
inputstream = fopen("file_in.txt", "rb");
outputstream = fopen("file_out.txt", "wb");
```

Now that the application has opened the channel and prepared the input and output streams, start to prepare an **M_Command** to process the **inputstream** through the channel.

```
M_Command channel_process_stream_command;
M_Reply channel_process_stream_reply;
M_Status channel_process_stream_rc;
int eof = 0;
unsigned char buffer[6144];
size_t bytes_read;
```

Next, read the bytes of the **inputstream** into a char buffer, updating the channel on each read.

```
do {
  bytes_read = fread(buffer, 1, sizeof buffer, inputstream);
  if(ferror(inputstream)) {
    fprintf(stderr, "error reading from %s: %s\n",
            input_path, strerror(errno));
    rc = -1;
    goto cleanup;
  }

  if(feof(inputstream))
    eof = 1;

  command.cmd = Cmd_ChannelUpdate;
  if(eof)
    command.args.channelupdate.flags |= Cmd_ChannelUpdate_Args_flags_final;
  command.args.channelupdate.idch = idch;
  command.args.channelupdate.input.ptr = buffer;
  command.args.channelupdate.input.len = (M_Word)bytes_read;

  rc = NFastApp_Transact(conn, cctx, &command, &reply, 0);
  rc = check_reply(rc, 0, "Cmd_ChannelUpdate");
  if(rc)
    goto cleanup;

  if(reply.reply.channelupdate.output.len) {
    if(outputstream) {
      fwrite(reply.reply.channelupdate.output.ptr,
             1, reply.reply.channelupdate.output.len,
             outputstream);
      /* Check for a write error */
      if(ferror(outputstream)) {
        fprintf(stderr, "error writing to %s: %s\n",
                output_path, strerror(errno));
        rc = -1;
        NFastApp_Free_Reply(app, cctx, 0, &reply);
        goto cleanup;
      }
    }
    if(outputdstr) {
      if(nf_dstr_putm(outputdstr, reply.reply.channelupdate.output.ptr,
                      reply.reply.channelupdate.output.len)) {
        fprintf(stderr, "error writing to dstr: %s\n", strerror(errno));
        rc = -1;
        goto cleanup;
      }
    }
  }

  NFastApp_Free_Reply(app, cctx, 0, &reply);
  memset(&reply, 0, sizeof reply);
} while(!eof);
```

If the file was successfully encrypted, save the file to disk:

```
if(channel_process_stream_reply.reply.channelupdate.output.len) {
  if(outputstream) {
    fwrite(channel_process_stream_reply.reply.channelupdate.output.ptr,
           1, channel_process_stream_reply.reply.channelupdate.output.len,
           outputstream);
    writefile(ciphertext_path,
              reply->reply.encrypt.cipher.data.generic128.cipher.ptr,
              reply->reply.encrypt.cipher.data.generic128.cipher.len);
  }
}
```

The final step is to free memory and close the **`outputstream`**.

```
NFastApp_Free_Reply(app, cctx, 0, &reply);
memset(&reply, 0, sizeof reply);
fclose(outputstream);
```

Cleaning up resources

Memory leaks and objects left in memory constitute a security risk. The following code removes all sensitive information from memory and cleanly shuts down the connection to the hardserver.

```
free(sigbytes.ptr);
if(keyid) NFKM_cmd_destroy(app, conn, 0, keyid, "keyid", cctx);
if(idch) NFKM_cmd_destroy(app, conn, 0, idch, "idch", cctx);
NFastApp_Free_Reply(app, cctx, 0, &reply);
if(rqcard_fips_initialized) RQCard_fips_free(&rqcard, &rqcard_fips);
if(rqcard_initialized) RQCard_destroy(&rqcard);
NFKM_freekey(app, keyinfo, cctx);
NFKM_freeinfo(app, &worldinfo, cctx);
if(conn) NFastApp_Disconnect(conn, cctx);
NFastApp_Finish(app, cctx);
if(inputstream) fclose(inputstream);
if(outputstream) fclose(outputstream);
```

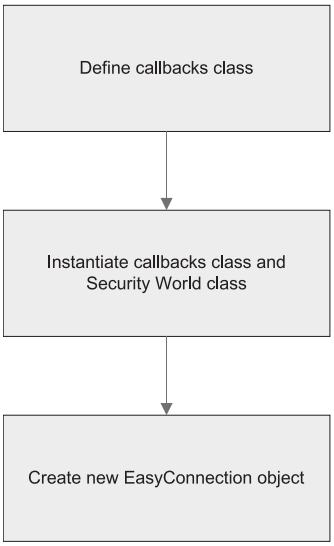
Chapter 4: Java tutorial

Overview

This overview section provides a description of how to achieve two fundamental nCore API programming tasks: connecting to the hardserver and transacting a command. These two tasks are common to almost all cryptographic applications. The rest of this chapter works through a simple example of a basic cryptographic application.

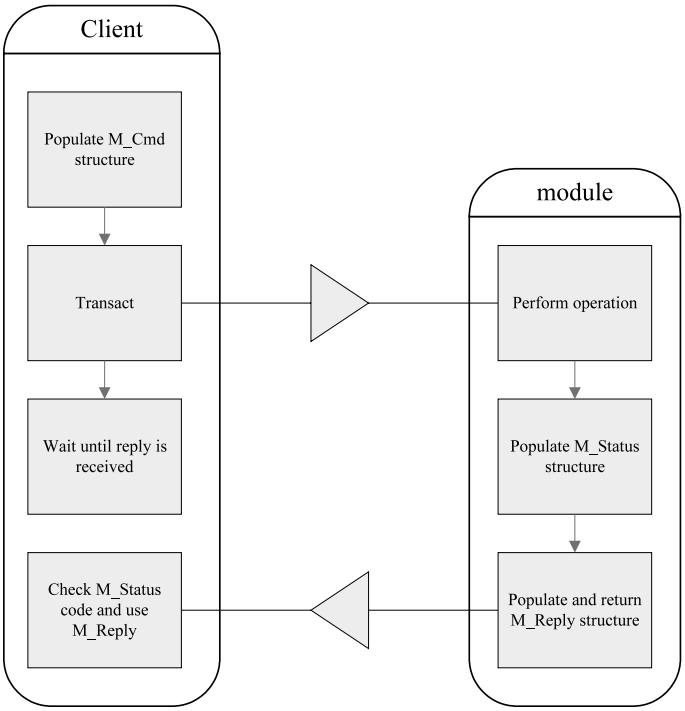
All applications that require nCore functionality will first need to create a connection to a hardserver running on a nShield module. Figure 11 illustrates the steps required to create a connection to a hardserver running on Thales hardware:

Figure 11. Connecting to the hardserver



Once connected to the hardserver, an application can send an `M_Command` to a module. The module processes the command and then returns the results along with any relevant error and status codes. Figure 12 illustrates the process of transacting a cryptographic operation with a module.

Figure 12. Transacting a cryptographic operation



The **M_Reply** structure contains the results of the operation and an **M_Status** message that indicates the outcome of the operation. If a problem is encountered, the **M_Status** value gives an indication of what went wrong. The **M_Reply** contains the results of the command, for example, a key handle or the bytes of an encrypted file.

Creating a softcard

This tutorial demonstrates how to protect a key using a softcard. Use the command line utility **ppmk** to create a softcard in a manner similar to the following:

In a terminal window, type:

```
ppmk --new --non-recoverable WorkedExampleSoftcard
```

ppmk prompts you to provide a pass phrase. Type a pass phrase and press `Enter`.

ppmk prompts you to confirm the pass phrase you have entered. Type the pass phrase again to confirm it, and press `Enter`.

nCore classes used in this tutorial

This tutorial describes some of the functionality in the following nCore classes. You may find it useful to familiarize yourself with these classes by reading the API documentation, which can be found at *nfast_dir*/**java/docs/index.html**.

com.ncipher.km.nfkm.*

Security World classes.

`com.ncipher.km.marshall.*`

Marshals Security World objects.

`com.ncipher.jutils.*`

Various utility classes provided by Thales.

`com.ncipher.nfast.*`

More utility classes.

`com.ncipher.nfast.marshall.*`

Classes which represent nCore commands and related data structures, and which can be used to marshal and unmarshal them from the nShield byte stream format for transmission.

`com.ncipher.nfast.connect.utils.*`

Connection and Channel utility classes.

The code in this chapter also uses two connection utility classes, **`Channel`** and **`EasyConnection`**. The source code for these examples can be found at *nfast_install_directory***`java\examples\connutils`**.

Variables used in this tutorial

The following table lists and describes the variables used in this tutorial. You may also find it useful to view the API documentation of these classes.

Variable name	Variable type	Variable description
`kid`	**`M_KeyID`**	Public key ID
`c`	**`EasyConnection`**	Connection to the hardserver
`wcb`	**`WorldCallbacks`**	Callback object which defines how user interaction is handled
`world`	**`SecurityWorld`**	Security World object
`appname`	**`String`**	Application name
`ident`	**`String`**	Key identity
`type`	**`String`**	Key type
`size`	**`int`**	Key size in bytes
`chanmech`	**`int`**	Cryptographic mechanism used by the secure channel
`chanop`	**`int`**	Secure channel ID
`iv`	**`M_IV`**	Initialization vector
`ch`	**`Channel`**	Secure channel object
`softcard`	**`SoftCard`**	Softcard object

Before connecting to the hardserver

The **WorldCallbacks** class defines how the hardserver interacts with the user when obtaining authorization to create or use a key. The **WorldCallbacks** class extends the **DefaultCallBack** class to customize how the user will be prompted to enter a softcard pass phrase. An instance of this class is used as a parameter when instantiating a **SecurityWorld** object. If you do not pass an instance of a similar class the behavior defined in the **DefaultCallBack** class is used.

```
class WorldCallbacks extends DefaultCallBack {
  public SoftCard configured_softcard = null;
  public String reqPPCallBack(String ReqPPAction) throws NFException {
    try {
      return Passphrase.readPassphrase("Enter softcard pass phrase: ");
    } catch(IOException e) {
      throw new NFException(e.toString());
    }
  }
  // Callback to choose a softcard
  public SoftCard getSoftCardCallback() throws NFException {
    return configured_softcard;
  };
};
```

Before connecting to the hardserver, instantiate a **WorldCallBacks** object and a **SecurityWorld** object as follows:

```
WorldCallbacks wcb = new WorldCallbacks();
SecurityWorld world = new SecurityWorld(null, wcb,
                                        null,
                                        true);
```

Connecting to the hardserver

The following code creates the connection to the hardserver using the **EasyConnection** utility class constructor to wrap an **NFConnection** object:

```
c = new EasyConnection(world.getConnection());
```

Generating a key

The first step is to specify the parameters of a key that can be used to sign a file. In this case we choose to generate a DSA key. We specify the key-generation parameters as follows:

```
appname = "simple";
ident = "worked-example-sign";
type = "DSA";
size = 1024;
chanmech = M_Mech.SHA1Hash;
sigmech = M_Mech.DSA;
iv = new M_IV();
chanop = M_ChannelMode.Sign;
```

Before attempting to generate a key, use the **getKey()** method of the **SecurityWorld** class to check if a key with the given **appname** and **ident** already exists. The **getKey()** method returns **null** if it cannot find the specified key.

```
Key k = world.getKey(appname, ident);
```

If **getKey()** returns **null** this example attempts to generate a key. If no softcard has been named to protect this key, the key is protected using module protection.

```
if(k == null) {
  if(softcard_name != "") {
    k = generate_key(wcb, world, type, size,
                     NFKM_Key_flags.f_ProtectionPassPhrase,
                     softcard_name,
                     appname, ident);
  } else {
     k = generate_key(wcb, world, type, size,
                      NFKM_Key_flags.f_ProtectionModule,
                      null,
                      appname, ident);
  }
}
```

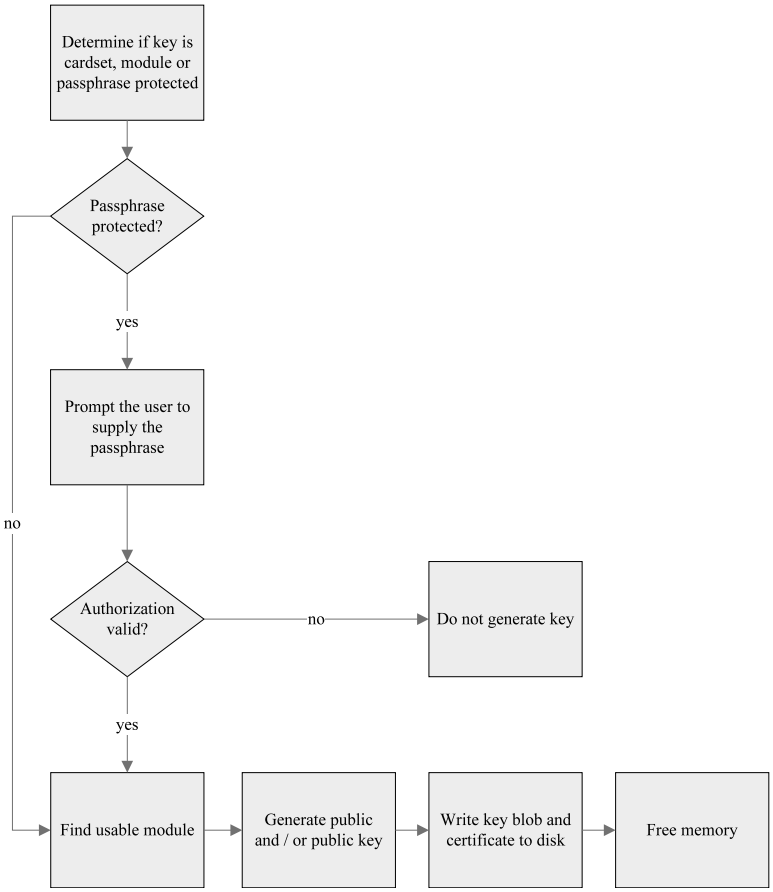
generate_key() is a utility function written specifically for this example. **generate_key()** uses an **AppKeyGenerator** object which is obtained by calling the **getAppKeyGenerator()** method of the **SecurityWorld** object.

The **AppKeyGenerator** class requires a **AppKeyGenProperty[]** array which contains the parameters that specify the key you want to generate. If a key cannot be generated using the specified parameters, **AppKeyGenerator** throws an **nfkmInvalidPropValuesException**. You can call the **check()** method to test whether the **AppKeyGenProperty[]** contains valid values. The properties themselves differ according to your Security World configuration.

The **generate_key** method uses two utility functions written specifically for this tutorial, **setStringProperty()** and **setMenuProperty()**, which are used to set the **AppKeyGenProperty[]** array.

Figure 13 demonstrates the process of generating a key:

Figure 13. Generating a key



Note: This tutorial does not cover details of ACL generation.

The parameters of the `generate_key()` function are:

Parameter name	Parameter Type	Parameter description
`wcb`	`WorldCallbacks`	Callback class that defines user interaction behavior.
`world`	`SecurityWorld`	Contains information about the Security World you are using.
`type`	`String`	The type of key, for example, AES, RSA, DSA.
`len`	`int`	The length of the key you want to generate, in bits.
`protection`	`int`	The type of key protection to be used. This can be any of the flags defined in NFKM_Key_ flags
`prot_name`	`String`	The name of the softcard / module / card that is used to protect the key you want to generate.

Parameter name	Parameter Type	Parameter description
appname	**String**	The name of the application that is requesting that a key is generated. The key name is formed by a combination of the **appname** and the **ident**.
ident	**String**	An arbitrary string that becomes part of the key name. The key name is formed by a combination of the **appname** and the **ident**.

The first step is to obtain an **AppKeyGenerator** object from the **SecurityWorld** object:

```
AppKeyGenerator akg = world.getAppKeyGenerator(appname);
```

Next, as a safety measure we check that all the required key properties are supported by this **AppKeyGenerator** object. In this example, the most likely reason that required key properties are not supported is that no softcard which can be used to protect the key to be generated exists in the Security World:

```
String[] properties = new String[] {
  "ident",
  "type",
  "size",
  "protect"
};
for (int i = 0; i < properties.length; i++) {
  if (akg.getProperty(properties[i]) == null) {
    System.out.println("Property " + properties[i] + " does not exist." +
      "Does your security world contain a usable softcard?");
    System.exit(0);
  }
}
```

If all properties exist, populate the **AppKeyGenProperty[]** using the **setStringProperty()** and **setMenuProperty()** functions. The **protect** property is set, dependent on how the key is to be protected. This example expects the key to be softcard protected. Failing that the example defaults to module protection. Card set protection is not supported in this example.

```
setStringProperty(akg, "ident", ident);
setMenuProperty(akg, "type", type);
setStringProperty(akg, "size", Integer.toString(len));
switch(protection) {
case NFKM_Key_flags.f_ProtectionModule:
  setMenuProperty(akg, "protect", "module");
  break;
case NFKM_Key_flags.f_ProtectionPassPhrase:
  setMenuProperty(akg, "protect", "softcard");
  SoftCard cards[] = world.getSoftCards();
  wcb.configured_softcard = null;
  for(int n = 0; n < cards.length; ++n) {
    if(cards[n].getName().equals(prot_name)) {
      wcb.configured_softcard = cards[n];
    }
    if(wcb.configured_softcard == null) {
      throw new NoSuchSoftCard(prot_name);
      break;
    }
  }
}
```

Before calling the **`generate()`** function of the **`AppKeyGenerator`** class to generate the key, it is good practice to check that the values assigned to the properties are valid. If the properties are valid, call the **`generate()`** function, which returns a reference to the newly created key:

```
InvalidPropValue badprops[] = akg.check();
if(badprops.length > 0) {
  throw new BadKeyGenProperties(badprops);
}
return akg.generate(getUsableModule(world), null);
```

Finally, call the **`cancel()`** method to destroy key information that is resident in memory.

```
akg.cancel();
```

Methods used in generate_key()

The **`getUsableModule()`** method was written for the purposes of this example and simply cycles through all the modules in the Security World until it finds one that is suitable:

```
public static Module getUsableModule(SecurityWorld world)
  throws NFException {
  Module modules[] = world.getModules();
  for(int m = 0; m < modules.length; ++m)
    if(modules[m].isUsable())
      return modules[m];
  throw new NoUsableModules();
}
```

To select a specific module, use the **`getModule()`** function of the **`SecurityWorld`** class. The **`getModule()`** function is overloaded to accept either a module number or a module Electronic Serial Number (ESN) as a parameter.

The **`setStringProperty()`** method was written for the purposes of this example and sets a string property.

```
public static void setStringProperty(AppKeyGenerator akg,
                                      String propname,
                                      String propvalue)
  throws NFException {
  PropValueString pvs = (PropValueString)akg.getProperty(propname).getValue();
  pvs.value = propvalue;
}
```

The **`setMenuProperty()`** method was written for the purposes of this example and sets a menu property.

```
public static void setMenuProperty(AppKeyGenerator akg,
                                   String propname,
                                   String propvalue)
                                   throws NFException {
  PropValueMenu pvm = (PropValueMenu)akg.getProperty(propname).getValue();
  MenuOption options[] = pvm.getOptions();
  for(int i = 0; i < options.length; ++i)
    if(options[i].getName().equals(propvalue)) {
      pvm.value = i;
      return;
    }
  }
  throw new InvalidMenuItem(propvalue);
}
```

Using a key

Before using a key the key must be loaded onto a module. In this example we expect the key being loaded to be softcard protected, or failing that, module protected.

```
Module module = getUsableModule(world);
SoftCard softcard = k.getSoftCard();
if(softcard != null) {
  softcard.load(module, wcb);
  kid = k.load(softcard, module);
} else {
  kid = k.load(module);
}
```

Signing a file

Now that the key is loaded onto the module, open a secure channel to use to sign a text file.

```
Channel ch = c.openChannel(chanop, kid, chanmech, iv, true, true);
```

The **openChannel()** method of the **EasyConnection** class returns a subclassed **Channel** object. For this example, the **openChannel()** function transacts an **M_Cmd.ChannelOpen** command and uses the **M_Cmd_Reply_ChannelOpen** object returned in the reply to instantiate and then return a **Channel.Sign** object.

```
M_Cmd_Args_ChannelOpen args = new M_Cmd_Args_ChannelOpen(
  new M_ModuleID(0), M_ChannelType.Simple, 0, how, mech);
if (!keyless) {
    args.set_key(key);
}

if (!generateIV) {
  args.set_given_iv(given_iv);
}
M_Reply rep = transactChecked(new M_Command(M_Cmd.ChannelOpen, 0,args));
M_Cmd_Reply_ChannelOpen corep = (M_Cmd_Reply_ChannelOpen) rep.reply;
if ( 0 != (corep.flags & corep.flags_new_iv) ) {
  given_iv.mech = corep.new_iv.mech;
  given_iv.iv = corep.new_iv.iv;
}
return new Channel.Sign(mech, key, corep.new_iv, corep.idch, this);
```

Channel.Sign extends the abstract **Channel** class. The **update()** function reads the specified **byte[]** into the channel. The **updateFinal()** method reads the specified byte array into the channel, but should only be called when reading the final **byte[]** array that you want to process through the channel.

```
public static class Sign extends Channel {
  public Sign(long mech, M_KeyID keyID, M_IV iv, M_KeyID channelID, EasyConnection parent) {
    super(M_ChannelMode.Sign, mech, keyID, iv,channelID, parent);
  }
  public void update(byte[] input) throws MarshallTypeError,
                                          CommandTooBig,
                                          ClientException,
                                          ConnectionClosed,
                                          StatusNotOK {
    super.update(input, false, false);
  }
  public byte[] updateFinal(byte[] input) throws MarshallTypeError,
                                                 CommandTooBig,
                                                 ClientException,
                                                 ConnectionClosed,
                                                 StatusNotOK {
    return super.update(input, true, false);
  }
}
```

Now that the signing channel is open, open the input file to be signed, and a **FileOutputStream** for the signature.

```
FileInputStream input = null;
FileOutputStream output = null;
input = new FileInputStream(plaintext_path);
```

Finally, use the channel to read in the input file bytes:

```
byte inputbytes[] = new byte[4096];
int len = input.read(inputbytes);
while(len != -1) {
  byte outputbytes[] = ch.update(arrayTruncate(inputbytes, len),
                                 false,
                                 false);
  if(output != null)
    output.write(outputbytes);
    len = input.read(inputbytes);
  }
}
byte outputbytes[] = ch.update(new byte[0],
                               true,
                               false);
```

The **`arrayTruncate()`** function was written specifically for this example, and ensures that the **`byte[]`** used to update the channel is consistently chunked.

```
static byte[] arrayTruncate(byte[] in, int len) {
  byte out[] = new byte[len];
  for(int i = 0; i < len; ++i)
    out[i] = in[i];
  return out;
}
```

Next, create the **`hash`** and **`plaintext`** objects.

```
hash = new M_Hash(outputbytes);
plaintext = new M_PlainText(M_PlainTextType.Hash,
                            new M_PlainTextType_Data_Hash(hash));
```

Transact an **`M_Cmd.Sign`** operation to sign the hashed plaintext:

```
cmd = new M_Command(M_Cmd.Sign,
                    0,
                    new M_Cmd_Args_Sign(0,
                                        kid,
                                        sigmech,
                                        plaintext));
try {
  reply = c.transactChecked(cmd);
} catch (StatusNotOK sno) {
  System.exit(0);
}
```

If the **M_Cmd.Sign** operation succeeded, marshal the **signature** to a stream of bytes, and saves the bytes as a signature file:

```
signature = ((M_Cmd_Reply_Sign)reply.reply).sig;
MarshallContext mc = new MarshallContext();
signature.marshall(mc);
output = new FileOutputStream(signature_path);
output.write(mc.getBytes());
if(output != null) output.close();
```

Cleaning up resources

Finally, unload the keys in the module memory.

```
if(kid != null) c.destroy(kid);
if(pubkid != null) c.destroy(pubkid);
```

Appendix A: Java examples

The example programs and source code described in this section are supplied on your Developer installation media. Several of the utilities are not designed to be executed directly but are used by other programs. For more information on these examples, see the in-line comments in the example source code and the Javadocs installed in your `nfast` directory.

The Java example files for the Java examples can be found in subdirectories of:

- Windows: *%NFAST_HOME%*`\java\`
- Unix-based: `/opt/nfast/java/`

Java key management example utilities (kmjava)

AppKeyGen.java

This example utility demonstrates application key generation and import.

GenerateExport.java

This example utility generates an RSA Key and optionally exports the public key out of a module as plain text.

It demonstrates the creation of an OCS.

KMJavaFloodTest.java

This example utility demonstrates the use of the `mergeKeyIDs` method in the `Key` class.

This method merges all the loaded private `keyid`s into a single `keyid` that can be used in nCore API calls when load-sharing is required.

NFKMInfo.java

Displays information about the Security World.

This example Java utility is analogous to its C version except that `NFKMInfo.java` does not return information on world/module generation.

NVRamRTCUtil.java

This is an example program to demonstrate interacting with the NVRAM and RTC. The program allows you to list all files in NVRAM, delete a file in NVRAM, delete all the files in NVRAM, display the current time in the RTC and to set the RTC to the system clock.

SimpleCrypt.java

This is a simple example that graphically encrypts and decrypts data with a Triple-DES (DES3) key from the Security World. Cipher Block Chaining mode (CBC) and initialization vectors are selected randomly. This information is prefixed to the cipher text.

`SimpleCrypt.java` only works with module protected Triple-DES (DES3) keys.

SlotPoller.java

This example utility polls all the available slots.

You can determine whether the state of the slot has changed by calling **`getIC()`** on the slot. This method is more efficient than using **`update()`**. The module serial number, slot number, and insertion count are displayed when a card is inserted or removed.

Java JCE/CSP example utilities (jcecsp)

AsymmetricEncryptionExample.java

This example generates an RSA key pair and an X509 public key specification. It performs encryption and decryption of random plain text.

ECDHExample.java

This example utility demonstrates:

- creation of an ECDH key
- ECDH key agreement handshake
- Encryption / decryption of a message using AES.

JCEChanTest.java

This example measures the data rate achieved by different symmetric encryption and decryption operations. You can use optional program arguments to change the **`cipher`**, **`key`**, **`data`**, and **`provider`** parameters.

JCEFloodTest.java

This example utility does performance testing for RSA, DSA and ECDSA private key operations.

It demonstrates:

- RSA/DSA/ECDSA Key Pair generation
- RSA/DSA/ECDSA signing
- RSA encryption/decryption
- use of the **`kmjava`** classes to load a key to use with the nCipherKM JCE provider
- load-balancing using **`kmjava`** and KeyStore-loaded keys.

JCESigTest.java

This example measures the data rate achieved by many threads simultaneously performing signing and verifying operations. You can use optional program arguments to change the **`thread`**, **`key`**, **`data`**, **`provider`**, and **`sampling`** parameters.

KeyLoadTimer.java

This example measures the time taken to get many keys from an **`nCipher.sworld`** key store. It also demonstrates how to create, load and store key stores, as well as how to set and get key entries.

KeyStorageExample.java

This example creates a new KeyStore containing an AES key. It performs load-balanced encryption and decryption of random plain text using a KeyStore loaded key.

NCipherLibraryInteropExample.java

This example loads an existing AES key from the Security World across all usable modules and performs load-balanced encryption and decryption of random plain text.

PrepareSslExamples.java

This example creates a KeyStore containing the trusted certificates installed in the local host VM.

Running this example is a prerequisite for running the **`SslClientExample.java`** example. For more information, see *SslClientExample.java* on page 59

PrepareSSLServerExamples.sh

This example creates a KeyStore.

Running this shell script (or the command line within it) is a prerequisite for running the **`SslServerExample.java`** example. For more information, see *SslServerExample.java* on page 60.

SignaturesExample.java

This example generates an RSA key pair with which it performs signing and verification of random plain text.

SslClientExample.java

Before building this example, the user will need to edit **`SslClientExample.java`** to insert an appropriate **`https`** web site address in the two relevant places. When run, this example connects to the user-specified secure web site over an encrypted SSL connection and dumps the index page to the console.

Before running this example, you must run **`PrepareSslExamples.java`**. For more information, see *PrepareSslExamples.java* on page 59

SslServerExample.java

This example creates a simple SSL Web server instance on the local host that can be accessed with a Web browser.

Before running this example, you must run **`PrepareSslExamples.java`**. For more information, see *PrepareSslExamples.java* on page 59

SymmetricEncryptionExample.java

This example generates symmetric keys and uses them to perform encryption and decryption of random plain text with different cipher modes and padding types.

SignatureTest.java

This example utility demonstrates:

- generation of an RSA/DSA/ECDSA Key Pair
- export of the **`PublicKey`** using X509 encoding
- signing some random data
- decoding the **`PublicKey`**
- verification of the signature.

Java generic stub examples (nfjava)

Note: The example utilities described in this section are directly analogous to their namesake C example utilities supplied with the nShield C generic stub. The Java incarnations are shipped as source code only.

BlobInfo.java

This example utility displays information in a blob. It demonstrates how to determine information about the contents of a blob.

`BlobInfo.java` is analogous to the C Generic Stub call **`NFast_ExamineBlob`**.

Channel.java

This example utility is a function-based wrapper to symmetric bulk-encryption channels for use by **`EasyConnection.java`**.

CheckMod.java

This example utility checks modulo-exponentiation operations against a test file.

CrypTest.java

This example utility is a test program for some module algorithms.

It demonstrates:

- the use of **`EasyConnection`**
- symmetric cryptography and channels.

DesKat.java

This example utility is for DES known answer tests.

It demonstrates simple nCore key management usage.

DKTest.java

This example utility provides a simple demonstration of the use of **`DeriveKey`**.

EasyConnection.java

This example utility is a function-based interface to a subset of nCore.

Enquiry.java

This example utility displays enquiry information.

It demonstrates:

- simple nCore usage
- the **`enquiry`** command.

FloodTest.java

This example utility does performance testing for modexp code.

It demonstrates:

- simple bignum usage
- asynchronous command processing (**`NFastApp_Wait`** and **`NFastApp_Query`**).

GenCert.java

This example utility generates a certificate.

It demonstrates the use of the **`BuildCmdCert`** class.

InitUnit.java

This example utility initializes a module with a dummy **`HKNSO`** (like the C **`initunit`** utility).

NFEnum.java

This example utility is a helper class used by **`SigTest`**. It is an example extension to **`jnfopt`** for looking up an nCore Enumeration class. It cannot be invoked by itself.

Option.java

This file is part of **`jnfopt`**, the standard nShield java options parser.

It forms part of the **jnfopt** example library and cannot be invoked by itself. It is shipped as part of **jutils.jar** because some nShield utilities rely on it.

ParseException.java

This file is part of **jnfopt**, the standard nShield java options parser.

It forms part of the **jnfopt** example library and cannot be invoked by itself. It is shipped as part of **jutils.jar** because some nShield utilities rely on it.

Parser.java

This file is part of **jnfopt**, the standard nShield java options parser.

It forms part of the **jnfopt** example library and cannot be invoked by itself. It is shipped as part of **jutils.jar** because some nShield utilities rely on it.

Reference.java

This file is part of **jnfopt**, the standard nShield java options parser.

It forms part of the **jnfopt** example library and cannot be invoked by itself. It is shipped as part of **jutils.jar** because some nShield utilities rely on it.

ReportVersion.java

This example utility reports the embedded version information from the current **nfjava** component. **ReportVersion.java** outputs the version of the **nfjava** library found on the class path.

These examples are not intended to be invoked directly. They are called by other programs. The following two utilities, **EasyConnection** and **Channel**, form a Java analog of the nCore simple command functions as shipped to C developers in **libexamples.a**. You can compare and contrast this example with the C example **simplecmd.h**.

You cannot invoke **EasyConnection** and **Channel** directly; **CrypTest** invokes them. For more information, see the Javadoc documentation.

ScoreKeeper.java

This example utility is shared code used by **SigTest** and **FloodTest** and cannot be invoked on its own. It has helper classes for output reporting by **SigTest** and **FloodTest**.

SigTest.java

This example utility does signature performance testing.

It demonstrates asynchronous command processing (**NFastApp_Wait** and **NFastApp_Query**).

Note: Java is not a high-performance language. On slow host systems or systems with multiple modules, it is very common to be limited by the CPU of the host machine. As a result, this example often does not show the true performance capabilities of the module. If you want to

test module performance, as distinct from application performance, use the C version of **`SigTest`** instead.

Appendix B: Key structures

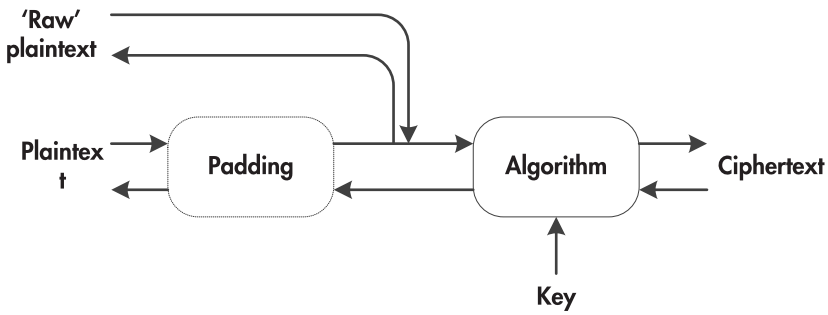
This chapter describes the data structures used by the Thales module to represent keys and their ACLs. It includes information about:

- **mechanisms** which are the combination of algorithm, padding, and mode that are used to transform plain text into cipher text or cipher text into plain text.
- **plain texts** which are the messages being processed. This chapter lists the plain text formats that are supported by the Thales module.
- **keys** which are the secret and public values used in an algorithm. The section of this chapter about keys describes:
  - the format for each key type
  - the mechanisms supported for that key type
  - the parameters required to generate a key or key pair of this type.
- **hash functions** which return a fixed-length string from arbitrary-length input. Hash functions can be used to identify a document without revealing its contents.
- **Access Control Lists (ACLs)** which describe the actions that can be performed with a specific key. This chapter describes the format of an ACL.
- **certificates** which are used to authorize actions on keys.

Mechanisms

A mechanism is a combination of padding, algorithm, mode, and so forth, which, together with a key, transforms a plaintext into a ciphertext (or a ciphertext into a plaintext).

Figure 14. Mechanisms



Each mechanism has a single ciphertext format represented by `M_CipherText`, a tagged union type for which the tag is an `M_Mech`. A mechanism may accept or generate various different plain text formats. The details of the padding and other processing may vary depending on the plain text format supplied or requested.

Note: Mechanisms with similar forms share the same member name in this union. For example, the 64-bit block ciphers all use Mech_Generic64 .

```
union M_Mech__Cipher {
     M_Mech_SHA384Hash_Cipher sha384hash;
     M_Mech_DSA_Cipher dsa;
     M_Mech_TLSFinishedMsg_Cipher tlsfinishedmsg;
     M_Mech_SHA256Hash_Cipher sha256hash;
     M_Mech_DLIESe3DEShSHA1_Cipher dliese3deshsha1;
     M_Mech_TigerHash_Cipher tigerhash;
     M_Mech_DHKeyExchange_Cipher dhkeyexchange;
     M_Mech_HAS160Hash_Cipher has160hash;
     M_Mech_ECDHKeyExchange_Cipher ecdhkeyexchange;
     M_Mech_SSL3FinishedMsg_Cipher ssl3finishedmsg;
     M_Mech_RSApPKCS1_Cipher rsappkcs1;
     M_Mech_Imech_Cipher imech;
     M_Mech_ArcFourpNONE_Cipher arcfourpnone;
     M_Mech_Generic256MAC_Cipher generic256mac;
     M_Mech_ElGamal_Cipher elgamal;
     M_Mech_RSApPKCS1pPKCS11_Cipher rsappkcs1ppkcs11;
     M_Mech_BlobCrypt_Cipher blobcrypt;
     M_Mech_Generic128_Cipher generic128;
     M_Mech_Generic192MAC_Cipher generic192mac;
     M_Mech_ECDSA_Cipher ecdsa;
     M_Mech_Generic64_Cipher generic64;
     M_Mech_SHA512Hash_Cipher sha512hash;
     M_Mech_SHA224Hash_Cipher sha224hash;
     M_Mech_Generic256_Cipher generic256;
     M_Mech_SSLRecordLayer_Cipher sslrecordlayer;
     M_Mech_Generic192_Cipher generic192;
     M_Mech_KCDSAHAS160_Cipher kcdsahas160;
     M_Mech_Generic64MAC_Cipher generic64mac;
     M_Mech_GenericGCM128_Cipher genericgcm128;
     M_Mech_RIPEMD160Hash_Cipher ripemd160hash;
     M_Mech_Generic128MAC_Cipher generic128mac;
     M_Mech_MD5Hash_Cipher md5hash;
     M_Mech_SHA1Hash_Cipher sha1hash;
};
```

Mech_Any

Instead of explicitly specifying a mechanism, you can let the module select the mechanism by specifying **Mech_Any** . The Thales module selects the mechanism as follows:

- for decryption or signature verification, the module uses the mechanism that is defined in the cipher text
- for encryption or signature generation, the module selects an appropriate mechanism based on the key type and the operation as listed in the following table.

Key Type	Encryption mechanism	Signing mechanism
RSAPublic	**Mech_RSApPKCS10AEP**, **Mech_RSApPKCS10AEPhSHA224**, **Mech_RSApPKCS10AEPhSHA256**, **Mech_RSApPKCS10AEPhSHA384** or **Mech_RSApPKCS10AEPhSHA512** chosen based on size of key.	

Key Type	Encryption mechanism	Signing mechanism
RSAPrivate		**Mech_RSAhSHA1pPSS**, **Mech_RSAhSHA224pPSS**, **Mech_RSAhSHA256pPSS**, **Mech_RSAhSHA384pPSS** or **Mech_RSAhSHA512pPSS** chosen based on size of key.
DHPublic	**Mech_ElGamal**	
DSAPrivate		**Mech_DSA**, **Mech_DSAhSHA224**, **Mech_DSAhSHA256**, **Mech_DSAhSHA384**, or **Mech_DSAhSHA512** chosen based on size of key.
ECDSAPrivate		**Mech_ECDSA**, **Mech_ECDSAhSHA224**, **Mech_ECDSAhSHA256**, **Mech_ECDSAhSHA384**, or **Mech_ECDSAhSHA512** chosen based on size of key.
DES *(not available in strict FIPS operational mode)*	**Mech_DESmCBCi64pPKCS5**	**Mech_DESmCBCMACi0pPKCS5**
DES2	**Mech_DES2mCBCi64pPKCS5**	**Mech_DES2mCBCMACi0pPKCS5**
DES3	**Mech_DES3mCBCi64pPKCS5**	**Mech_DES3mCBCMACi0pPKCS5**
CAST	**Mech_CASTmCBCi64pPKCS5**	**Mech_CASTmCBCMACi0pPKCS5**
CAST256	**Mech_CAST256mCBCi128pPKCS5**	**Mech_CAST256mCBCMACi0pPKCS5**
ArcFour	**Mech_ArcFourpNone**	
Rijndael	**Mech_RijndaelmCBCi128pPKCS5**	**Mech_RijndaelmCBCMACi0pPKCS5**
Blowfish	**Mech_BlowfishmCBCi64pPKCS5**	**Mech_BlowfishmCBCMACi0pPKCS5**
Twofish	**Mech_TwofishmCBCi128pPKCS5**	**Mech_TwofishmCBCMACi0pPKCS5**
Serpent	**Mech_SerpentmCBCi128pPKCS5**	**Mech_SerpentmCBCMACi0pPKCS5**

Key Types

The following sections list the keys types for the different algorithms and mechanisms that are supported by the Thales module. The table below shows which mechanisms are supported by which key types.

Key type	Block size	Encrypt	Decrypt	Sign	Verify
ArcFour	N/A	Y	Y	-	-
Blowfish	64				
		Y	Y	-	-
CBC MAC		-	-	Y	Y

Key type	Block size	Encrypt	Decrypt	Sign	Verify
ECB		Y	Y	-	-
CAST	64				
CBC		Y	Y	-	-
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
Cast256	128				
CBC		Y	Y	-	-
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
DES	64				
CBC		Y	Y	-	-
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
DES2	64				
CBC		Y	Y	-	-
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
Triple DES	64				
CBC		Y	Y	-	-
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
SEED	128				
CBC		Y	Y	-	-
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
Serpent	128				
CBC		Y	Y	-	-
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
SSLMasterSecret					
RecordLayer		Y	Y	-	-
FinishedMessage		-	-	Y	Y
Rijndael	128				
CBC		Y	Y	-	-

Key type	Block size	Encrypt	Decrypt	Sign	Verify
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
GCM		Y	Y	-	-
Twofish	128				
CBC		Y	Y	-	-
CBC MAC		-	-	Y	Y
ECB		Y	Y	-	-
Diffie-Hellman	N/A				
Key Exchange		-	Y	-	-
ElGamal		Y	Y	-	-
DSA	N/A	-	-	Y	Y
ECDSA	N/A	-	-	Y	Y
ECDH	N/A				
Key Exchange		-	Y	-	-
KCDSA		-	-	Y	Y
RSA	N/A	-	-	Y	Y
HMAC	N/A				
HMACMD2		-	-	Y	Y
HMACMD5		-	-	Y	Y
HMACSHA-1		-	-	Y	Y
HMACRIPEMD160		-	-	Y	Y
HMACSHA224		-	-	Y	Y
HMACSHA256		-	-	Y	Y
HMACSHA384		-	-	Y	Y
HMACSHA512		-	-	Y	Y
HMACTiger		-	-	Y	Y
Random	N/A				
Template	N/A	-	-	-	-
Wrapped	N/A	-	-	-	-

For each key type, the tables below list:

- the data that is stored in the key (separately for public and private halves of key pairs)
- the parameters required to generate the key (or key pair):

```
typedef struct {
    M_KeyType type;
    union M_KeyType__Data data;
}     M_PlainText ;

 typedef struct {
    M_KeyType type;
    union M_KeyType__GenParams params;
}     M_KeyGenParams;
```

Note: Key types with similar forms for key data or generation parameters share the same member name in these unions. For example, keys whose data is a single block of random bytes (CAST, ArcFour, Random, HMACMD2, HMACMD5, HMACRIPEMD160, and Wrapped) all use the **`Random`** members of these unions.

Random

Key data

```
typedef struct {
    M_ByteBlock k    data
} M_KeyType_Random_Data;
```

Key generation parameters

```
typedef struct {
    M_Word lenbytes;                          length in bytes
} M_KeyType_Random_GenParams;
```

Notes

The FIPS 46-3 validation requires DES keys to have valid parity bits for which bit 0 of each byte is set to give odd parity. If you attempt to import a Triple DES key that does not have the parity set correctly, the module returns **`Status_InvalidData`**.

ArcFour

This key type is a symmetric algorithm that is compatible with Ron Rivest's RC4 cipher. It uses the key data **`M_KeyType_Random_Data`**.

Key data

```
struct M_Mech_ArcFourpNONE_Cipher {
    M_ByteBlock cipher;                       192-bit key
};
```

Key generation parameters

```
typedef struct {
    M_Word lenbytes;                          length in bytes
} M_KeyType_Random_GenParams;
```

Mechanisms

Mech_ArcFourpNONE

The cipher text is a byte block. This mechanism has no IV.

Blowfish

Blowfish uses the key data **M_KeyType_Random_Data**. The key data length must be at least one byte. The maximum permitted key data length is 56 bytes. Recommended key lengths are 16, 24, 32 and 56 bytes.

Key data

```
typedef struct {
    M_ByteBlock k;                            data
} M_KeyType_Random_Data;
```

Key generation parameters

```
typedef struct {
    M_Word lenbytes;                          length in bytes
} M_KeyType_Random_GenParams;
```

Mechanisms

Mech_BlowfishmECBpNONE	ECB
Mech_BlowfishmCBCpNONE	CBC
Mech_BlowfishmCBCi64PKCS5	CBC
Mech_BlowfishmCBCMACi64PKCS5	CBC MAC *see note*
Mech_BlowfishmECBpPKCS5	ECB
Mech_BlowfishmCBCMACi0PKCS5	CBC MAC

Note: The mechanism **Mech_BlowfishmCBCMACi64PKCS5** is deprecated and may be withdrawn in future firmware.

CAST

This key type uses the key data **M_KeyType_Random_Data**, with a key length from 5 to 16 bytes as specified in RFC2144.

Mechanisms

Mech_CASTmCBCi64pPKCS5	CBC
Mech_CASTmCBCMACi64pPKCS5	*see note*
Mech_CASTmECBpPKCS5	ECB
Mech_CASTmCBCMACi0pPKCS5	CBC MAC

Note: The mechanism **Mech_CASTmCBCMACi64pPKCS5** is deprecated and may be withdrawn in future firmware.

The cipher text and initialization vectors are the same as for the equivalent DES mechanisms.

CAST256

This uses the same key generation parameters and data as **KeyType_Random**, and allows key lengths of 16, 20, 24, 28 or 32 bytes as specified in RFC2612.

Mechanisms

Mech_CAST256mCBCi128pPKCS5	CBC with PKCS #5 padding
Mech_CAST256mECBpPKCS5	ECB with PKCS #5 padding
Mech_CAST256mCBCpNONE	CBC with no padding
Mech_CAST256mECBpNONE	ECB with no padding
Mech_CAST256mCBCMACi128pPKCS5	*see note*
Mech_CAST256mCBCMACi0pPKCS5	CBC MAC

The mechanism **Mech_CAST256mCBCMACi128pPKCS5** is deprecated and may be withdrawn in future versions.

DES

Note: The implementation of DES that is used in the Thales module has been validated by NIST as conforming to FIPS 46-2 and FIPS 81, certificate number 24.

Key data

```
typedef struct {
    M_DESKey k;                                 64 bit key
} M_KeyType_DES_Data;
```

```
typedef union {
    unsigned char bytes[8];
    M_Word words[2];
} M_DESKey;
```

Note: 56 bits plus 8 parity bits

Key generation parameters

```
typedef struct {
    M_Word lenbytes;                          length in bytes
} M_KeyType_Random_GenParams;
```

Notes

The FIPS 46-2 validation requires DES keys to have valid parity bits for which bit 0 of each byte is set to give odd parity. If you attempt to import a DES key that does not have the parity set correctly, the module will return **`Status_InvalidData`** .

Mechanisms

`Mech_DESmCBCpNONE`	CBC no padding
`Mech_DESmCBCi64pPKCS5`	CBC with PKCS5 padding
`Mech_DESmCBCMACi64pPKCS5`	*see note*
`Mech_DESmECBpNONE`	ECB no padding
`Mech_DESmECBpPKCS5`	ECB with PKCS5 padding
`Mech_DESmCBCMACi0pPKCS5`	CBC MAC with PKCS5 padding
`Mech_DESmCBCMACi0pNONE`	CBC MAC with no padding

PKCS5 padding is 1 to 8 bytes, valued 1 to 8

Note: The mechanism **`Mech_DESmCBCMACi64pPKCS5`** is deprecated and may be withdrawn in future versions.

CBC

Cipher text

```
typedef struct {
    M_ByteBlock cipher;
} M_Mech_Generic64_Cipher;
```

IV

```
typedef struct {
    M_Block64 iv;
} M_Mech_Generic64_IV;
```

CBC MAC

Cipher text

```
typedef struct {
    M_Block64 mac;
} M_Mech_Generic64MAC_Cipher;
```

Note: The **DESmCBCMACi0pPKCS5** mechanism uses an **IV** of all zero bytes. This replaces the **DESmCBCMACi64pPKCS5** mechanism, which required the **IV** to be passed in. This mechanism is deprecated: if an attacker is able to manipulate this data he is able to forge a message. For this reason, if you use -i64 mechanisms you must ensure the IV data is fixed.

DES2

Note: The implementation of DES used in the Thales module has been validated by NIST as conforming to FIPS 46-3 certificate numbers 24 and 173.

Key data

```
typedef struct {
    M_DES2Key k;                        128 bit key
} M_KeyType_DES2_Data;
```

```
typedef union {
    unsigned char bytes[16];
    M_Word words[4];
} M_DESKey;
```

Note: 112 bit plus 16 parity bits.

Key generation parameters

There are no key generation parameters.

Notes

The FIPS 46-2 validation requires DES2 keys to have valid parity bits for which bit 0 of each byte is set to give odd parity. If you attempt to import a DES2 key that does not have the parity set correctly, the module will return **Status_InvalidData**.

Mechanisms

Mech_DES2mCBCpNONE	CBC no padding
Mech_DES2mCBCi64pPKCS5	CBC with PKCS5 padding
Mech_DES2mCBCMACi64pPKCS5	*see note*
Mech_DES2mECBpNONE	ECB no padding
Mech_DES2mECBpPKCS5	ECB with PKCS5 padding
Mech_DES2mCBCMACi0pPKCS5	CBCMAC with PKCS5 padding
Mech_DES2mCBCMACi0pNONE	CBC MAC with no padding

Note: The mechanism **Mech_DES2mCBCMACi64pPKCS5** is deprecated and may be withdrawn in future versions.

CBC

Cipher text

```
typedef struct {
    M_ByteBlock cipher;
} M_Mech_Generic64_Cipher;
```

IV

```
typedef struct {
    M_Block64 iv;
} M_Mech_Generic64_IV;
```

Triple DES

Note: The implementation of DES used in the Thales module has been validated by NIST as conforming to FIPS 46-3 certificate numbers 24 and 173.

Key data

```
typedef struct {
    M_DES3Key k                                                    192 bit key
} M_KeyType_DES3_Data;
```

```
typedef union {
    unsigned char bytes[24];
    M_Word words[6];
} M_DES3Key;
```

Note: The key is 3 ×(56+8) bits. nShield performs Triple DES as encrypt, decrypt, and encrypt (using separate keys for each stage).

Key generation parameters

There are no key generation parameters.

Mechanisms

Mech_DES3mCBCi64pPKCS5	CBC with PKCS #5 padding
Mech_DES3mCBCMACi64pPKCS5	*see note*
Mech_DES3mCBCpNONE	CBC with no padding
Mech_DES3mECBpNONE	ECB with no padding
Mech_DES3mECBpPKCS5	ECB with PKCS #5 padding
Mech_DES3mCBCMACi0pPKCS5	CBCMAC with PKCS #5 padding
Mech_DES3mCBCMACi0pNONE	CBC MAC with no padding

Note: The mechanism **Mech_DES3mCBCMACi64pPKCS5** is deprecated and may be withdrawn in future versions.

The cipher text and initialization vectors are the same as for the equivalent DES mechanisms.

Note: nShield uses outer CBC.

Rijndael

Note: Rijndael is now FIPS approved as the AES. The Thales implementation has been validated by NIST as conforming to FIPS 197, certificate number 15.

This key type uses the key data **M_KeyType_Random_Data**.

Mechanisms

Mech_RijndaelmCBCpNONE	CBC
Mech_RijndaelmCBCi128pPKCS5	CBC with PKCS5 padding
Mech_RijndaelmCBCMACi128pPKCS5	*see note*
Mech_RijndaelmECBpNONE	ECB
Mech_RijndaelmECBpPKCS5	ECB with PKCS5 padding
Mech_RijndaelmCBCMACi128pPKCS5	CBC MAC with PKCS5 padding
Mech_RijndaelmCBCMACi128pNone	CBC MAC with no padding
Mech_RijndaelmGCM	GCM

Note: The mechanism **Mech_RijndaelmCBCMACi128pPKCS5** is deprecated and may be withdrawn in future versions.

These mechanisms use the Generic128 cipher text and initialization vectors, except **Mech_RijndaelmGCM** which uses GenericGCM128.

Key generation

Rijndael keys use the same key generation parameters and data format as the Random key type. They must be either 128, 192, or 256 bits (that is, 16, 24 or 32 bytes long).

SEED

The SEED algorithm was developed by KISA (Korea Information Security Agency) and a group of experts. SEED is a Korean national industrial association standard (TTA KO-12.0004, 1999) and was set as a Korean Information Communication Standard (KICS) in the year 2000. This standard is promoted by the Korean Ministry of Information and Communication.

SEED has been optimized for the security systems most widely used in Korea, in particular the S-boxes and configurations associated with current computing technology.

Note: If you wish to use the SEED algorithm, you must order and enable it as part of the Thales **KISAAlgorithms** feature, as described in the User Guide.

Key data

```
typedef struct {
    M_ByteBlock k;                      fixed-length 128-bit key
} M_KeyType_SEED;
```

Key generation parameters

```
typedef struct {
    M_Word lenbytes;                            must be 16 bytes
} M_KeyType_SEED_GenParams;
```

Mechanisms

Mech_SEEDmECBpNONE	ECB with no padding
Mech_SEEDmECBpPKCS5	ECB with PKCS #5 padding
Mech_SEEDmCBCpNONE	CBC with no padding
Mech_SEEDmCBCi128pPKCS5	CBC with PKCS #5 padding
Mech_SEEDmCBCMACi128pPKCS5	*see note*
Mech_SEEDmCBCMACi0pPKCS5	CBCMAC

Note: The mechanism **Mech_SEEDmCBCMACi128pPKCS5** is deprecated and may be withdrawn in future versions.

Serpent

Serpent uses the key data **M_KeyType_Random_Data**. The maximum permitted key data length is 32 bytes. Recommended key lengths are 16, 24 and 32 bytes.

Note: A change was made to the interpretation of the Serpent algorithm specification regarding byte ordering, which occurred between versions 2.12.x and earlier, and 2.18.x and later, of module firmware. Thus, later versions of firmware are incompatible with earlier versions when using Serpent mechanisms.

Key data

```
typedef struct {
    M_ByteBlock k;                          data
} M_KeyType_Random_Data;
```

Key generation parameters

```
typedef struct {
    M_Word lenbytes;                        length in bytes
} M_KeyType_Random_GenParams;
```

Mechanisms

Mech_SerpentmECBpNONE	ECB with no padding
Mech_SerpentmCBCpNONE	CBC with no padding
Mech_SerpentmCBCi128PKCS5	CBC with PKCS #5 padding
Mech_SerpentmCBCMACi128PKCS5	*see note*
Mech_SerpentmECBpPKCS5	ECB with PKCS #5 padding
Mech_SerpentmCBCMACi0PKCS5	CBCMAC

Note: The mechanism **Mech_SerpentmCBCMACi128PKCS5** is deprecated and may be withdrawn in future versions.

SSLMasterSecret

An SSL Master Secret is used to encrypt messages using the SSL and TLS protocols.

Thales supply a SEE library CodeSafe SSL that supports a complete SSL stack within SEE.

Key data

```
typedef union M_SSLMasterSecret {
    unsigned char bytes[48];
    M_Word words[12];
} M_SSLMasterSecret;
```

Key generation parameters

This key type cannot be generated. It can only be derived using DeriveKey with one of the mechanisms **DeriveMech_SSL3withDH**, **DeriveMech_SSL3withRSA**, **DeriveMech_TLSwithDH**, **DeriveMech_TLSwithRSA**.

Mechanisms

Mech_SSLRecordLayer	*see note 1*
Mech_SSL3FinishedMsg	*see note 2*
Mech_TLSFinishedMsg	*see note 3*

1. Encrypting with **Mech_SSLRecordLayer** requires the following **IV**:

```
struct M_Mech_SSLRecordLayer_IV {
    M_Mech_SSLRecordLayer_IV_flags flags;
    SSLClientRandom crnd;       32-byte Client Random field
    M_SSLServerRandom srnd;     The 32-byte Server Random field.
    M_SSLCipherSuite algs;      The SSL cipher suite to use
};
```

The following flag is defined:

```
Mech_SSLRecordLayer_IV_flags_IsClient
```

Set this flag if the module is acting as the SSL client.
The module supports the following cipher suites:

- **SSLCipherSuite_SSL3_NULL_MD5**
- **SSLCipherSuite_SSL3_NULL_SHA**
- **SSLCipherSuite_SSL3_RC4_40_MD5**
- **SSLCipherSuite_SSL3_RC4_128_MD5**
- **SSLCipherSuite_SSL3_RC4_128_SHA**
- **SSLCipherSuite_SSL3_DES_40_SHA**
- **SSLCipherSuite_SSL3_DES_SHA**
- **SSLCipherSuite_SSL3_DES3_SHA**
- **SSLCipherSuite_TLS_NULL_MD5**
- **SSLCipherSuite_TLS_NULL_SHA**
- **SSLCipherSuite_TLS_RC4_40_MD5**
- **SSLCipherSuite_TLS_RC4_128_MD5**
- **SSLCipherSuite_TLS_RC4_128_SHA**
- **SSLCipherSuite_TLS_DES_40_SHA**
- **SSLCipherSuite_TLS_DES_SHA**
- **SSLCipherSuite_TLS_DES3_SHA**
- **SSLCipherSuite_TLS_AES_128_SHA**
- **SSLCipherSuite_TLS_AES_256_SHA**

2. Used to sign or verify a SSL message. The mechanism uses the following ciphertext:

```
struct M_Mech_SSL3FinishedMsg_Cipher {
    M_Hash16 md5hash;
    M_Hash20 sha1hash;
};
```

3. Used to sign or verify a TLS message. The mechanism uses the following ciphertext:

```
struct M_Mech_TLSFinishedMsg_Cipher {
    M_Hash12 msg;
};
```

Twofish

Twofish uses the key data **M_KeyType_Random_Data**. The maximum permitted key data length is 32 bytes. Recommended key lengths are 16, 24 and 32 bytes.

Key data

```
typedef struct {
    M_ByteBlock k    data
} M_KeyType_Random_Data;
```

Key generation parameters

```
typedef struct {
    M_Word lenbytes;                          length in bytes
} M_KeyType_Random_GenParams;
```

Mechanisms

Mech_TwofishmECBpNONE	ECB with no padding
Mech_TwofishmCBCpNONE	CBC with no padding
Mech_TwofishmCBCi128PKCS5	CBC with PKCS #5 padding
Mech_TwofishmCBCMACi128PKCS5	*see note*
Mech_TwofishmECBpPKCS5	ECB with PKCS #5 padding
Mech_TwofishmCBCMACi0PKCS5	CBCMAC

Note: The mechanism **Mech_TwofishmCBCMACi128PKCS5** is deprecated and may be withdrawn in future versions.

Diffie-Hellman and ElGamal

Diffie-Hellman key exchange shares a common key type with ElGamal encryption and decryption.

Private key

```
typedef struct {
    M_DiscreteLogGroup dlg;
    M_Bignum x;
} M_KeyType_DHPrivate_Data
```

M_DiscreteLogGroup is a discrete log group that may be shared between users.

Public key

```
typedef struct {
    M_DiscreteLogGroup dlg;
    M_Bignum gx;
} M_KeyType_DHPublic_Data
```

M_DiscreteLogGroup is a discrete log group that may be shared between users.

Key generation parameters

```
typedef struct {
    M_Word flags;
    M_Word plength;
    M_Word xlength;
    M_DiscreteLogGroup *dlg;
} M_KeyType_DHPrivate_GenParams;
```

- The following **flags** are defined:
  - **KeyType_DHPrivate_GenParams_flags_dlg_present** (If this is set, the specified **DiscreteLogGroup** will be used.)
  - **KeyType_DHPrivate_GenParams_flags_SafePrimes** (If this is set, the module will generate the key, so that the key validation code can verify that the key has known good sub-group.)
  - **KeyType_DHPrivate_GenParams_flags__allflags**
- **plength** is key size in bits up to a maximum of 4096.
  Note: The present implementation uses the DSA/FIPS algorithm for generating **G** and **P** parameters, such that **P** must be a multiple of 64 bits in length and at least 512 bits long.
- **xlength** is the length in bits of private key X . DSA specifies 160 bits.
  There is no upper limit on the length of **P**. **(P - 1)** will have one prime factor of at least 160 bits, which is required in order to make Pohlig-Hellman discrete logs unworkable. The length of the private exponent **X** can be specified separately.
- **M_DiscreteLogGroup** is a discrete log group that may be shared between users.

```
typedef struct {
    M_Bignum p                  prime
    M_Bignum g                  generator mod P
} M_DiscreteLogGroup;
```

Note: DSA considers an exponent of 160 bits to be sufficient for security. An attempt to make the length of **X** greater than the length of **P** will have no effect.

Mechanisms

Mech_DHKeyExchange
Mech_ElGamal
Mech_DLIESe3DEShSHA1
Mech_DLIESeAEShSHA1

Diffie-Hellman

There is only one cryptographic operation, **Decrypt**, which is supported with the mechanism **DHKeyExchange** and the key type **DHPrivate**. A Diffie-Hellman key exchange goes as follows:

1. Alice generates a DH key pair and exports her public key.
2. Bob generates a DH key pair by using Alice's **G** and **P** values and by setting the **dlg_present** bit in the flags to **GenerateKeyPair**. He then exports his public key.

3. Alice takes Bob's public key and passes it as a ciphertext to **Decrypt** using her private key. This returns, in bignum format:
   $$\left(G^{X_A}\right)^{X_B} = G\left(X_A X_B\right) \mathrm{mod} P$$
4. Bob takes Alice's public key and passes it to Decrypt using his private key. This returns, in bignum format:
   $$\left(G^{X_B}\right)^{X_A} = G\left(X_B X_A\right) = G\left(X_A X_B\right) \mathrm{mod} P$$
   This result is the same as that which Alice derived.
5. The session key can then be derived from this multi-precision number.

ElGamal

At present, ElGamal encryption only takes nShield bignums as the plain text input and the output format.

DLIES

The **DLIESe3DEShSHA1** and **DLIESeAEShSHA1** mechanisms implement the DLIES encryption and decryption primitive as described in IEEE P1363A (Draft 11, December 16 2002), with the following options:

- DLSVDP-DH as the secret value derivation primitive
- KDF2 key derivation function, using SHA-1 as the underlying hash function
- Triple-DES-CBC-IV0 with 24-byte keys (**Mech_DLIESe3DEShSHA1**) or AES256-CBC-IV0 with 16-byte keys (**Mech_DLIESeAEShSHA1**) as the symmetric encryption scheme
- MAC1 based on SHA-1 as the message authentication scheme, with 160-bit output length and 160-bit key length

Note: The Asymmetric Encryption Scheme (DHAES) mode is not used.

Cipher text

Diffie-Hellman

```
typedef struct {
    M_Bignum gx;
} M_Mech_DHKeyExchange_Cipher;
```

ElGamal

```
typedef struct {
    M_Bignum a    g^k mod p
    M_Bignum b    M * (gx^k) mod p
} M_Mech_ElGamal_Cipher;
```

where **k** is a random integer **1 < k < (p–1)**

Note: ElGamal signature creation and verification are not currently implemented.

DSA

DSA enables users to share Discrete Log parameters, with each user having their own public and private key. DSA has 'communities', which are sets of keys that share a common **DSADiscreteLogGroup** but that have different (x, y) pairs. These are represented by the key type **DSAComm**, which consists of a **DSADiscreteLogGroup** set of values together with the initialization values (**seed**, **h**, and **counter**) from which the **DSADiscreteLogGroup** values were derived (as specified by the FIPS DSA specification).

A **DSAComm** key can be generated once, and then the **DSADiscreteLogGroup** from this **DSAComm** generation can be used in subsequent **DSAPrivate** generations.

DSAComm key generation also allows seed values to be checked as follows:

1. When generating a **DSAComm** key, set the **iv_present** flag bit, and pass in the **seed**, **counter**, and **h** values.
2. **GenerateKey** will follow the FIPS algorithm to generate a **p**, **q**, and **g** set, together with the associated **h** and **counter** values.
3. You can now export the resulting **DSAComm** key and check that **p**, **q**, **g**, **h**, and **counter** are what you were expecting.
4. **GenerateKey** will return **Status_InvalidData** if the given seed cannot be used to produce a valid **p**, **q**, or **g** value.

Note: The implementation of DSA that is used in Thales modules has been validated by NIST as conforming to FIPS 186, certificate number 11.

DSA keys

DSA common key

```
typedef struct {
    M_DSAInitValues iv;
    M_DSADiscreteLogGroup dlg;
} M_KeyType_DSAComm_Data;
```

DSA private key

```
typedef struct {
    M_DSADiscreteLogGroup dlg;
    M_Bignum x;
} M_KeyType_DSAPublic_Data;
```

DSA public key

```
typedef struct {
    M_DSADiscreteLogGroup dlg;
    M_Bignum y;
} M_KeyType_DSAPrivate_Data;
```

M_DSAInitValues:

```
typedef struct {
    M_Hash seed    seed
    M_Word counter    counter
    M_Word h    h
} M_DSAInitValues;
```

These are the initialization values, which can be used to check that the discrete logarithm parameters have been generated correctly.

M_DSADiscreteLogGroup:

```
typedef struct {
    M_Bignum p
    M_Bignum q
    M_Bignum g
} M_DSADiscreteLogGroup;
```

where

- **p** is a 512-bit to 1024-bit prime number;
- **q** is a 160-bit prime factor of **p–1**;
- **g** is $\mathbf{h}^{((p-1)/q}$, where **h < p–1** and $\mathbf{h}^{((p-1)/q)}$ **mod p > 1**.
- This is the discrete logarithm group. These values may be shared between users.
- A 160-bit number **< q**.

$\mathbf{g}^{x}$ **mod p** (a **p**-bit number).

DSA common generation parameters

```
typedef struct {
    M_Word flags;
    M_Word lenbits;
    M_DSAInitValues *iv;
} M_KeyType_DSAComm_GenParams;
```

The following **flags** are defined:

- **KeyType_DSAComm_GenParams_flags_iv_present**
- **KeyType_DSAComm_GenParams_flags__allflags**

lenbits is the length in bits

M_DSAInitValues:

```
typedef struct {
    M_Hash seed    seed
    M_Word counter    counter
    M_Word h    h
} M_DSAInitValues;
```

These are the initialization values, which can be used to check that the discrete logarithm parameters have been generated correctly.

DSA private key generation parameters

```
typedef struct {
    M_Word flags;
    M_Word lenbits;
    M_DSADiscreteLogGroup *dlg;
} M_KeyType_DSAPrivate_GenParams;
```

The following **flags** are defined:

- **KeyType_DSAPrivate_GenParams_flags_dlg_present** (If this flag is set, **GenerateKey** will use the specified **DSADiscreteLogGroup**.)
- **KeyType_DSAPrivate_GenParams_flags_Strict** (If this flag is set, the generated key is subjected to extra consistency tests at the expense of efficiency. There is normally no need to set this flag, unless you are supplying **p**, **q**, and **g** values and need to check them, or unless you require strict compliance with the FIPS 140-2 level 3 standard. Setting the **Strict** flag limits the maximum key size to 1024 bits. Otherwise, there is no maximum limit on key size.)
- **KeyType_DSAPrivate_GenParams_flags__allflags**

M_DSADiscreteLogGroup is the discrete logarithm group. These values may be shared between users.

```
typedef struct {
    M_Bignum p;
    M_Bignum q;
    M_Bignum g;
} M_DSADiscreteLogGroup;
```

where:

- **p** is a 512-bit to 1024-bit prime number;
- **q** is a 160-bit prime factor of **p–1**;
- **g** is $h^{((p-1)/q}$, where **h < p–1** and $h^{((p-1)/q)}$ **mod p > 1**.

Cipher text

```
typedef struct {
    M_Bignum r;
    M_Bignum s;
} M_Mech_DSA_Cipher;
```

r is g^k **mod p mod q**

s is k^{-1} **(H(m)+xr)) mod q**

Plain text

Because DSA is defined to sign a SHA-1 hash directly, it has no separate raw plain text format. Instead, the format **Hash** is used to indicate that the plain text which has been provided is the SHA-1 hash.

Mech	Unhashed plain text type	Hash used for bytes plaintext
Mech_DSA	Hash	
Mech_DSAhSHA224	Hash28	SHA-224
Mech_DSAhSHA256	Hash32	SHA-256
Mech_DSAhSHA384	Hash48	SHA-384
Mech_DSAhSHA512	Hash64	SHA-512
Mech_DSAhRIPEMD160	Hash	RIPEMD-160

If the plain text format is **Bytes**, then the mechanism will hash the plain text itself before signing.

Mechanisms

Mech_DSA	10
Mech_DSAhSHA224	
Mech_DSAhSHA256	
Mech_DSAhSHA384	
Mech_DSAhSHA512	
Mech_DSAhRIPEMD160	

Elliptic Curve ECDH and ECDSA

The Thales module supports key exchange, ECDH, and signature mechanisms.

The module supports a wide range of curves, including all the the curves listed in FIPS 186-2 and some curves from X9.62. It also allows a user to specify a custom curve.

Note: The implementation of ECDSA over curves recommended for US Government use has been validated by NIST, as conforming to FIPS 186-2, certificate 2.

When you create a key, you must create it as either an ECDSA key or an ECDH key. However, both keys use the same underlying structure. This ensures keys are used for the correct purpose and prevents inadvertent use of a signing key for key exchange, or an exchange key for signing message.

Elliptic Curve keys

Private keys

```
struct M_KeyType_ECPrivate_Data {
    M_EllipticCurve curve;
    M_Bignum d;
};
```

- **curve** is the curve used.
- **d** is an integer up to the order of the group.

Public keys

```
struct M_KeyType_ECPublic_Data {
    M_EllipticCurve curve;
    M_ECPoint Q;
};
```

- **curve** is the curve used.
- **Q** is a point on the curve.

Key generation parameters

```
struct M_KeyType_ECPrivate_GenParams {
    M_EllipticCurve curve;
};
```

- **curve** is the curve used.

Cipher text - ECDH

```
struct M_Mech_ECDHKeyExchange_Cipher {
    M_ECPoint gd;
};
```

- **gd** is the public point provided in the public key supplied in the key exchange.

Cipher text - ECDSA

```
struct M_Mech_ECDSA_Cipher {
    M_Bignum r;
    M_Bignum s;
};
```

r is x_1 **mod n**

s is **s = k** $^{-1}$ **(e + dr) mod n**.

Plain text - ECDH

Mech ECDHKeyExchange can return plaintext as:

- **M_ECPoint** the canonical form;
- **M_Bignum** the x coordinate of the point;
- **M_Byteblock** in uncompressed octet string representation.

Plain text - ECDSA

ECDSA can accept plain text as either **hash** or **bytes**.

Mech	Unhashed plain text type	Hash used for bytes plaintext
Mech_ECDSA	Hash	
Mech_ECDSAhSHA224	Hash28	SHA-224
Mech_ECDSAhSHA256	Hash32	SHA-256
Mech_ECDSAhSHA384	Hash48	SHA-384
Mech_ECDSAhSHA512	Hash64	SHA-512
Mech_ECDSAhRIPEMD160	Hash	RIPEMD-160

Mechanisms

```
Mech_ECDSA
Mech_ECDH
Mech_ECDSAhSHA224
Mech_ECDSAhSHA256
Mech_ECDSAhSHA384
Mech_ECDSAhSHA512
Mech_ECDSAhRIPEMD160
```

Note: Neither Mech_ECDSA nor Mech_ECDH handle normal representations.

KCDSA

KCDSA is a Korean algorithm that has been standardized by the Korean government as KCS221. The compliance of nShield's implementation compliance to this standard has not been independently verified.

Note: If you wish to use the KCDSA algorithm, you must order and enable it as part of the Thales **KISAAlgorithms** feature, as described in the User Guide. If you are outside Korea, contact Thales for information about obtaining the appropriate export licence.

KCDS'A enables users to share Discrete Log parameters, with each user having their own public and private key. KCDSA has 'communities', which are sets of keys that share a common

KCDSADiscreteLogGroup but that have different (x, y) pairs. These are represented by the key type **KCDSAComm**, which consists of a **KCDSADiscreteLogGroup** set of values together with the initialization values (**seed** and **counter**) from which the **KCDSADiscreteLogGroup** values were derived (as specified by the KCDSA specification).

A **KCDSAComm** key can be generated once, and then the **KCDSADiscreteLogGroup** from this **KCDSAComm** generation can be used in subsequent **KCDSAPrivate** generations.

KCDSAComm key generation also allows seed values to be checked as follows:

1. When generating a **KCDSAComm** key, set the **iv_present** flag bit, and pass in the **seed** and **counter** values.
2. **GenerateKey** will follow the KCDSA algorithm to generate a **p**, **q**, and **g** set.
3. You can now export the resulting **KCDSAComm** key and check that **p**, **q**, and **g** are what you were expecting.
4. **GenerateKey** will return **Status_InvalidData** if the given **seed** and **counter** cannot be used to produce a valid **p**, **q**, or **g** value.

KCDSA keys

KCDSA common key

```
typedef struct {
    M_KCDSAInitValues iv;
    M_KCDSADiscreteLogGroup dlg;
} M_KeyType_KCDSAComm_Data;
```

M_KCDSAInitValues

```
typedef struct {
    M_ByteBlock seed;                          seed
    M_Word counter    counter
} M_KCDSAInitValues;
```

These are the initialization values, which can be used to check that the discrete logarithm parameters have been generated correctly.

M_KCDSADiscreteLogGroup is the discrete logarithm group. These values may be shared between users.

```
typedef struct {
    M_Bignum p;
    M_Bignum q;
    M_Bignum g;
} M_KCDSADiscreteLogGroup;
```

where:

- **p** is a 1024-bit to 2048-bit prime number which is a multiple of 256 bits long;
- **q** is always 160 bits long;
- **g** is $h^{((p-1)/q)}$, where **h < p–1** and $h^{((p-1)/q)}$ **mod p > 1**.

KCDSA private key

```
typedef struct {
    M_KCDSADiscreteLogGroup dlg;
    M_Bignum y;
    M_Bignum x;
} M_KeyType_KCDSAPublic_Data;
```

M_KCDSADiscreteLogGroup is the discrete logarithm group. These values may be shared between users.

```
typedef struct {
    M_Bignum p;
    M_Bignum q;
    M_Bignum g;
} M_KCDSADiscreteLogGroup;
```

where:

- **p** is a 1024-bit to 2048-bit prime number which is a multiple of 256 bits long;
- **q** is always 160 bits long;
- **g** is $h^{((p-1)/q)}$, where **h < p–1** and $h^{((p-1)/q)}$ **mod p > 1**.
- **x** is an arbitrary number where **0 < x < q**.
- **y** is $g^{(1/x \bmod q)}$ **mod p** (a number less than **p**).

KCDSA public key

```
typedef struct {
    M_KCDSADiscreteLogGroup dlg;
    M_Bignum y;
} M_KeyType_KCDSAPrivate_Data;
```

M_KCDSADiscreteLogGroup is the discrete logarithm group. These values may be shared between users.

```
typedef struct {
    M_Bignum p;
    M_Bignum q;
    M_Bignum g;
} M_KCDSADiscreteLogGroup;
```

where:

- **p** is a 1024-bit to 2048-bit prime number which is a multiple of 256 bits long;
- **q** is always 160 bits long; ;

- **g** is $\mathbf{h}^{((p-1)/q}$, where **h < p–1** and $\mathbf{h}^{((p-1)/q)}$ **mod p > 1**.
- **y** is $\mathbf{g}^{(1/x \bmod q)}$ **mod p** (a number less than **p**).

Key generation parameters

KCDSA common generation parameters

```
typedef struct {
    M_Word flags;
    M_Word plen;
    M_Word qlen;
    M_KCDSAInitValues *iv;
} M_KeyType_KCDSAComm_GenParams;
```

- The following **flags** are defined:
  - **KeyType_KCDSAComm_GenParams_flags_iv_present**
  - **KeyType_KCDSAComm_GenParams_flags__allflags**
- **plen** is the length of **p** in bits, a multiple of 256 where 1024 ≤ **plen** ≤ 2048.
- **qlen** is the length of **q** in bits, a multiple of 32 where 160 ≤ **qlen** ≤256. This value must currently be 160.

M_KCDSAInitValues

```
typedef struct {
    M_ByteBlock seed;                          seed
    M_Word counter      counter
} M_KCDSAInitValues;
```

These are the initialization values, which can be used to check that the discrete logarithm parameters have been generated correctly.

KCDSA private key generation parameters

```
typedef struct {
    M_Word flags;
    M_Word plen;
    M_Word qlen;
    M_KCDSADiscreteLogGroup *dlg;
} M_KeyType_KCDSAPrivate_GenParams;
```

- The following **flags** are defined:
  - **KeyType_KCDSAPrivate_GenParams_flags_dlg_present** (If this flag is set, **GenerateKey** will use the specified**KCDSADiscreteLogGroup**.)
  - **KeyType_KCDSAPrivate_GenParams_flags__allflags**
- **plen** is the length of **p** in bits.
- **qlen** is the length of **q** in bits.
- **M_KCDSADiscreteLogGroup** is the discrete logarithm group. These values may be shared between users.

```
typedef struct {
    M_Bignum p;
    M_Bignum q;
    M_Bignum g;
} M_KCDSADiscreteLogGroup;
```

where:

- **p** is a 1024-bit to 2048-bit prime number which is a multiple of 256 bits long;
- **q** is always 160 bits long; ;
- **g** is $\mathbf{h}^{((p-1)/q}$, where **h < p–1** and $\mathbf{h}^{((p-1)/q)}$ **mod p > 1**.

Cipher text

```
typedef struct {
    M_ByteBlock r;
    M_Bignum s;
} M_Mech_KCDSA_Cipher;
```

- **r** is $\mathbf{h(g}^{k}$ **mod p)**.
- **s** is **x(k - (r** ⊕ **h (z||m))) mod q**

Note: The symbol ⊕ represents a bit-wise XOR operation. The symbol || represents concatenation of **Byteblock**s

Plain text

Note: See *Key Types* on page 66 for a list of plain text formats.

KCDSA hashes the message **m** as **h(z||m)**, where **z** is derived from the public key. For short messages, **m** may be supplied directly as**PlainTextType_Bytes**. For longer messages, the hash **h(z||m)** may be computed externally and supplied as **PlainTextType_Hash**.

Mechanisms

Mech_KCDSAHAS160	110
Mech_KCDSASHA1	111
Mech_KCDSARIPEMD160	112
Mech_KCDSASHA224	
Mech_KCDSASHA256	

RSA

Public key

```
typedef struct {
    M_Bignum e    Exponent
    M_Bignum n    Modulus
} M_KeyType_RSAPublic_Data;
```

RSA public keys contain exponent and modulus only. The exponent is usually simple, reducing the complexity of the modular exponentiation. RSA keys generated by an Thales module have the public exponent **0x10001** by default.

Private key

```
typedef struct {
    M_Bignum p
    M_Bignum q
    M_Bignum dmp1
    M_Bignum dmq1
    M_Bignum iqmp
    M_Bignum e
} M_KeyType_RSAPrivate_Data;
```

- **dmp1** is $D \bmod P-1$
- **dmq1** is $D \bmod Q-1$
- **iqmp** is $Q^{-1} \bmod P$

RSA private keys, for which the exponent is usually large, contain additional information that enables the modular exponentiation to be optimized by using the Chinese Remainder Theorem.

Generation parameters

```
Generation parameters
typedef struct {
    M_Word flags;
    M_Word lenbits;
    M_Bignum *given_e;
    M_Word *nchecks;
} M_KeyType_RSAPrivate_GenParams;
```

- The following **flags** are defined:
  - **KeyType_RSAPrivate_GenParams_flags_given_e_present**
    If this flag is set, the user can specify which public exponent is to be used. If this flag is not set, the public exponent will be set to **0x10001** or, for very short keys, **0x11**.
  - **KeyType_RSAPrivate_GenParams_flags_nchecks_present**

If this flag is set, the user can specify the number of Rabin-Miller checks that are to be done on the primes. The default for this number varies with key size to give a 2^{-100} probability of error.

- **KeyType_RSAPrivate_GenParams_flags_UseStrongPrimes**
  Setting this flag requests key generation in accordance with ANSI X9.31 requirements. Specifically:
  - the key length must be at least 1024 bits, and a multiple of 256 bits
  - primes **p** and **q** are 'strong' - that is **p+1**, **p-1**, **q+1** and **q-1** each have at least one prime factor $>2^{100}$
  - primes **p** and **q** each pass 8 iterations of the Rabin-Miller test followed by the Lucas test
  - **p** and **q** differ somewhere in their most significant 100 bits.
- **KeyType_RSAPrivate_GenParams_flags__allflags**

- ***given_e** specifies the public exponent to be used. This must be an odd value greater than 1 and less than half the requested key length.
- ***nchecks** specifies the number of Rabin-Miller checks to be performed.

Mechanisms

For RSAPublic and RSAPrivate keys, the following mechanisms are provided:

Mech_RSApPKCS1=	*see note 1*
Mech_RSAhSHA1pPKCS1=	*see note 2*
Mech_RSAhRIPEMD160pPKCS1=	*see note 2*
Mech_RSApPKCS1OAEP=	*see note 3*
Mech_RSApPKCS1OAEPhSHA224	
Mech_RSApPKCS1OAEPhSHA256	
Mech_RSApPKCS1OAEPhSHA384	
Mech_RSApPKCS1OAEPhSHA512	
Mech_RSAhSHA1pPSS	
Mech_RSAhRIPEMD160pPSS	
Mech_RSAhSHA224pPSS	
Mech_RSAhSHA256pPSS	
Mech_RSAhSHA384pPSS	
Mech_RSAhSHA512pPSS	

1. This mechanism has the following behavior:
   - **Encrypt**
     - accepts plain text of the type **Bignum** or **Bytes**
     - for plaintext type **Bytes** pads and encrypts the message according to PKCS #1
     - for plaintext type **Bignum** encrypts the input directly
     - returns a cipher text of the type **M_Mech_RSApPKCS1_Cipher**
   - **Decrypt**
     - accepts cipher text of the appropriate type **M_Mech_RSApPKCS1_Cipher**
     - decrypts the message and strips the padding
     - returns plain text in format **Bytes**

- **Sign**
  - accepts plain text of the type **Bignum** or **Bytes**
  - for plaintext type **Bytes** pads and encrypts the message according to PKCS #1
  - for plaintext type **Bignum** signs the input directly
  - returns a cipher text of the type **M_Mech_RSApPKCS1_Cipher**
- **Verify**
  - accepts plain text of the type **Bignum** or **Bytes**
  - accepts cipher text of the type **M_Mech_RSApPKCS1_Cipher**, which is decrypted and compared to the appropriate hash of the plain text.

Note: This mechanism does not hash the message before signing it.

You should use the **Hash** command in order to produce a hash to pass to the **Sign** or **Verify** command. For PKCS #1 compatible signatures, the **ObjectID** that identifies the hash algorithm should be placed before the hash value itself to form a plain text of the type **Bytes**. Alternatively, you can use **RSAhMD5pPKCS1** and similar mechanisms that hash the plaintext first.

Note: Although the **RSApPKCS1** mechanism will accept a hash plain text for signature or verification, this operation will not result in a valid PKCS #1 signature.

2. These mechanisms will **Sign** and **Verify** only. They have the following behavior:
   - **Sign** accepts plain text of the type **Bignum**, **Bytes** or appropriate hash.
     - for **Bignum** no padding is performed
     - for **Bytes**, **Sign** hashes this plain text with the selected hash function, adds the correct **ObjectID**, pads the result using PKCS #1 padding.
     - the **hash** must be the correct size for the hash mechanism specified: adds the correct **ObjectID**, pads the hash using PKCS #1 padding, the resulting padded string is then encrypted.
   - **Verify** accepts plain text of type **Bytes** and cipher text of the type **M_Mech_RSApPKCS1_Cipher**, which is decrypted, has its padding stripped, and is then compared to the plain text.

   Note: You must ensure that the message fits into a single command block. If the message is too large to fit into a single block, the server will use channel commands to pass the command, which will fail because channel commands do not support RSA. If you are not certain that the data will fit into a single command block, use separate **Hash** and **Sign** commands.
3. This mechanism performs encryption and decryption with OAEP padding. It implements the **RSAES-OAEP-ENCRYPT** and **RSAES-OAEP-DECRYPT** primitives as given in PKCS #1 v2.0, using SHA-1 as the Hash option and MGF1-with-SHA1 as the MGF function.

   Note: This is similar in concept to, but in practice totally incompatible with, the OAEP as used in SET.

   The input to the **Encrypt** function must be a **Bytes** type plain text with a length from **0** to **(modulus length in bytes minus 42)** bytes inclusive.

   Thus, a 512-bit modulus (of 64 bytes) will be able to encode up to 22 bytes of information.

   Note: This quantity is insufficient to make a direct blob. You must use at least a 528-bit modulus to make a direct blob.

   Unlike the SET OAEP mechanism, PKCS #1 OAEP preserves the length of the plain text block. RSAES-OAEP defines an encoding parameters string, **p**. This string is a byte block that is used as extra padding. In order to pass encoding parameters to the Encrypt command, set the **given_iv_present** flag, and enter the encoding parameters as the IV. In order to pass encoding parameters to the **Decrypt** command, set the IV in the **iv** member of the **cipher** parameter. The IV is in the form of a byte block **p**, the length of which may be 0.

Cipher text - PKCS #11 padding

```
typedef struct {
    M_Bignum m;
} M_Mech_RSApPKCS1_Cipher;
```

Cipher text - OAEP padding

```
typedef struct {
    M_Bignum m;
} M_Mech_RSApSETOAEP_Cipher;
```

DeriveKey

DKTemplate

A **DKTemplate** is a template key whose key data contains a marshalled ACL and application data. **DKTemplate** keys cannot be created with **GenerateKey** because this would produce a random ACL. You must **Import** the key.

```
typedef struct {
    M_ByteBlock appdata;
    M_ByteBlock nested_acl;
} M_KeyType_DKTemplate_Data;
```

- **appdata** specifies application data for the new key.
- **nested_acl** is the marshalled ACL for the new key. Use the function **NFastApp_MarshalACL()** in order to produce an ACL in the correct format.

Wrapped

A wrapped key contains encrypted key data as a byte block. A wrapped key has the same structure as a random key, but is a separate type.

You can generate a wrapped key by generating two random numbers and XORing them together to create a key. If you randomly generate both halves of a DES or a triple DES key, you must use one of the mechanisms that sets the parity of the resultant key: **DeriveMech_DESjoinXORsetParity** or **DeriveMech_DES3joinXORsetParity**.

Alternatively, you can marshal keys, as described in *Mechanisms* on page 64.

Generation parameters

```
typedef struct {
    M_Word flags;
    M_Word length;
} M_KeyType_Wrapped_GenParams;
```

- No **flags** are defined.
- **length** specifies the length in bytes:
  - 8 bytes for a wrapped DES key
  - 24 bytes for a wrapped Triple DES key

Derive Key Mechanisms

DeriveMech_DESsplitXOR	*see note 1*
DeriveMech_DESjoinXOR	*see note 2*
DeriveMech_DES2splitXOR	*see note 1*
DeriveMech_DES2joinXOR	*see note 2*
DeriveMech_DES3splitXOR	*see note 1*
DeriveMech_DES3joinXOR	*see note 2*
DeriveMech_DESjoinXORsetParity	*see note 2*
DeriveMech_DES2joinXORsetParity	*see note 2*
DeriveMech_DES3joinXORsetParity	*see note 2*
DeriveMech_RandsplitXOR	*see note 1*
DeriveMech_RandjoinXOR	*see note 2*
DeriveMech_CASTsplitXOR	*see note 1*
DeriveMech_CASTjoinXOR	*see note 2*
DeriveMech_EncryptMarshalled	*see note 3*
DeriveMech_DecryptMarshalled	*see note 3*
DeriveMech_RSAComponents	*see note 4*
DeriveMech_PKCS8Encrypt	*see note 5*
DeriveMech_PKCS8Decrypt	*see note 5*
DeriveMech_RawEncrypt	*see note 6*
DeriveMech_RawDecrypt	*see note 6*
DeriveMech_SSL3withRSA	*see note 7*
DeriveMech_SSL3withDH	*see note 8*
DeriveMech_TLSwithRSA	*see note 7*
DeriveMech_TLSwithDH	*see note 8*

DeriveMech_AESsplitXOR	*see note 1*
DeriveMech_AESjoinXOR	*see note 2*
DeriveMech_SignedKDPKeyWrapDES3	*see note 9*
DeriveMech_Any	
DeriveMech_PublicFromPrivate	*see note 10*
DeriveMech_KDPKeyWrapDES3	
DeriveMech_ECCMQV	
DeriveMech_ConcatenateBytes	
DeriveMech_ConcatenationKDF	
DeriveMech_ NISTKDFmCTRpRijndaelCMACr32	
DeriveMech_RawEncryptZeroPad	
DeriveMech_RawDecryptZeroPad	
DeriveMech_AESKeyWrap	
DeriveMech_AESKeyUnwrap	

1. These mechanisms take a base key of the specified type and a wrapping key of type **Random** to produce an output key of type **Wrapped**.
2. These mechanisms take a base key of type **Wrapped** and a wrapping key of type **Random** to produce an output key of the specified type.
3. The **EncryptMarshalled** and **DecryptMarshalled** mechanisms are provided to allow export of keys from a module in Strict FIPS 140-2 mode and import into a module in the same mode.

The **EncryptMarshalled** mechanism takes a template key, a base key of any marshallable type, and a wrapping key of any type capable of encrypting, and does the following:

a. Marshals an **M_PlainText** structure that represents the base key to produce a byte string.
b. Turns the byte string into **Bytes** plaintext, and encrypts it with the wrapping key to produce ciphertext.
c. Marshals the ciphertext into a further byte string.
d. Creates a key of the type **Wrapped** that has the ACL given in the template key and contains the byte string from step c as data. That is, the wrapped data is a marshalled ciphertext which is an encryption of the marshalled key data.

   Note: All marshalling is done in module-internal format (little-endian arrays of little-endian words).

   Template and **Wrapped** keys can be imported into the module even in strict FIPS mode. The import must be authorized by a certificate signed by the Thales Security Officer's key K_{NSO}.

   The **DecryptMarshalled** mechanism performs the complementary operation: it unmarshals and decrypts a ciphertext represented as a **Wrapped** key, then unmarshals the resulting plaintext to recover the **M_PlainText** structure for the output key.
   An example of importing keys using the **DecryptMarshalled** mechanism:
e. Generate an RSA key pair **Kpub**, **Kpriv**. **Kpub** must have export-as-plain permissions; **Kpriv** must have a **DeriveKey** action group that specifies a role of **WrapKey** and a mechanism of **DecryptMarshalled**. Export **Kpub**.
f. Marshall the key **Ki** to be imported. Pad the result according to PKCS #1 and encrypt it with **Kpub** (for example, using the **ModExp** command).
g. Marshal the ciphertext: write **Mech_RSApPKCS1** as an **M_Word (02 00 00 00)**, the length of the bignum, then the bytes in little-endian order. Import the resulting byteblock as a key **Kw** of type **Wrapped**.
h. Create a template key **Kt** that contains the desired ACL for the key to be imported, and import it.
i. Use **DeriveKey** with **Kt** as the template, the **Kw** as the base key, and **Kpriv** as the wrapper key.
   The resulting key is **Ki** imported with the correct ACL.

4. The **RSAComponents** mechanism is provided to allow export of RSAPrivate key types.

This mechanism is not intended for secure transport of key data between Thales modules. It has a number of security weaknesses, not least poor protection of key integrity. It is provided only as an aid to interoperating with other systems when more secure methods are not available.

The **RSAComponents** mechanism has the following structure:

```
typedef M_IV *M_vec_IV;
struct M_DeriveMech_RSAComponents_DKParams {
    int n_ivs;
M_vec_IV ivs;
};
```

The **RSAComponents** mechanism takes a Base key of type **RSAPrivate** and a Wrap key of any symmetric type capable of encrypting byte streams. The **ivs** table must contain exactly five **IV** values. These values can be different, and can specify different mechanisms if required. They must all be valid encryption mechanisms for the Wrap key type.
The mechanism processes the keys as follows:

a. The block length is determined by taking the greater of the **p** or **q** components of the RSA key (normally **p**) and rounding its size up to the next 64-bit boundary. For a 1024-bit RSA key, the lengths of p and q are normally 512 bits (or 64 bytes) each.
b. The value of each of the following key components is written out as a big-endian byte stream. Leading zeroes are added to bring the size up to the block length.
   - **p**
   - **q**
   - **d mod (p-1)**
   - **d mod (q-1)**
   - **inv (q) mod p**
c. Each block is encrypted using the corresponding entry from the **IVs** table. That is, **ivs[0]** is used to encrypt **p**, **ivs[1]** is used for **q**, and so on.
d. The bytes of the output ciphertext are stored.
e. All five ciphertext blocks are concatenated in order and converted to a key of the type Wrapped.
f. No corresponding import/decrypt mechanism is currently supported.

5. The **PKCS8Encrypt** and **PKCS8Decrypt** mechanisms are provided to allow private key data for asymmetric algorithms to be imported and exported.

This mechanism is not intended for secure transport of key data between Thales modules. It has a number of security weaknesses, not least poor protection of key integrity. It is provided only as an aid to interoperating with other systems when more secure methods are not available.

The **PKCS8Encrypt** and **PKCS8Decrypt** mechanisms have the following structure:

```
struct M_DeriveMech_PKCS8Encrypt_DKParams {
    M_IV iv;
};
struct M_DeriveMech_PKCS8Decrypt_DKParams {
    M_IV iv;
};
```

The **PKCS8Encrypt** mechanism takes a Base key of type **RSAPrivate**, **DSAPrivate**, **ECDSAPrivate**, **ECDHPrivate** or **DHPrivate**, and a Wrap key of any symmetric type capable of encrypting byte streams. The private key data is BER-encoded according to PKCS #8. (This process is also described in the PKCS #11 specification under **Wrapping/unwrapping private keys**.) The resulting byte block is encrypted, using the given **iv**, which includes a mechanism. The data of the ciphertext is converted into a key of type **Wrapped**.

The **PKCS8Decrypt** mechanism performs the opposite process: it takes a **Wrapped** key type as the Base key and a symmetric key as the Wrapping key. The data is decrypted using the given **iv** and mechanism, and then BER-decodes to give a **RSAPrivate**, **DSAPrivate**, **ECDSAPrivate** or **DHPrivate** output key.

The following errors may indicate mechanism-specific problems:

- **TypeMismatch**: The ciphertext type for the given mechanism is not a simple byteblock, and so cannot be converted to or from a Wrapped key type.
- **NotYetImplemented**: During encoding, this error indicates that the Base key is not of a type for which BER-encoding is supported. During decoding, this error indicates that an element has been encountered which is not used for the supported key types (for example, a negative integer value). This may indicate the data has been corrupted.
- **UnknownParameter**: During decoding, this error indicates that a key type other than those supported, or an unknown 'version' integer, has been encountered.
- **Malformed**: The BER-decoding has been unsuccessful, probably due to corrupted data, for example, because the data is too short, or because an illegal byte value has been encountered).

6. The **RawEncrypt** and **RawDecrypt** mechanisms are provided to allow raw key data to be encrypted and decrypted using any key that accepts a cipher text as Bytes.

This mechanism is not intended for secure transport of key data between Thales modules. It has a number of security weaknesses, not least poor protection of key integrity. It is provided only as an aid to interoperating with other systems when more secure methods are not available.

These mechanisms have the following structure:

```
struct M_DeriveMech_RawEncrypt_DKParams {
    M_IV iv;
};
struct M_DeriveMech_RawDecrypt_DKParams {
    M_IV iv;
    M_KeyType dst_type;
};
```

The **RawEncrypt** mechanism processes the key as follows:

a. It extracts the key data of the Base key as a byte block and encrypts it using the Wrapping key, **IV** and the mechanism specified in the IV, which must be a valid mechanism for the given Wrapping Key. Mechanisms that do not perform padding cannot encrypt plain texts which are not multiples of the block length. For example, **DESmECBpNONE** can encrypt only base keys that are a multiple of 8 bytes in length.
b. The resulting ciphertext is converted directly into a Wrapped key. No mechanism, **IV**, or base key type information is saved with the Wrapped data. This data must be transported separately.

RawDecrypt performs the reverse process.The type of the key to be created, and the **IV** to be used when decrypting, are passed in the **dst_type** and **iv** fields, respectively.

The following errors have specific meanings:

- **TypeMismatch**: The chosen Base key type is not a DES or simple ByteBlock key type (for example, an RSAPrivate key), so it cannot be converted to or from a byte block plaintext for encryption. Alternatively, the specified encryption or decryption mechanism doesn't use a byte block for its ciphertext (for example, it uses ciphertexts containing Bignums) so the ciphertext cannot be converted to or from Wrapped key data.
- **InvalidData**: The data cannot be made into a key of the given type. For example, the decrypted data was too short or too long for the given destination key type, or the destination key type was a DES, DES2 or DES3 key and the decrypted data had parity errors. You can force the parity to be set correctly, by using **RawDecrypt** to produce a key of type Wrapped, and importing a Random key of the right length with all bytes zero. Then use the **DESjoinXORsetParity** mechanisms on these two keys to produce a DES key with correct parity bits.

7. No Wrapping key is required for the **DeriveMech_SSL3withRSA** or **DeriveMech_TLSwithRSA** mechanism. Instead, the **params** field in the **DeriveKey** command must contain the following details:

```
struct M_DeriveMech_SSL3withRSA_DKParams {
    M_Bignum ct;
    M_SSLClientRandom crnd;
    M_SSLServerRandom srnd;
};
```

Where:

- **ct** is the ciphertext from the SSL key exchange message, in the form of an **M_Bignum**
- **crnd** is the 32-byte Client Random field.
- **srnd** is the 32-byte Server Random field.

The resulting key is of type **SSLMasterSecret**, which can be used with the following mechanisms:

- **Mech_SSLRecordLayer** (encryption/decryption)
- **Mech_SSL3FinishedMsg** (sign/verify)
- **Mech_TLSFinishedMsg** (sign/verify)

8. No Wrapping key role is required for the **DeriveMech_SSL3withDH** or **DeriveMech_TLSwithDH** mechanism. Instead, the **params** field in the **DeriveKey** command must contain the following details:

```
struct M_DeriveMech_SSL3withDH_DKParams {
    M_Bignum y;
    M_SSLClientRandom crnd;
    M_SSLServerRandom srnd;
};
```

Where:

- **y** is the ciphertext from the SSL key exchange message, in the form of an **M_Bignum**
- **crnd** is the 32-byte Client Random field.
- **srnd** is the 32-byte Server Random field.

The resulting key is of type **SSLMasterSecret**, which can be used with the following mechanisms:

- **Mech_SSLRecordLayer** (encryption/decryption)
- **Mech_SSL3FinishedMsg** (sign/verify)
- **Mech_TLSFinishedMsg** (sign/verify)

9. Mechanism for the MicroHSM key delivery protocol. For Thales internal use only.
10. **DeriveMech_PublicFromPrivate** constructs the corresponding public key given one private key of any type.
    The following is a non-exhaustive list of common error returns specific to this key derivation mechanism:
    - **TypeMismatch**: given key is not a private key.
    - **InvalidParameter**: more than one key supplied.

Hash functions

Hash functions take an input of arbitrary length and return an output of fixed length.

The **Hash** function supports the RIPEMD-160, SHA-1, SHA-256, SHA-384, SHA-512, Tiger, MD2, and MD5 mechanisms.

All the hashes that the module uses internally employ the SHA-1 algorithm.

SHA-1

SHA-1 is a hash function that has been approved by NIST. SHA-1 returns a 20-byte result.

Note: The implementation of SHA-1, SHA-256, SHA-384 and SHA-512 in the Thales module has been validated by NIST as conforming to FIPS 18-2, certificate 255.

Mechanism

```
Mech_SHA1Hash
```

Reply

```
typedef struct {
    M_Hash20 h;
} M_Mech_SHA1Hash_Cipher;
```

Tiger

Tiger is a hash function designed by Ross Anderson and Eli Biham. It is designed to be efficient on 64-bit processors and to be no slower than MD5 on 32-bit processors.

Mechanism

```
Mech_TigerHash
```

Reply

```
typedef struct {
    M_Hash24 h;
} M_Mech_TigerHash_Cipher;
```

SHA-224

SHA-224 is a member of the SHA-2 hash function family that yields a 28-byte result.

Mechanism

```
Mech_SHA224
```

Reply

```
typedef struct {
    M_Hash28 h;
} M_Mech_SHA224Hash_Cipher;
```

SHA-256

SHA-256 is a member of the SHA-2 hash function family that yields a 32-byte result.

Mechanism

```
Mech_SHA256
```

Reply

```
typedef struct {
    M_Hash32 h;
} M_Mech_SHA256Hash_Cipher;
```

SHA-384

SHA-384 is a member of the SHA-2 hash function family that yields a 48-byte result.

Mechanism

```
Mech_SHA384Hash
```

Reply

```
typedef struct {
    M_Hash48 h;
} M_Mech_SHA384Hash_Cipher;
```

SHA-512

SHA-512 is a member of the SHA-2 hash function family that yields a 64-byte result.

Mechanism

```
Mech_SHA512Hash
```

Reply

```
typedef struct {
    M_Hash64 h;
} M_Mech_SHA512Hash_Cipher;
```

MD2

MD2 is a hash function that was designed by Ron Rivest. MD2 returns a 16-byte hash.

Mechanism

```
Mech_MD2Hash
```

Reply

```
typedef struct {
    M_Hash16 h;
} M_Mech_MD2Hash_Cipher;
```

MD5

MD5 is a hash function that was designed by Ron Rivest. MD5 returns a 16-byte hash.

Mechanism

```
Mech_MD5Hash
```

Reply

```
typedef struct {
    M_Hash16 h;
} M_Mech_MD5Hash_Cipher;
```

RIPEMD 160

RIPEMD 160 is a hash function that was developed as part of the European Union's RIPE project. RIPEMD 160 returns a 20-byte hash.

Mechanism

```
Mech_RIPEMD160Hash
```

Reply

```
typedef struct {
    M_Hash20 h;
} M_Mech_RIPEMD160Hash_Cipher;
```

HAS160

HAS160 is a hash function designed for use with the KCDSA algorithm. (See *KCDSA* on page 88.) HAS160 returns a 20-byte hash.

Note: If you wish to use the HAS160 hash function, you must order and enable it as part of the Thales **`KISAAlgorithms`** feature, as described in the *User Guide*.

Mechanism

```
Mech_HAS160Hash                                    109
```

Reply

```
typedef struct {
    M_Hash20 h;
} M_Mech_HAS160Hash_Cipher;
```

HMAC signatures

The **sign** and **verify** commands can create and verify MACs that have been created with the HMAC procedure and any supported hashing algorithm.

See RFC2104 for a description of HMAC.

Note: The nShield implementations of HMAC SHA-1, HMAC SHA-224, HMAC SHA-256, HMAC SHA-384 and HMAC SHA-512 have been validated by NIST as conforming to FIPS 198, certificate 3.

The following key types are defined:

- **KeyType_HMACMD2**
- **KeyType_HMACMD5**
- **KeyType_HMACSHA1**
- **KeyType_HMACRIPEMD160**
- **KeyType_HMACSHA224**
- **KeyType_HMACSHA256**
- **KeyType_HMACSHA384**
- **KeyType_HMACSHA512**
- **KeyType_HMACTiger**

All these key types contain random data that is stored in byte blocks of variable length.

They use the key type **Random** for their data and key generation parameters.

The following mechanisms are defined:

- **Mech_HMACMD2**
- **Mech_HMACMD5**
- **Mech_HMACSHA1**
- **Mech_HMACRIPEMD160**
- **Mech_HMACSHA224**
- **Mech_HMACSHA256**
- **Mech_HMACSHA384**
- **Mech_HMACSHA512**
- **Mech_HMACTiger**

ACLs

An ACL is a list of actions that are permitted for this object. An ACL consists of a list of *permission groups*.

Each permission group is a list of actions combined with an optional set of limits, either numerical limits or time limits, and optionally the hash of the key needed to authorize these actions.

By creating multiple permission groups with different **`use limits`** and **`certifiers`** you create an ACL:

```
typedef struct {
    int n_groups
    M_ PermissionGroup *groups;
} M_ACL;
```

- **`n_groups`** is the number of groups.
- **`*groups`** This is a list of permission groups. Each permission group consists of the following items:
  - optionally, the key hash of a key that must be used to certify all operations within this permission group. The given key must be used to produce a certificate that accompanies the request. This certificate can also be required to be 'fresh'. If no key hash is given, this is a public permission group and defines operations available without a certificate.
  - 0 or more **`use limits`** for this permission group. If a permission group has use limits, operations permitted by this group are only allowed if the use limits have not been exhausted. If a permission group has no use limits, these actions are always permitted.
    Each use limit specifies either an identifier for a counter or a time limit. If a permission group specifies both a counter and a time limit, the action will fail if either limit is exhausted.
    Performing any of the **`actions`** listed as action elements for this permission group decreases the count of the specified counter by 1 for each **`action`**.
  - one or more **`action`** elements. These specify the operations to which the use limits apply.
- typedef struct {
  M_Word flags;
  int n_limits;
  M_UseLimit *limits;
  int n_actions;
  M_Action *actions;
  M_KeyHash *certifier;
  M_KeyHashAndMech *certmech;
  M_ASCIIString *moduleserial;
  } M_PermissionGroup;

- The following **flags** are defined:
  - **PermissionGroup_flags_certmech_present** Set this flag if actions in this group must be certified with a key that matches the given hash and mechanism. [See 6].
  - **PermissionGroup_flags_certifier_present** Set this flag if actions in this group must be certified with a key that matches the given hash. [See 6] . If none of flags **PermissionGroup_flags_certifier_present**, **PermissionGroup_flags_certmech_present**, or **PermissionGroup_flags_NSOCertified** have been set, then this is a public permission group, and actions can be performed without a certificate.

    Note: The **PermissionGroup_flags_certifier_present** flag is included for backwards compatibility only. If you are creating a new ACL, use **PermissionGroup_flags_certmech_present**.
  - **PermissionGroup_flags_FreshCerts** Set this flag if the certificate must be freshly produced. If this flag is *not* set, certificates may be reused indefinitely.
  - **PermissionGroup_flags_moduleserial_present**Set this flag if the actions in this group can only be performed on a specific module, whose serial number matches the given serial number. [See 7] .
  - **PermissionGroup_flags_NSOCertified** Set this flag if the actions in this group must be certified by the Thales Security Officer's key K_{NSO}, whatever that is set to for this module at this time.

  Note: If you set more than one of **PermissionGroup_flags_certifier_present**, **PermissionGroup_flags_certmech_present**, or **PermissionGroup_flags_NSOCertified**, the module returns **Status_InvalidACL**.
- **n_limits** is the number of limits.
- ***limits** is a list of use limits, defined below.
  If more than one set of use limits is defined:
  - if the use limits are in the same permissions group, all counters and time limits must be valid, and all referenced counters are decreased by 1
  - if the use limits are in different permission groups, the module uses the first permission group that permits the action.
- **n_actions** is the number of actions.
- ***actions** is the list of actions to which the use limits apply.
- ***certifier** is either the hash of the key that is required to authorize the use of this ACL entry or a NULL pointer indicating that no further authorization is required.

  Note: The **certifier** field is included for backwards compatibility only. You are encouraged to use the **certmech** field. The **certifier** field may be removed in future releases.

***certmech**: **M_KeyHashAndMech** has the following structure:

```
typedef struct {
    M_KeyHash hash;
    M_PlainText mech;
} M_KeyHashAndMech;
```

- **hash** is the hash of the key that is required to authorize the use of this ACL entry or a **NULL** pointer, indicating that no further authorization is required.
- **mech** is the mechanism that is to be used to sign the certificate. You can specify **Mech_Any**, in which case the ACL will behave exactly as if you had used the certifier field.

Note: **Signingkey** certificates do not check the mechanism.

- **`*moduleserial`** is the serial number of the module on which the actions in this permission group must be performed. This must be the exact string returned by the **`NewEnquiry`** command for the module.

Use limits

```
Use limits
typedef struct {
    M_UseLim type;
    union M_UseLim__Details details;
} M_UseLimit;
```

The following **`Uselim types`** are defined:

- **`UseLim_Global`**
- **`UseLim_AuthOld`**
- **`UseLim_Time`**
- **`UseLim_NonVolatile`**
- **`UseLim_Auth`**

The **`details`** depend on the action type:

```
union M_UseLim__Details {
    M_UseLim_Global_Details global;
    M_UseLim_Time_Details time;
    M_UseLim_NonVolatile_Details nonvolatile;
    M_UseLim_Auth_Details auth;
};
```

A **`global`** use limit has the following structure:

```
typedef struct {
    M_LimitID id;
    M_Word max;
} M_UseLim_Global_Details;
```

- **`id`** is a unique 20-byte identifier for the counter for this use limit. When a counter is created, it is set to 0. Any time a user performs an action that requires a use limit, the module compares the value of the counter to the limit in the ACL. If the counter value is less than the limit, the action is permitted and the counter's value is increased by 1. Otherwise, the action is prohibited.
  Global and per-authorization counters are stored separately on the module. Therefore, a global use limit may have the same hash as a per-authorization use limit, and these hashes will refer to separate counters.
  Global counters are stored separately for each key, and per-authorization counters are stored separately for each logical token.

- This means that the two matching **LimitID** s will only refer to the same counter if either:
  - they are both in Global use limits in the same ACL
  - they are both in Auth use limits for keys loaded using the same logical token.
- **max** is the absolute maximum number of times that the actions specified in this permission group can be performed. Global limit counters are created when a key object is imported, generated or derived using the **DeriveKey** command. They are destroyed when that object is destroyed. They are never reset.
  When a key is duplicated (using the **Duplicate** command), or loaded with the **LoadBlob** command, all permission groups containing Global use limits are removed from its ACL. This is to ensure that actions subject to Global use limits can only be performed when the key was originally imported, generated or derived.

A **time** limit has the following structure:

```
typedef struct {
    M_Word seconds;
} M_UseLim_Time_Details;
```

- **seconds** is a per authorization limit that sets the length of time, in seconds, during which the actions specified in this permission group can be performed before the key needs to be reauthorized. Time limits only apply to keys protected by a logical token. The time is taken from the point at which the token was recreated.
  If you specify more than one time limit within an ACL, the shortest time limit will apply. If you specify a time limit and a use count limit, both must be valid in order for an action to be authorized.
  Note: If you apply a time limit to a key that is not loaded from a logical token protected blob, all permission groups with time limits will be unavailable and attempting to use these limits will return **Status_AccessDenied**.

nonvolatile limits are only available on nShield modules. The use limit is stored in a NVRAM file. A non-volatile limit has the following structure:

```
struct M_UseLim_NonVolatile_Details {
    M_UseLim_NonVolatile_Details_flags flags;
    M_FileID file;
    M_NVMemRange range;
    M_Word maxlo;
    M_Word maxhi;
    M_Word prefetch;
};
```

- No **flags** are defined.
- **file** is the **fileId** of the NVRAM file containing the use limit.
- **range** is the memory range within the file for this limit.
- **maxlo** and **maxhi** are the values for the limit stored as two 32-bit words.
- **prefetch**: In order to reduce the number of NVRAM write cycles, you can specify a number of limits to prefetch. The module will update the limit by this number and decrement an in-memory counter for each use. When the counter reaches zero the NVRAM value will again update the NVRAM.

A per-authorization use limit (**auth**) has the following structure:

```
typedef struct {
    M_LimitID id;
    M_Word max;
} M_UseLim_Auth_Details;
```

- **id** is a unique 20-byte identifier for the counter for this use limit. When a counter is created, it is set to 0. Any time a user performs an action that requires a use limit, the module compares the value of the counter to the limit in the ACL. If the counter value is less than the limit, the action is permitted and the counter's value is increased by 1. Otherwise, the action is prohibited.
  Global and per-authorization counters are stored separately on the module. Therefore, a global use limit may have the same hash as a per-authorization use limit, and these hashes will refer to separate counters.
  Global counters are stored separately for each key, and per-authorization counters are stored separately for each logical token.
  This means that the two matching **LimitID**s will only refer to the same counter if either:
  - they are both in Global use limits in the same ACL
  - they are both in Auth use limits for keys loaded using the same logical token.
- **max** is the number of times that the actions specified in this permission group can be performed before the logical token needs to be reauthorized.
  Per-authorization limit counters are created when a key is loaded from a token blob, unless a counter with the same **LimitID** already exists for this token (in which case, the existing counter is used). This can mean that all the per-authorization use limits for a key have been exhausted already when it is loaded. In such a case, you must reload the logical token.
  Keys that have been loaded from blobs under different tokens have separate counters even if they have the same **LimitID**.
  Firmware versions 2.12.0 or later contain logic to prevent an attacker loading the same logical token twice and thereby gaining two separate sets of counters. It works as follows:
  Every time a smart card is inserted, all the logical token shares on it are marked **available**. When a share is loaded for use in a logical token, it is marked **used**, unless the **ReadShare** command sets the **UseLimitsUnwanted** flag.
  If any share is loaded - locally or remotely - when it is already marked **used**, the logical token is marked **UseLimitsUnavailable**. No per-authorization use limits are allowed for any keys loaded using this second logical token. This ensures only one set of use limits counters can be created for each physical insertion of a token.
  Note: The mechanism for controlling per-authorization limits changed in firmware 2.12.0 to prevent a possible attack which may have resulted in the limit being circumvented. On new firmware ACLs using **UseLim_Auth** and **UseLimAuth_Old** both use the new mechanism. However, the **nfkmverify** program will note use of the old style limit as this will use the old behavior on old firmware.
  Although it is possible to load a logical token on several modules, using remote slots, only one copy of the logical token can be allocated the per-authorization use limits.

Actions

```
typedef struct {
    M_Act type;
    union M_Act__Details details;
} M_Action;
```

type must be one of the actions listed below:

- **Act_NoAction=**
- **Act_OpPermissions=** - see *OpPermissions* on page 113
- **Act_MakeBlob=** - see *MakeBlob* on page 114
- **Act_MakeArchiveBlob=** - see *MakeArchiveBlob* on page 115
- **Act_NSOPermissions=** - see *NSO* on page 116
- **Act_DeriveKey=** - see *DeriveKey* on page 119
- **Act_NVMemOpPerms=** - see *NVRAM* on page 117
- **Act_FeatureEnable=** - see *NVRAM* on page 117
- **Act_NVMemUseLimit=**
- **Act_SendShare=** - see *SendShare* on page 118
- **Act_ReadShare=** - see *ReadShare* on page 118
- **Act_StaticFeatureEnable=**
- **Act_UserAction=** - see *UserAction* on page 119
- **Act_FileCopy=** - see *FileCopy* on page 118

details depend on the chosen action type:

```
union M_Act__Details {
    M_Act_FeatureEnable_Details featureenable;
    M_Act_DeriveKey_Details derivekey;
    M_Act_SendShare_Details sendshare;
    M_Act_NVMemUseLimit_Details nvmemuselimit;
    M_Act_NVMemOpPerms_Details nvmemopperms;
    M_Act_StaticFeatureEnable_Details staticfeatureenable;
    M_Act_NSOPermissions_Details nsopermissions;
    M_Act_OpPermissions_Details oppermissions;
    M_Act_FileCopy_Details filecopy;
    M_Act_MakeArchiveBlob_Details makearchiveblob;
    M_Act_MakeBlob_Details makeblob;
    M_Act_UserAction_Details useraction;
    M_Act_ReadShare_Details readshare;
};
```

Action types

OpPermissions

```
typedef struct {
    M_Word perms;
} M_Act_OpPermissions_Details;
```

The following flags (**perms**) are defined:

- **Act_OpPermissions_Details_perms_DuplicateHandle**: Setting this flag grants permission to create a copy of the key with the same ACL. Duplicating a key does not enable you to perform any further actions, because both copies use the same use counters.
- **Act_OpPermissions_Details_perms_UseAsCertificate**: Setting this flag allows use of the **KeyID** to authorize a command that requires a certificate.
- **Act_OpPermissions_Details_perms_ExportAsPlain**
- **Act_OpPermissions_Details_perms_GetAppData**
- **Act_OpPermissions_Details_perms_SetAppData**
- **Act_OpPermissions_Details_perms_ReduceACL**
- **Act_OpPermissions_Details_perms_ExpandACL**
- **Act_OpPermissions_Details_perms_Encrypt**
- **Act_OpPermissions_Details_perms_Decrypt**
- **Act_OpPermissions_Details_perms_Verify**
- **Act_OpPermissions_Details_perms_UseAsBlobKey**: Setting this flag allows use of this key either in the **MakeBlob** command to encrypt a key blob or in the **LoadBlob** command to decrypt a key from a blob.
- **Act_OpPermissions_Details_perms_UseAsKM**: Only **DES3** keys can be used for module keys, $\mathbf{K}_{M}$.
- **Act_OpPermissions_Details_perms_UseAsLoaderKey**: When this flag is set, an encryption key is only permitted to perform decryption when loading an SEE machine or SEE World onto the module.
- **Act_OpPermissions_Details_perms_Sign**
- **Act_OpPermissions_Details_perms_GetACL**
- **Act_OpPermissions_Details_perms_SignModuleCert**
- **Act_OpPermissions_Details_perms__allflags**

MakeBlob

This action type allows the creation of module key, or token, key blobs with the given key (see also **MakeArchiveBlob**).

```
typedef struct {
    M_Word flags;
    M_KMHash *kmhash;
    M_TokenHash *kthash;
    M_TokenParams *ktparams;
    M_MakeBlobFilePerms *blobfile;
} M_Act_MakeBlob_Details;
```

- The following **flags** are defined:
  - **Act_MakeBlob_Details_flags_AllowKmOnly**
    If this flag is set, you can create blobs directly under a module key or under a logical token. If this flag is *not* set, you must use a logical token.
  - **Act_MakeBlob_Details_flags_AllowNonKm0**
    If this flag is set, you can create blobs for this key using module keys, or logical tokens based on module keys, except for the internally generated $\mathbf{K}_{M0}$. If this flag is *not* set, you must use $\mathbf{K}_{M0}$ or logical tokens based on $\mathbf{K}_{M0}$.
  - **Act_MakeBlob_Details_flags_kmhash_present**

Set this flag in order to restrict the blobs that can be made with this key to blobs that use the module key whose hash is specified or to logical tokens that are based on this module key. If this flag is *not* set, any module key may be used. If this hash is not K_{M0}, you must set the **AllowNonKM0** flag.

- **Act_MakeBlob_Details_flags_kthash_present**
  Set this flag in order to restrict the blobs that can be made with this key to blobs that use the token whose hash is specified. If this flag is *not* set, any token may be used. If this token is not based on K_{M0}, you must set the **AllowNonKM0** flag.
- **Act_MakeBlob_Details_flags_ktparams_present**
  Set this flag in order to restrict the blobs that can be made with this key to blobs that use a token with either the given parameters or with more restrictive ones. If this flag is *not* set, any token can be used.
- **Act_MakeBlob_Details_flags_AllowNullKmToken**
  If this flag is set, the user can create token blobs for this key with a token protected by the null module key.
- **Act_MakeBlob_Details_flags_blobfile_present**
  If this flag is set the blob will be stored in the NVRAM or smart card file specified - it will not be returned to the host.
- **Act_MakeBlob_Details_flags__allflags**

Note: The key blob must meet the requirements of all the flags.

- ***kmhash** - see **Act_MakeBlob_Details_flags_kmhash_present** above.
- ***kthash** - see **Act_MakeBlob_Details_flags_kthash_present** above.
- ***ktparams** - see **Act_MakeBlob_Details_flags_ktparams_present** above.
- ***blobfile**
  The following structure specifies the NVRAM or smart card files to which you want to restrict writing the blob.

```
struct M_MakeBlobFilePerms {
    M_MakeBlobFilePerms_flags flags;
    M_PhysToken *devs;
    M_KeyHash *aclhash;
};
```

  - The following **flags** are defined:
    - **MakeBlobFilePerms_flags_devs_present**
    - If set, the blob may only be stored in the storage devices specified by the **M_FileDeviceFlags** word.
    - **MakeBlobFilePerms_flags_aclhash_present**
      Set this flag if the structure contains a **M_KeyHash**.
  - ***devs** is the device on which to store the blob.
  - ***aclhash** is the hash of a Template Key defining the ACL to use for the file storing the key. The key must be provided when making the blob.

  If you want to restrict the making of blobs to a set of module keys, or to a set of tokens, then you must include a **MakeBlob** entry for each module or token hash.

MakeArchiveBlob

This action type allows the creation of direct and indirect archive key blobs with the given key.

```
typedef struct {
    M_Word flags;
    M_PlainText mech;
    M_KMHash *kahash;
    M_MakeBlobFilePerms *blobfile;
} M_Act_MakeArchiveBlob_Details;
```

- The following **flags** are defined:
  - **Act_MakeArchiveBlob_Details_flags_kahash_present**
    If this flag is set, you can make an archive key blob for this key with the key whose hash is specified. If this flag is *not* set, any archive key may be used.

    Including an **Act_MakeArchiveBlob** entry without **kahash_present** in an open permission group creates a security loophole.

  - **Act_MakeArchiveBlob_Details_flags_blobfile_present**
    If this flag is set the blob will be stored in the NVRAM or smart card file specified - it will not be returned to the host.
- **mech**
  For making direct archive blobs, this must be **Mech_DES3mCBCi64pPKCS5** or **Mech_Any**; for indirect blobs this specifies the mechanism which must be used to encrypt the session key. If set to **Mech_Any**, any mechanism appropriate for the type of the archiving key is allowed. See *Mechanisms* on page 64.
- ***kahash** is the key hash.
- ***blobfile** — see *MakeBlob* on page 114

NSO

This action type is used only in certificates that approve critical functions that have been defined in the **SetKNSO** command. It should not be used in an ACL for a key.

```
typedef struct {
    M_NSOPerms perms;
} M_Act_NSOPermissions_Details;
```

M_NSOPerms has the following structure:

```
typedef struct {
    M_Word ops;
} M_NSOPerms;
```

The following flags (**ops**) are defined. These are identical to those used in the **SetNSOPerms** command.

- **NSOPerms_ops_LoadLogicalToken**
- **NSOPerms_ops_ReadFile**
- **NSOPerms_ops_WriteShare**

- **NSOPerms_ops_WriteFile**
- **NSOPerms_ops_EraseShare**
- **NSOPerms_ops_EraseFile**
- **NSOPerms_ops_FormatToken**
- **NSOPerms_ops_SetKM**
- **NSOPerms_ops_RemoveKM**
- **NSOPerms_ops_GenerateLogToken**
- **NSOPerms_ops_ChangeSharePIN**
- **NSOPerms_ops_OriginateKey**
- **NSOPerms_ops_NVMemAlloc**
- **NSOPerms_ops_NVMemFree**
- **NSOPerms_ops_GetRTC**
- **NSOPerms_ops_SetRTC**
- **NSOPerms_ops_DebugSEEWorld**
- **NSOPerms_ops_SendShare**
- **NSOPerms_ops_ForeignTokenOpen**
- **NSOPerms_ops__allflags**

NVRAM

This action type allows operations to be performed upon files that have been stored in the nonvolatile memory or on a smart card or soft token.

```
struct M_Act_NVMemOpPerms_Details {
    M_Act_NVMemOpPerms_Details_perms perms;
    M_NVMemRange *subrange;
    M_NVMemRange *exactrange;
    M_Word *incdeclimit;
};
```

- The following operations (**perms**) are defined.
  - **Act_NVMemOpPerms_Details_perms_Read**
  - **Act_NVMemOpPerms_Details_perms_Write**
  - **Act_NVMemOpPerms_Details_perms_Incr**
  - **Act_NVMemOpPerms_Details_perms_Decr**
  - **Act_NVMemOpPerms_Details_perms_BitSet**
  - **Act_NVMemOpPerms_Details_perms_BitClear**
  - **Act_NVMemOpPerms_Details_perms_Free**
  - **Act_NVMemOpPerms_Details_perms_subrange_present**
  - **Act_NVMemOpPerms_Details_perms_exactrange_present**
  - **Act_NVMemOpPerms_Details_perms_incdeclimit_present**
  - **Act_NVMemOpPerms_Details_perms_GetACL**
  - **Act_NVMemOpPerms_Details_perms_LoadBlob**
    This permission allows the contents to be used as a blob by the **Loadblob** command.
  - **Act_NVMemOpPerms_Details_perms_Resize**
- ***subrange**
  This specifies the subrange to which this operation can be applied; the operation can apply to any part of the specified range in the ACL.
- ***exactrange**

This is a subrange to which this operation can be applied only if the range exactly matches the specified range in the ACL.

- **`*incdeclimit`**
  This is the maximum amount that this range can be increased or decreased in one operation.

ReadShare

This action type enables a logical token share to be read normally using the **`ReadShare`** command.

```
typedef struct {
    M_ReadShareDetails rsd;
} M_Act_ReadShare_Details;
```

```
typedef struct {
    M_ReadShareDetails_flags flags;         No flags are defined
} M_ReadShareDetails;
```

SendShare

This action type enables a logical token share to be read remotely and sent over an impath.

```
typedef struct {
    M_Act_SendShare_Details_flags flags;
    M_RemoteModule *rm;
    M_ReadShareDetails *rsd;
} M_Act_SendShare_Details;
```

- The following **`flags`** are currently defined:
  - **`Act_SendShare_Details_flags_rm_present`**
    This flag is set if the action contains a **`RemoteModule`** structure.
  - **`Act_SendShare_Details_flags_rsd_present`**
    This flag is set if the action contains a **`ReadShareDetails`** structure.
- **`*rm`**
  The impath over which the share data is to be sent must match this **`RemoteModule`** structure.
- **`*rsd`** — see *ReadShare* on page 118

FileCopy

This action permits files stored on a smart card, soft token or in NVRAM to be copied to another location. The action specifies which location the file can be copied to and from.

```
struct M_Act_FileCopy_Details {
    M_Act_FileCopy_Details_flags flags;
    M_PhysToken to;
    M_PhysToken from;
};
```

The following **flag** is defined:**Act_FileCopy_Details_flags_ChangeName**.

If set the new file may have a different **FileID** from the original file.

UserAction

This action does not permit any operations. Instead it can be checked by the **CheckUserAction** command. This enables applications to make use of all modules ACI checking features - including use limits, time limits, certifiers and so on - to restrict actions in their own code.

```
struct M_Act_UserAction_Details {
    M_UserActionInfo allow;
};
```

DeriveKey

This action type enables the key to be used in the **DeriveKey** command. It allows the key to be used in a single specific role. If you want to create a key that can be used in more than one role, you must include a separate action entry for each role.

```
typedef struct {
    M_Word flags;
    M_DeriveRole role;
    M_DeriveMech mech;
    int n_otherkeys;
    M_KeyRoleID *otherkeys;
    M_DKMechParams *params
} M_Act_DeriveKey_Details;
```

- The following **flags** are defined:
  - **Act_DeriveKey_Details_flags_params_present**
- **role** can be one of the following:
  - **DeriveRole_TemplateKey** (template)
  - **DeriveRole_BaseKey** (base key)
  - **DeriveRole_WrapKey** (wrapping key)
- **mech** — see *Mechanisms* on page 64.
- **n_otherkeys** - the number of keys in the **role** table
- ***otherkeys**
  The following keys can be used in the other roles of the DeriveKey command:

  ```
  typedef struct {
      M_DeriveRole role;
      M_KeyHash hash;
  } M_KeyRoleID;
  ```

  - **role**
    You can define keys for any or all of the roles. You can specify one or more keys for each role. If you do not specify a key for a particular role, then any key can be used in that role.
  - **hash** is an SHA-1 hash.
- ***params**
  The mechanism parameters to use for the **DeriveKey** operation.

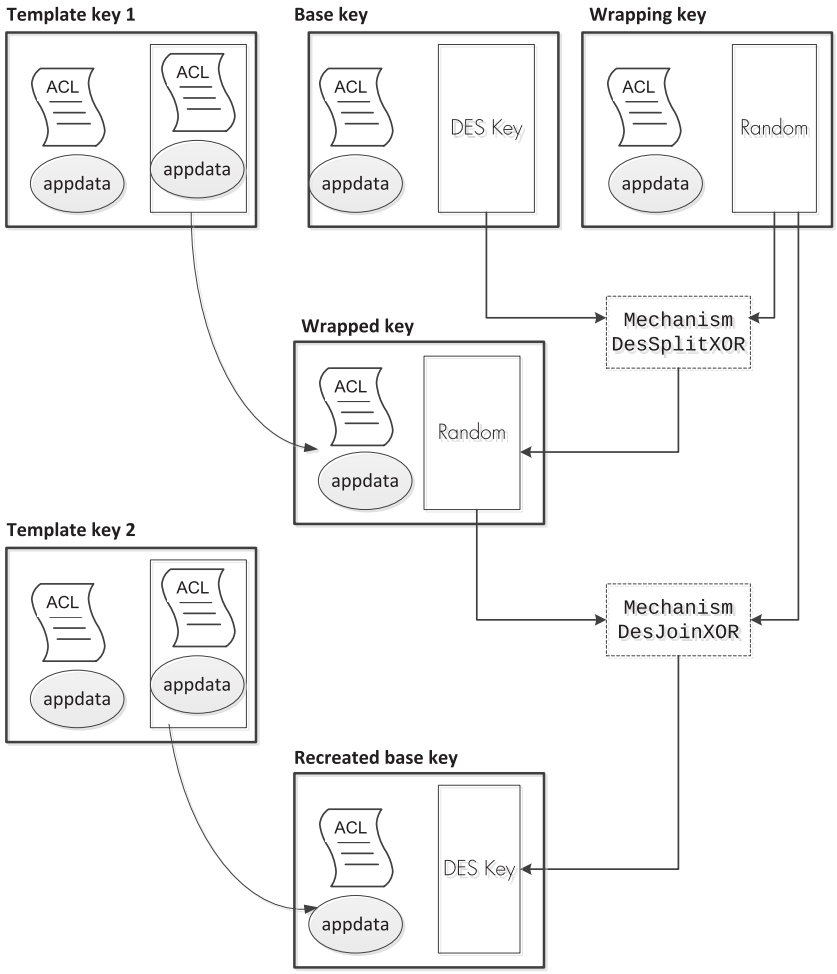
```
struct M_DKMechParams {
    M_DeriveMech mech;
    union M_DeriveMech__DKParams params;
};
```

- **mech**
  The mechanism to use. The module will not permit you to set a **M_DKMechParams** with a mechanism that is different to that previously defined in the ACL. If you attempt this the module returns **Status_InvalidACL**.
- **params**
  The derive key mechanism parameters— see *Derive Key Mechanisms* on page 97.
  The module applies the following rules to determine which derive key operations are permitted:
  - If any of the requested or allowed **DeriveMech** values mismatch, the operation is never allowed.
  - If the allowed **DKMechParams** are not present, any requested parameters are allowed.
  - If the mechanism has an empty **DKParams**, the operation is allowed
  - If the mechanism is of type **DeriveMech_KDPKeyWrapDES3**, the operation is allowed if and only if all of the following are true:
    - the **ExactPolicyMatch** flag is set
    - the requested kind and details fields exactly match the corresponding allowed fields, or the allowed kind field is **0xFFFFFFFF**.
    - the requested duration is nonzero and less than the allowed duration, or the allowed duration field is zero.
  - For other mechanisms, this comparison is not at present defined. The module will return **NotYetImplemented** for attempts to set, in a key's ACL, **DKMechParams** with mechanisms for which this is the case.

Using DeriveKey — an example

The following example shows how to use the **DeriveKey** command to split a DES key into two random halves and then recombine these halves to recreate the original key. This process is illustrated in Figure 15 below.

Figure 15. DeriveKey process



First, import the two template keys. A template key contains an ACL and an **`Appdata`** file that can be applied to the results of the **`DeriveKey`** operation. By importing these elements first, their key hashes can be determined, and these hashes can be referenced in the ACLs for the remaining keys. This ensures that the two derived keys will have the correct ACL.

Next, create the wrapping key. Determine its hash and then that of the base key. After all the input keys have been have created, use the **`DeriveKey`** command to combine the base key and the wrapping key.

Finally, unwrap the wrapped key. Check that the new DES key has the same hash, and therefore the same data, as the original. Also check that the new DES key has correctly inherited the ACL and application data from the template key.

1. Use the **`Import`** command with the following parameters to import a template key for an ACL that allows the use of **`DeriveKey`** with this key as the base key, any mechanism, and any other keys:

module	1
type	DKTemplate
appdata	02020202
nested_acl	01000000 00000000 00000000 02000000 01000000 6c200000 05000000 00000000 01000000 00000000 00000000
ACL	
n_groups	1
groups[0]	
flags	0x0
n_limits	0
n_actions	1
actions[0]	
type	DeriveKey
flags	0x0
role	TemplateKey
mech	Any
n_otherkeys	0
appdata	0

Note: Create the **`nested_acl`** by using the **`NFastApp_MarshalACL()`** command.
Such use of the Import command will return:

```
idka=                                   IDKA 0010
```

2. Get this key's hash by using the **`GetKeyInfo`** command with the following parameters:

```
flags;          0x0
key;            IDKA 0010
```

This command returns:

```
type;           DKTemplate
hash;           HKA 0010
```

3. Use the **Import** command with the following parameters to import a template key for an ACL that contains **oppermissions** as follows:
   - **ExportAsPlain GetAppData Encrypt Decrypt Verify Sign GetACL**

module	1
type	DKTemplate
appdata	01010101
nested_acl	01000000 00000000 00000000 01000000 01000000 8c330000
ACL	
n_groups	1
groups[0]	
flags	0x0
n_limits	0
n_actions	1
actions[0]	
type	DeriveKey
flags	0x0
role	TemplateKey
mech	Any
n_otherkeys	0
appdata	0

Such use of the Import command will return:

```
key             IDKA 0011
```

In order to create a **nested_acl** in C, use the **NFastApp_MarshalACL()** command.
In order to create a nested ACL in Java, use the **marshall()** method from the **M_ACL** class. The following Java fragment demonstrates the use of this method:

```
...
    M_ACL acl;
    MarshallContext tempMctx;
    M_ByteBlock bb;
    M_KeyType_Data_Template data;

    acl = yourACL;
    tempMctx = new MarshallContext();
    acl.marshall(tempMctx);
    bb = new M_ByteBlock (tempMctx.getBytes());
    data = new M_KeyType_Data_Template();
    data.nested_acl = bb;
...
```

4. Get this key's hash by using the **`GetKeyInfo`** command with the following parameters:

```
key       IDKA 0011
```

This command returns:

```
type          DKTemplate
hash          HKA 0011
```

5. Make a wrapping key by using the **`GenerateKey`** command:
   - when wrapping, insist on using template *HKA 0010*
   - when unwrapping, insist on using template *HKA 0011*

flags	0x0
module	1
type	Random
lenbytes	8
ACL	
n_groups	1
groups[0]	
flags	0x0
n_limits	0
n_actions	3
actions[0]	
type	OpPermissions

perms	`DuplicateHandle`
	`ExportAsPlain`
	`ReduceACL`
	`GetACL`

actions[1]	
type	DeriveKey
flags	0x0
role	WrapKey
mech	DESsplitXOR
n_otherkeys	1
role	TemplateKey
hash	HKA 0010
actions[2]	
type	DeriveKey
flags	0x0
role	WrapKey
mech	DESjoinXOR
n_otherkeys	1
role	TemplateKey
hash	HKA 0011

Such use of the **`GenerateKey`** command returns:

```
key         IDKA 0012
```

6. Get this key's hash by using the GetKeyInfo command:

```
key         IDKA 0012
```

This command returns:

```
type        Random
hash            HKA 0012
```

7. Use the **GenerateKey** command with the following parameters to generate a DES key that can only be wrapped using:
   - the **DESsplitXOR** mechanism
   - *HKA 0012* as the wrapping key

module	1
type	DES
n_groups	1
groups[0]	
flags	0x0
n_limits	0
n_actions	2
actions[0]	
type	OpPermissions
perms	ReduceACL GetACL
actions[1]	
type	DeriveKey
flags	0x0
role	BaseKey
mech	DESsplitXOR
n_otherkeys	1
role	WrapKey
hash	HKA 0012

This returns:

```
key             IDKA 0013
```

8. Get this key's hash by using the **GetKeyInfo** command with the following parameters:

```
key             IDKA 0013
```

This command returns:

```
type            DES
hash                HKA 0013
```

9. The DES key can now be combined with the random key to produce a second random key by using the **DeriveKey** command with the following parameters:

flags	0x0
mech	DESsplitXOR
n_keys	3
keys[0]	IDKA 0010
keys[1]	IDKA 0013
keys[2]	IDKA 0012

This command returns:

```
key             IDKA 0014
```

At this point, this process has produced:

- a **DES** key *IDKA 0013*
- two random keys: *IDKA 0012* and *IDKA 0014*.

The two random keys can be combined to recreate the key data in the DES key. This can be demonstrated by combining the keys that use the **DeriveKey** command and then using the GetKeyInfo to check that the hash of the new key matches the hash of the DES key that was determined in [Step 8].

10. Use the **DeriveKey** command with the following parameters to combine the keys:

flags	0x0
mech	DESjoinXOR
n_keys	3
keys[0]	IDKA 0011
keys[1]	IDKA 0014
keys[2]	IDKA 0012

This command returns:

```
key         IDKA 0015
```

Note: This is a new **KeyID** because this is a new instance of the key. This instance of the key has taken its **appdata** and **ACL** from the template key that was created earlier: **IDKA 0011**.

11. Get the hash of this new key by using the **GetKeyInfo** command with the following parameters:

```
key         IDKA 0015
```

This command returns:

```
type        DES
hash              HKA 0013
```

Note: This is the same hash as before, which proves that the key has been combined correctly.

12. Check that the new key has inherited the application data from the template key by using the **GetAppData** command with the following parameters:

```
key         IDKA 0015
```

This command returns:

```
appdata     01010101
```

This is the application data that was provided by the template key.

13. Check that the new key has inherited the ACL from the nested ACL in the template key by using the **GetACL** command with the following parameters:

```
key         IDKA 0015
```

This command returns:

acl.n_groups	1
groups[0]	
flags	none 0x00000000
n_limits	0
n_actions	1
actions[0]	
type	OpPermissions
perms	ExportAsPlain GetAppData Encrypt Decrypt Verify Sign GetACL

Certificates

The Thales module uses certificates to enable a given user to authorize another user to perform an action.

A certificate is a signed message. The key that is used to sign the certificate must match the key hash in the ACL it is authorizing. The message can contain an ACL; this ACL can be used to restrict the operation to be approved by the certificate, or it can be used to require a further certificate.

Certificates can be either fresh or reusable.

A fresh certificate includes a challenge value. This is a random number that was generated previously by the module with the **`GetChallenge`** command. The module stores the eight most recent challenges for which the associated certificates have not yet been presented.

If there are eight outstanding challenges and a user issues the **`GetChallenge`** command, the module deletes the oldest challenge, and any certificate presented that contains that challenge value will be rejected.

When a user presents a fresh certificate, the module deletes the matching challenge from the list. If the same certificate is presented a second time, it will be rejected.

The list of challenges is cleared whenever the unit is reset.

Therefore, a fresh certificate:

- can only be used once
- must be used on the module that generated the challenge
- may become invalid if left too long before it is used

If you submit a certificate containing a challenge that is not on the module's list of current challenges, the server returns the status **Status_UnknownChallenge**.

A reusable certificate does not contain a challenge and can be used as often as is required. It can also be used on any module.

An ACL can specify that the required certificate must be fresh. If you present a reusable certificate when the ACL requires a fresh certificate, the certificate will be rejected.

If you possess the required key, you can always create a fresh certificate. However, this requires a certain amount of processing, both on the module and on the host. In order to prevent unnecessary load, you can authorize a command by presenting a certificate that contains the required key's **KeyID** . In order for this certificate to be valid, you must have loaded the key yourself. You cannot pass the **KeyID** to another user. In order to authorize another user, you must create a properly signed certificate.

Code executing in the SEE can be signed by one or more keys by using the signature tools provided with the **CodeSafe Developer Kit**. By presenting a certificate of the type **CertType_SEECert**, code signed in this way can perform any operation for which the signing key has permission.

Using a certificate to authorize an action

If you are given a certificate, you must include it with the command it authorizes, after all the arguments for that command.

For situations in which you are presenting a single certificate:

- it must not require further authorization
- the hash of the key that signed the certificate must match the hash that is specified in the ACL

For situations in which you need to present a chain of certificates, the first certificate must not require any further authorization. For every certificate in the chain, the module checks to see that the hash of the signing key matches the hash given in the certifier field of the ACL that is included in the next certificate or, if this is the last certificate in the chain, the certifier field of the ACL for the key being authorized. The ACL in each certificate in the chain must permit the operation to be performed.

If a certificate, or any certificate in a certificate chain, does not authorize the requested action, the module will return the status **Status_AccessDenied**.

Generating a certificate to authorize another operation

It is the responsibility of the cryptographic application to build certificates. This process is assisted by the **NFast_BuildCmdCert()** function that is provided in the generic stub library.

Structure

```
typedef struct {
    M_KeyHash keyhash;
    M_CertType type;
    union M_CertType__CertBody body;
} M_Certificate;
```

- **keyhash** is the hash of the key that is used to sign the certificate. This hash must match the hash that is specified in the key's ACL or in the previous certificate in the chain.
- The following **type** values are defined:
  - **CertType_Invalid**
  - **CertType_SigningKey**
  - **CertType_SingleCert**
  - **CertType_SEECert**
- The certificate body (**body**) has one of the following formats:

```
union M_CertType__CertBody {
    M_CertType_SigningKey_CertBody signingkey;
    M_CertType_SingleCert_CertBody singlecert;
};
```

- A **signingkey** has the following body:

```
typedef struct {
    M_PlainText key;
} M_CertType_SigningKey_CertBody;
```

- Where:
  - **key** is the **KeyID** of the key that must be loaded in order to authorize this command. The key must have the following properties:
  - the hash of the key must match the hash that was given in the ACL
  - the key must have **UseAsCertificate** permission set in its ACL in an open group.
- A **singlecert** certificate has the following body:

```
typedef struct {
    M_PlainText pubkeydata;
    M_CipherText signature;
    M_ByteBlock certsignmsg;
} M_CertType_SingleCert_CertBody;
```

  - **signature** is the **certsignmsg**, which is signed with the private key that corresponds to **pubkeydata**.
- A **certsignmsg** has the following structure, which must be marshalled into a byte block:

```
typedef struct {
    M_MagicValue header;
    M_Word flags;
    int n_hks;
    M_KeyHash *hks;
    M_Nonce *nonce;
    M_ACL *acl;
    M_MagicValue footer;
} M_CertSignMessage;
```

- **header**
  This must be set to the value **MagicValue_CertMsgHeader**, defined in **messages-ags-dh.h**.
- **flags**
  The following flags are defined:
  - **CertSignMessage_flags_nonce_present**
  - **CertSignMessage_flags_acl_present**
  - **CertSignMessage_flags_do_not_cache**
- **n_hks** and ***hks**
  This table can be used to restrict the keys to which this certificate applies. If there are entries in this table, then the hash of the key object used—or, for an NSO certificate, the hash of the module key used—must also be in this table. If the table is empty (**n_hks = 0**), then the certificate can be used to authorize any operations on a key with a matching ACL.
- ***nonce**
  This is a nonce returned by the **GetChallenge** command.
- ***acl**
  Optionally, this is a valid ACL that authorizes the action to be performed. If this ACL contains a **certmech** or a **certifier** field in a permission group, then a valid certificate signed by the key whose hash is in the permission group must precede this certificate in the chain.
- **footer**
  This must be set to the value **MagicValue_CertMsgFooter**, defined in **messages-ags-dh.h**.

The **certsgnmsg** block should be passed to a suitable signature algorithm. For RSA signature keys, use a mechanism that hashes the block first (for example, **RSAhSHA1pPKCS1**).

The module checks all of the above and returns:

- **Status_BadCertKeyHash** if the verification key does not match the given hash
- **Status_VerifyFailed** if the signature cannot be verified with the given key
- **Status_UnknownChallenge** if the nonce was not one that the module had issued recently
- **Status_AccessDenied** if the ACL still does not permit your request for some other reason.

The certificate type **CertType_SEECert**, however, has an empty **CertBody** . In order to use certificates of this type:

1. Specify in the **M_Certificate** structure the hash of the signing key that was used to sign the SEE World data that authorized the action.
2. The access control system checks to ensure that the SEE World data was, in fact, signed by the specified key.
3. If so, the certificate is accepted much as a **signingkey** certificate would be. However, because a **signingkey** certificate is always treated as fresh but an SEE certificate is not, the flag **PermissionGroup_flags_FreshCerts** must not be set in the next ACL in the stack.

Thus, code executing within the SEE can authorize itself to perform an action requiring authorization from a key that signed the code. It can do this by creating an **`M_Certificate`**, setting its key hash appropriately, and setting its type to **`SEECert`**.

Appendix C: NKFM Functions

This chapter describes the functions and structures that are used in the C NFKM library. This library gives access to security world key-management functions.

Debugging NFKM functions

Most of the NFKM functions that are described in this chapter can write data to a debug or error log. However, they do not usually do so except under circumstances outside of those encountered during normal operation (for example, if the module is not properly initialized). You can control the writing of data to a debug or error log with the **NFKM_LOG** environment variable. For more information on the **NFKM_LOG** environment variable, see the User Guide.

Use the **NFKM_getinfo** call to get the current state before using any other call that relies on the data in the **NFKM_SlotInfo** structure being up-to-date.

Functions

Several operations, especially card set creation and loading, require multiple function calls. In this case there is usually a ***_begin** function which must be called first. There is a ***_nextxxx** function that can be called a number of times. Finally there is a ***_done** function. If, due to user input you decide not to complete the operation there is a ***_abort** function which clears up memory.

NKFM_changepp

Change the pass phrase on a card.

```
M_Status NFKM_changepp(
                                NFast_AppHandle app,
                                NFastApp_Connection conn,
                                const NFKM_SlotInfo *slot,
                                unsigned flags,
                                const M_Hash *oldpp,
                                const M_Hash *newpp,
                                NFKM_ShareFlag remove,
                                NFKM_ShareFlag set,
                                struct NFast_Call_Context *cctx
);
```

- **const NFKM_SlotInfo *slot** is the slot in which the card is loaded
- **unsigned flags** is a flags word, the following flag is defined:

```
#define NFKM_changepp_flags_NoPINRecovery 1u
```

- **const M_Hash *oldpp** is a pointer to the current pass phrase hash
- **const M_Hash *newpp** is a pointer to the new pass phrase hash
- **NFKM_ShareFlag remove** is a list of shares whose pass phrases you want to remove, regardless of newpp
- **NFKM_ShareFlag** set is a list of shares whose pass phrases you want to set or change.

Note: The **remove** and **set** flags must be disjoint. A default appropriate to the type of card in the slot is used if both **remove** and **set** are zero.

NFKM_checkconsistency

This function checks the general consistency of the security world data:

```
M_Status NFKM_checkconsistency(
                                        NFast_AppHandle app,
                                        NFKM_DiagnosticContextHandle callctx,
                                        NFKM_diagnostic_callback *informational,
                                        NFKM_diagnostic_callback *warning,
                                        NFKM_diagnostic_callback *fatal,
                                        struct NFast_Call_Context *cctx
);
```

It returns **Status_OK** unless:

- there was a fatal error, in which case it returns the return value from **fatal()**, which *must* be nonzero
- any other diagnostic callback returned nonzero, in which case it returns that callback's return value (because checking was aborted at that point).

NFKM_checkpp

Verifies that a pass phrase is correct for a given card. Each share on the card which has a pass phrase set is checked.

```
M_Status NFKM_checkpp(
                                NFast_AppHandle app,
                                NFastApp_Connection conn,
                                const NFKM_SlotInfo *slot,
                                const M_Hash *pp,
                                struct NFast_Call_Context *cctx
);
```

NFKM_cmd_generaterandom

Utility function: calls the nCore **GenerateRandom** command. Requires an **app** handle and an existing connection.

```
M_Status NFKM_cmd_generaterandom(
                                          NFast_AppHandle app,
                                          NFastApp_Connection conn,
                                          M_Word wanted,
                                          unsigned char **block_r,
                                          struct NFast_Call_Context *cctx
);
```

Sets ***block_r** to point to newly allocated memory containing the random data.

NFKM_cmd_destroy

Utility function: calls the nCore **Destroy** command to destroy an nCore object. Requires an **app** handle and an existing connection.

```
M_Status NFKM_cmd_destroy(
                                    NFast_AppHandle app,
                                    NFastApp_Connection conn,
                                    M_ModuleID mn,
                                    M_KeyID idka,
                                    const char *what,
                                    struct NFast_Call_Context *cctx
);
```

The **what** argument should describe what sort of thing you are destroying, for the benefit of people reading log messages created when things go wrong.

NFKM_cmd_loadblob

Utility function: calls the nCore **Loadblob** command to load a blob. Requires an **app** handle and an existing connection.

```
M_Status NFKM_cmd_loadblob(
                                     NFast_AppHandle app,
                                     NFastApp_Connection conn,
                                     M_ModuleID mn,
                                     const M_ByteBlock *blob,
                                     M_KeyID idlt,
                                     M_KeyID *idk_r,
                                     const char *whatfor,
                                     struct NFast_Call_Context *cctx
);
```

Set **idlt** to zero if the blob is module-only.

The **whatfor** argument should describe what blob you are loading, for the benefit of people reading log messages created when things go wrong.

NFKM_cmd_getkeyplain

Utility function: calls the nCore **Export** command to obtain the plain text of a key object. Requires an **app** handle and an existing connection.

```
M_Status NFKM_cmd_getkeyplain(
                                        NFast_AppHandle app, NFastApp_Connection
                                        conn,
                                        M_ModuleID mn,
                                        M_KeyID idka,
                                        M_KeyData *keyvalue_r,
                                        const char *what,
                                        struct NFast_Call_Context *cctx
);
```

The **what** argument should describe what sort of key you are querying the plain text of, for the benefit of people reading log messages created when things go wrong.

When you've finished with the exported key data, call **NFastApp_Free_KeyData** on it.

NFKM_erasecard

This function erases an operator card in the given slot:

```
M_Status NFKM_erasecard(
                                    NFast_AppHandle app,
                                    NFastApp_Connection conn,
                                    const NFKM_SlotInfo *slot,
                                    NFKM_FIPS140AuthHandle fips140auth,
                                    struct NFast_Call_Context *cctx
);
```

NFKM_erasemodule

Erases a module. The module must be in (pre-)init mode. All NSO permissions are granted, and the security officer's key is reset to its default.

```
M_Status NFKM_erasemodule(
                                      NFast_AppHandle app,
                                      NFastApp_Connection conn,
                                      const NFKM_ModuleInfo *m,
                                      struct NFast_Call_Context *cc
);
```

- **const NFKM_ModuleInfo *m** is a pointer to the module to be erased.

NFKM_hashpp

This function hashes a pass phrase for use as an Operator Card Set pass phrase:

```
M_Status NFKM_hashpp(
                                NFast_AppHandle app,
                                NFastApp_Connection conn,
                                const char *string,
                                M_Hash *hash_r,
                                struct NFast_Call_Context *cctx
);
```

NFKM_initworld_*

NFKM_initworld_abort

Destroys a security world initialization context.

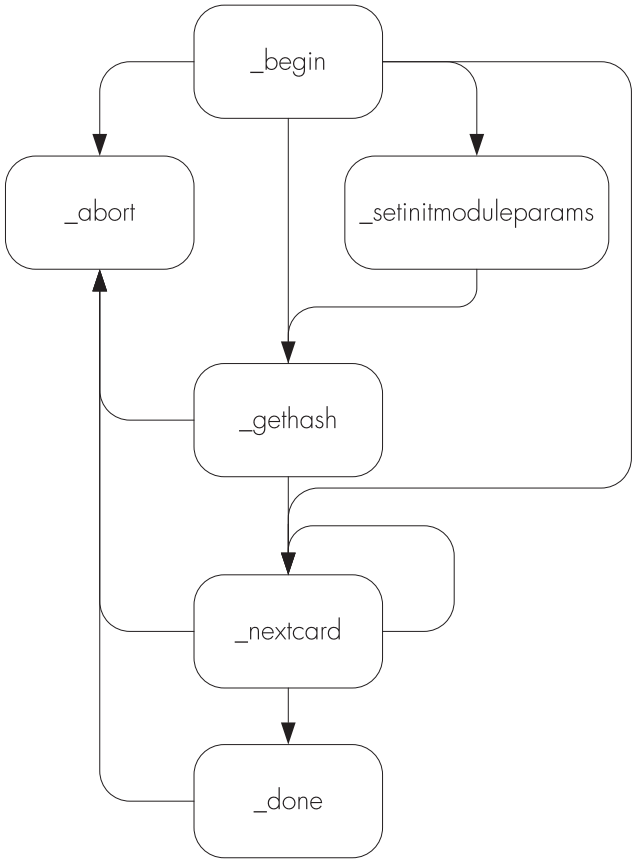
```
void NFKM_initworld_abort(
                                        NFKM_InitWorldHandle iwh
);
```

- **`NFKM_InitWorldHandle iwh`** is the handle for security world initialization returned by **`NFKM_initworld_begin`**.

NFKM_initworld_begin

Does the initial part of work for a security world initialization. Figure 16 illustrates the paths through the NFKM_initworld process.

Figure 16. NFKM_Initworld_*




```
M_Status NFKM_initworld_begin(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        NFKM_InitWorldHandle *iwh,
                                        const NFKM_ModuleInfo *m,
                                        const NFKM_InitWorldParams *iwp,
                                        struct NFast_Call_Context *cc
);
```

- **`NFKM_InitWorldHandle *iwh`** is a pointer to the address of handle to set
- **`const NFKM_ModuleInfo *m`** is a pointer to the module to be initialized
- **`const NFKM_InitWorldParams *iwp`** is a pointer to the parameters for new world

If this function fails, nothing will have been allocated and no further action need be taken; if it succeeds, the handle returned must be freed by calling **`NFKM_initworld_done`** or **`NFKM_initworld_abort`**.

It will help if you call **`NFKM_getinfo`** again after this function — otherwise you won't be able to refer to the module's slots since it was in **`PreInitialisation`** mode last time you looked.

NFKM_initworld_done

Finishes security world initialization.

```
M_Status NFKM_initworld_done(
                                        NFKM_InitWorldHandle iwh
);
```

- **`NFKM_InitWorldHandle iwh`** is the handle for security world initialization returned by **`NFKM_initworld_begin`**.

If this function succeeds, the handle will have been freed; if it fails, you must still call **`NFKM_initworld_abort`**.

NFKM_initworld_gethash

Fetches the identifying hash for new administrator cards created by this job.

```
void NFKM_initworld_gethash(
                                        NFKM_InitWorldHandle iwh,
                                        M_Hash *hh
);
```

- **`NFKM_InitWorldHandle iwh`** is the handle for security world initialization returned by **`NFKM_initworld_begin`**.
- **`M_Hash *hh`** is a pointer to a memory location to which you want the function to write the hash

NFKM_initworld_nextcard

Writes an administrator card.

```
M_Status NFKM_initworld_nextcard(
                                              NFKM_InitWorldHandle iwh,
                                              NFKM_SlotInfo *s,
                                              const M_Hash *pp,
                                              int *left
);
```

- **`NFKM_InitWorldHandle iwh`** is the handle for security world initialization returned by **`NFKM_initworld_begin`**.
- **`NFKM_SlotInfo *s`** is a pointer to the slot containing the admin card
- **`const M_Hash *pp`** is a pointer to the passphrase for the card
- **`int *left`** is the address to store number of cards remaining.

NFKM_initworld_setinitmoduleparams

Configures the parameters for module initialization at the end of the world initialization.

```
M_Status NFKM_initworld_setinitmoduleparams(
                                                    NFKM_InitWorldHandle iwh,
                                                    const NFKM_InitModuleParams *imp
);
```

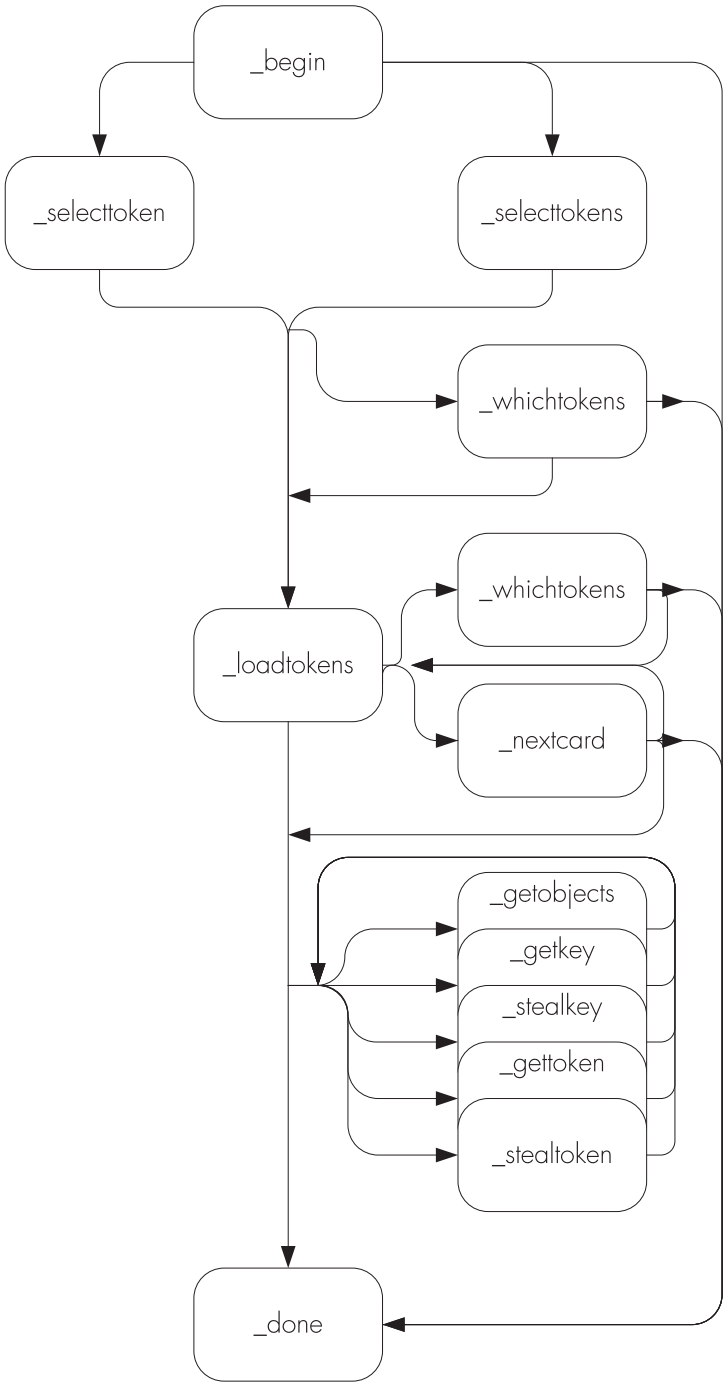
- **`NFKM_InitWorldHandle iwh`** is the handle for security world initialization returned by **`NFKM_initworld_begin`**.
- **`const NFKM_InitModuleParams *imp`** is a pointer to the module initialization params.

NFKM_loadadminkeys_*

NFKM_loadadminkeys_begin

Initializes an operation to load administrator keys. Initially, no tokens are selected for loading. Figure 17 illustrates the paths through the **`NFKM_loadadminkeys`** process.

Figure 17. NFKM_loadadminkeys_*




```
M_Status NFKM_loadadminkeys_begin(
                                                NFast_AppHandle app,
                                                NFastApp_Connection conn,
                                                NFKM_LoadAdminKeysHandle *lakh,
                                                const NFKM_ModuleInfo *m,
                                                struct NFast_Call_Context *cc
);
```

- **`NFKM_LoadAdminKeysHandle *lakh`** is a pointer to the address to which the function writes a handle for this operation.
- **`const NFKM_ModuleInfo *m`** is a pointer to the module on which you wish to load the keys.

NFKM_loadadminkeys_done

Frees a key loading context. Any keys and tokens remaining owned by the context are destroyed.

```
void NFKM_loadadminkeys_done(
                                        NFKM_LoadAdminKeysHandle lakh
);
```

- **`NFKM_LoadAdminKeysHandle lakh`** is the handle returned by **`NFKM_loadadminkeys_begin`**

NFKM_loadadminkeys_{get,steal}{key,token}

These are convenience functions which offer slightly simpler interfaces than **`NFKM_loadadminkeys_getobjects`**.

The **`steal`** functions set the **`NFKM_LAKF_STEAL`** flag, which the **`get`** functions do not; the **`key`** functions load keys whereas the **`token`** functions fetch logical tokens. See **`NFKM_loadadminkeys_getobjects`** for full details about the behavior of these functions.

```
M_Status NFKM_loadadminkeys_getkey(
                                        NFKM_LoadAdminKeysHandle lakh,
                                        int i,
                                        M_KeyID *k
);
```

```
M_Status NFKM_loadadminkeys_stealkey(
                                        NFKM_LoadAdminKeysHandle lakh,
                                        int i,
                                        M_KeyID *k
);
```

```
M_Status NFKM_loadadminkeys_gettoken(
                                        NFKM_LoadAdminKeysHandle lakh,
                                        int i,
                                        M_KeyID *k
);
```

```
M_Status NFKM_loadadminkeys_stealtoken(
                                        NFKM_LoadAdminKeysHandle lakh,
                                        int i,
                                        M_KeyID *k
);
```

- **`NFKM_LoadAdminKeysHandle lakh`** is the handle returned by **`NFKM_loadadminkeys_begin`**
- **`int i`** is the label for the key or token
- **`M_KeyID *k`** is a pointer to the address to store the keyid

Note: A key cannot be loaded once its logical token has been stolen. Therefore, if you want to steal a key and its token, you must steal the key first.

NFKM_loadadminkeys_getobjects

Extracts objects from the **`admin keys`** context.

```
M_Status NFKM_loadadminkeys_getobjects(
                                                NFKM_LoadAdminKeysHandle lakh,
                                                M_KeyID *v,
                                                const int *v_k,
                                                const int *v_lt,
                                                unsigned f
);
```

- **`NFKM_LoadAdminKeysHandle lakh`** is the handle returned by **`NFKM_loadadminkeys_begin`**
- **`M_KeyID *v`** is a pointer to the output vector of keyids
- **`const int *v_k`** is a vector of key labels
- **`const int *v_lt`** is a vector of token labels
- **`unsigned f`** is a bitmap of flags

Extracts objects from the admin keys context. Logical tokens must have been loaded using the **`selecttokens`**, **`loadtokens`** and **`nextcard`** interface; keys must have their protecting logical token loaded already. The **`KeyIDs`** for the objects are stored in the array **`v`** in the order of their labels in the **`v_k`** and **`v_lt`** vectors, keys first. The label vectors are terminated by an entry with the value -1. Either **`v_k`** or **`v_lt`** (or both) may be null to indicate that no objects of that type should be loaded.

Usually, the context retains *ownership* of the objects extracted: the objects will remain available to other callers, and will be Destroyed when the context is freed. If the flag **`NFKM_LAKF_STEAL`** is set in **`f`**, the context will forget about the object; it will not be available to subsequent callers, nor be Destroyed automatically.

Note: Stealing a logical token will prevent keys from being loaded from blobs until that token is reloaded. However, note that keys which have already been loaded but not stolen will remain available.

As an example, consider the case where $\mathbf{LT}_R$ has been loaded. Two calls are made to getobjects: one which fetches $\mathbf{K}_{RE}$, and a second which steals the token $\mathbf{LT}_R$. It is no longer possible to get $\mathbf{K}_{RA}$ (because $\mathbf{LT}_R$ is now unavailable), but further requests to get $\mathbf{K}_{RE}$ will be honoured.

If an error occurs, the contents of the vector **`v`** are unspecified, and no objects will have been stolen. However, some of the requested keys may have been loaded.

NFKM_loadadminkeys_loadtokens

Starts loading the necessary tokens. It might be possible that they're all loaded already, in which case **`*left`** is reset to zero on exit.

```
M_Status NFKM_loadadminkeys_loadtokens(
                                                    NFKM_LoadAdminKeysHandle lakh,
                                                    int *left
);
```

- **`NFKM_LoadAdminKeysHandle lakh`** is the handle returned by **`NFKM_loadadminkeys_begin`**
- **`int *left`** is the address at which to store the number of cards remaining.

NFKM_loadadminkeys_nextcard

Reads an admin card.

```
M_Status NFKM_loadadminkeys_nextcard(
                                                  NFKM_LoadAdminKeysHandle lakh,
                                                  const NFKM_SlotInfo *s,
                                                  const M_Hash *pp, int *left
);
```

- **`NFKM_LoadAdminKeysHandle lakh`** is the handle returned by **`NFKM_loadadminkeys_begin`**
- **`const NFKM_SlotInfo *s`** is a pointer to slot to read
- **`const M_Hash *pp`** is a pointer to pass phrase hash, or **`NULL`** if the card has no pass phrase
- **`int *left`** is the address at which to store the number of cards remaining.

NFKM_loadadminkeys_selecttoken

Selects a single token to be loaded.

```
M_Status NFKM_loadadminkeys_selecttokens(
                                                    NFKM_LoadAdminKeysHandle lakh,
                                                    int k
);
```

- **`NFKM_LoadAdminKeysHandle lakh`** is the handle returned by **`NFKM_loadadminkeys_begin`**
- **`int *k`** is a key or token label. A key label requests that the token protecting that key be loaded.

NFKM_loadadminkeys_selecttokens

Selects a collection of tokens to be loaded.

```
M_Status NFKM_loadadminkeys_selecttokens(
                                                    NFKM_LoadAdminKeysHandle lakh,
                                                    const int *k
);
```

- **`NFKM_LoadAdminKeysHandle lakh`** is the handle returned by **`NFKM_loadadminkeys_begin`**
- **`const int *k`** is an array of key or token labels

The array is terminated by an entry containing the value -1. Each entry may be either a key or token label. A key label requests that the token protecting that key be loaded.

NFKM_loadadminkeys_whichtokens

Discovers which logical tokens will be read in the next or current **`loadtokens`** operation.

```
NFKM_ShareFlag NFKM_loadadminkeys_whichtokens(
                                                NFKM_LoadAdminKeysHandle lakh
);
```

- **`NFKM_LoadAdminKeysHandle lakh`** is the handle returned by **`NFKM_loadadminkeys_begin`**

Returns a bitmap of logical tokens to be loaded.

NFKM_loadcardset_*

NFKM_loadcardset_abort

This function aborts the loading of a card set:

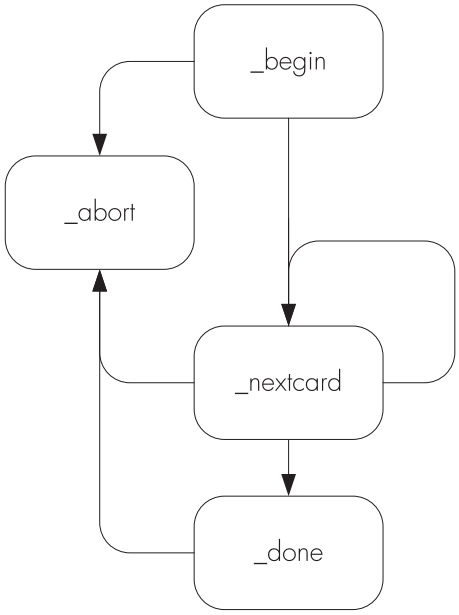
```
void NFKM_loadcardset_abort(
                                      NFKM_LoadCSHandle state
);
```

NFKM_loadcardset_begin

Note: Use the **`NFKM_getinfo`** call to get the current state before using any other call that relies on the data in the **`NFKM_SlotInfo`** structure being up to date.

This function prepares to load a card set. Figure 18 illustrates the paths through the **`NFKM_loadcardset process`**.:

Figure 18. NFKM_loadcardset_*




```
M_Status NFKM_loadcardset_begin(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        const NFKM_ModuleInfo *module,
                                        const NFKM_CardSet *cardset,
                                        NFKM_LoadCSHandle *state_r,
                                        struct NFast_Call_Context *cctx
);
```

NFKM_loadcardset_done

This function completes the loading of a card set:

```
M_Status NFKM_loadcardset_done(
                                        NFKM_LoadCSHandle state,
                                        M_KeyID *logtokid_r
);
```

NFKM_loadcardset_nextcard

Note: Use the **`NFKM_getinfo`** call to get the current state before using any other call that relies on the data in the **`NFKM_SlotInfo`** structure being up to date.

This function attempts to load the next card in a card set:

```
M_Status NFKM_loadcardset_nextcard(
                                                NFKM_LoadCSHandle state,
                                                const NFKM_SlotInfo *slot,
                                                const M_Hash *pp,
                                                int *sharesleft_r,
                                                struct NFast_Call_Context *cctx
);
```

It returns **Status_OK** if the card was loaded successfully. Otherwise, in the event of an error, the return value will be **TokenIOError**, **PhysTokenNotPresent**, **DecryptFailed**, or potentially something else in the event of an unrecoverable error. After any error, even a recoverable one, ***sharesleft_r** is not changed.

NFKM_loadworld_*

NFKM_loadworld_abort

Destroys a security world loading context.

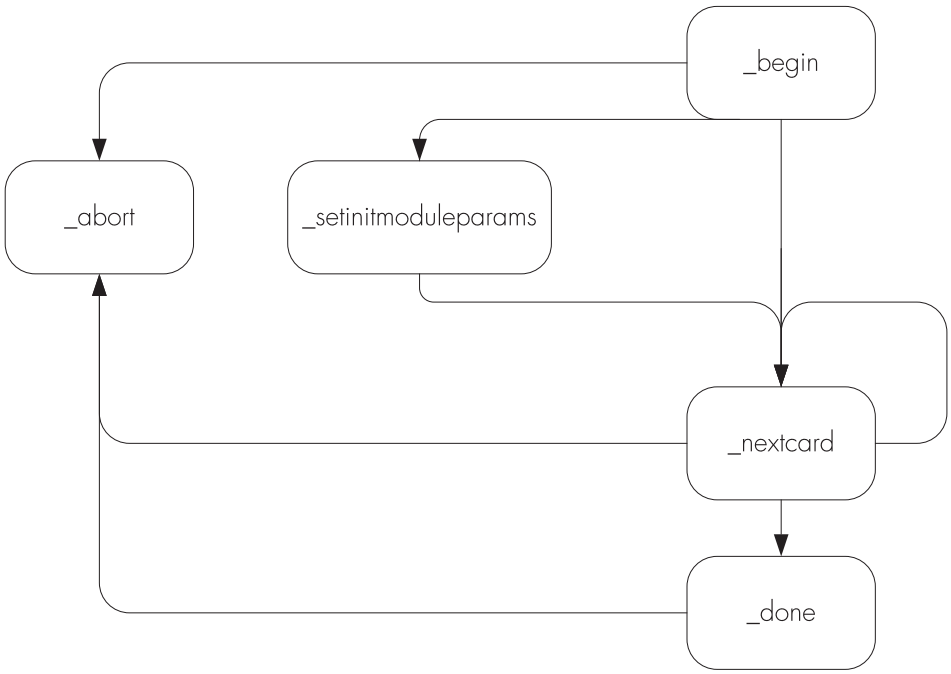
```
void NFKM_loadworld_abort(
                                        NFKM_LoadWorldHandle lwh
);
```

- **NFKM_LoadWorldHandle lwh** is the handle for the security world to be loaded returned by **NFKM_loadworld_begin**.

NFKM_loadworld_begin

Initializes an operation to program a module with an existing security world. Figure 19 illustrates the paths through the **NFKM_loadworld** process.

Figure 19. NFKM_loadworld_*




```
M_Status NFKM_loadworld_begin(
                                    NFast_AppHandle app,
                                    NFastApp_Connection conn,
                                    NFKM_LoadWorldHandle *lwh,
                                    const NFKM_ModuleInfo *m,
                                    struct NFast_Call_Context *cc
);
```

- **`NFKM_LoadWorldHandle *lwh`** is a pointer to the address of handle to fill in
- **`const NFKM_ModuleInfo *m`** is a pointer to the module to be initialized

If this function fails, nothing will have been allocated and no further action need be taken; if it succeeds, the handle returned must be freed by calling **`NFKM_loadworld_done`** or **`NFKM_loadworld_abort`**.

As with initializing new security worlds, it will help if you call **`NFKM_getinfo`** again after this function.

NFKM_loadworld_done

Finishes security world loading.

```
M_Status NFKM_loadworld_done(
                                    NFKM_LoadWorldHandle lwh
);
```

- **`NFKM_LoadWorldHandle lwh`** is the handle for the security world to be loaded returned by **`NFKM_loadworld_begin`**.

If this function succeeds, the handle will have been freed; if it fails, you must still call **NFKM_loadworld_abort**.

NFKM_loadworld_nextcard

Reads an administrator card.

```
M_Status NFKM_loadworld_nextcard(
                                        NFKM_LoadWorldHandle lwh,
                                        const NFKM_SlotInfo *s,
                                        const M_Hash *pp, int *left
);
```

- **NFKM_LoadWorldHandle lwh** is the handle for the security world to be loaded returned by **NFKM_loadworld_begin**.
- **const NFKM_SlotInfo *s** is a pointer to the slot containing the admin card
- **const M_Hash *pp** is a pointer to the passphrase for the card
- **int *left** is a pointer to the address to store number of cards remaining

NFKM_loadworld_setinitmoduleparams

Configures the parameters for module initialization at the end of the world initialization.

```
M_Status NFKM_loadworld_setinitmoduleparams(
                                                NFKM_LoadWorldHandle lwh,
                                                const NFKM_InitModuleParams *imp
);
```

- **NFKM_LoadWorldHandle lwh** is the handle for the security world to be loaded returned by **NFKM_loadworld_begin**
- **const NFKM_InitModuleParams *imp** is a pointer to the module initialization parameters.

NFKM_makecardset_*

NFKM_makecardset_abort

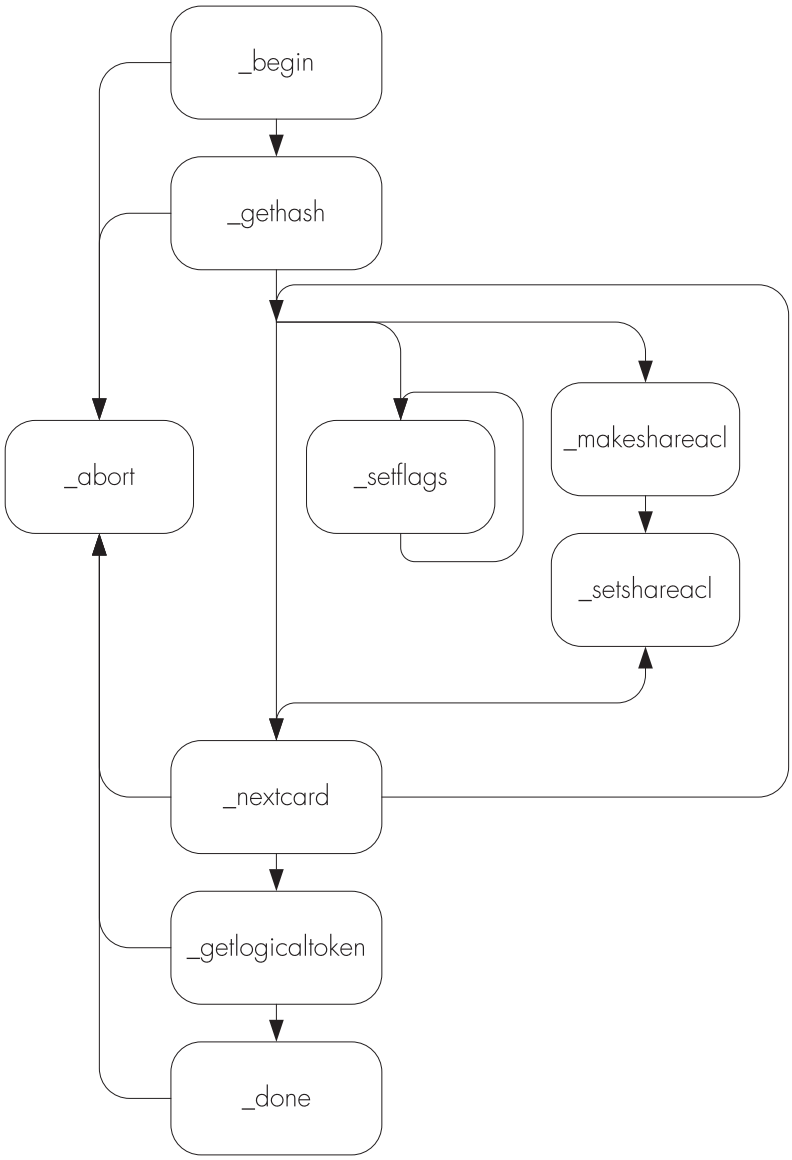
This function aborts the creation of a card set:

```
void NFKM_makecardset_abort(
                                    NFKM_MakeCSHandle state
);
```

NFKM_makecardset_begin

This function prepares to make a new card set. Figure 20 illustrates the paths through the **NFKM_makecardset** process.

Figure 20. NFKM_makecardset_*



Note: `NFKM_makecardset_setflags`, `NFKM_makecardset_makeshareacl` or `NFKM_makecardset_setshareacl` are not recommended for normal use

Note: Use the `NFKM_getinfo` call to get the current state before using any other call that relies on the data in the `NFKM_SlotInfo` structure being up to date.

```
M_Status NFKM_makecardset_begin(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        const NFKM_ModuleInfo *module,
                                        NFKM_MakeCSHandle *state_r,
                                        const char *name,
                                        int n,
                                        int k,
                                        M_Word flags,
                                        int timeout,
                                        NFKM_FIPS140AuthHandle fips140auth,
                                        struct NFast_Call_Context *cctx
);
```

- **`const NFKM_ModuleInfo *module`** is a pointer to the module to use to make the card set
- **`NFKM_MakeCSHandle *state_r`** is a pointer to the card set state.

```
 typedef struct NFKM_MakeCSState
*NFKM_MakeCSHandle;
```

- **`const char *name`** is the name to use for this card set.
- **`int n`** is the total number of cards in the set
- **`int k`** is the the quorum, the number of cards that must be read to recreate the logical token.
- **`M_Word`** flags a flags word, the following flag is defined:

```
NFKM_SAF_REMOTE 1u /*Allow remote reading of shares */
```

- **`int timeout`** is the time out for the card set or 0 for no time out. This is the time in seconds from the laoding fo the card set after which the module will destroy the logical tokens protected by the card set.

`NFKM_FIPS140AuthHandle fips140auth` is only required in FIPS 140-2 level 3 security worlds.

NFKM_makecardset_done

This function completes the creation of a card set:

```
M_Status NFKM_makecardset_done(
                                        NFKM_MakeCSHandle state,
                                        NFKM_CardSetIdent *ident_r,
                                        NFKM_FIPS140AuthHandle fips140auth
);
```

NFKM_makecardset_gethash

The functions fetches the identifying hash for cards created by this **`makecardset`** job.

```
void NFKM_makecardset_gethash(
                                          NFKM_MakeCSHandle mch,
                                          M_Hash *hh
);
```

NFKM_makecardset_getlogicaltoken

Fetches the logical token id for a card set which has been written.

```
M_Status NFKM_makecardset_getlogicaltoken(
                                                NFKM_MakeCSHandle mch,
                                                M_KeyID *ltid,
                                                unsigned f
);
#define NFKM_MCF_STEAL 1u
```

Only call this function *after* **`NFKM_makecardset_nextcard`** says there are no shares left.

If you set **`NFKM_MCF_STEAL`** in **`f`** then you get to keep the logical token id and **`NFKM_makecardset_done`** won't destroy it.

NFKM_makecardset_makeshareacl

Constructs a share ACL.

```
M_Status NFKM_makecardset_makeshareacl(
                                                  NFKM_MakeCSHandle mch,
                                                  M_Word f,
                                                  M_ACL *acl
);
```

Dispose of the ACL using **`NFastApp_FreeACL`** when you've finished.

NFKM_makecardset_nextcard

Note: Use the **`NFKM_getinfo`** call to get the current state before using any other call that relies on the data in the **`NFKM_SlotInfo`** structure being up to date.

This function writes the next card in a new card set:

```
M_Status NFKM_makecardset_nextcard(
                                            NFKM_MakeCSHandle state,
                                            const char *name,
                                            NFKM_SlotInfo *slot,
                                            const M_Hash *pp,
                                            int *sharesleft_r,
                                            NFKM_FIPS140AuthHandle fips140auth
);
```

It returns values and semantics as for **`NFKM_loadcardset_nextcard`**.

The per-card name must be **NULL** for **n=1** card sets, and non-**NULL** for all other card sets.

NFKM_makecardset_setflags

```
M_Word NFKM_makecardset_setflags(
                                        NFKM_MakeCSHandle mch,
                                        M_Word bic,
                                        M_Word xor
);
```

Returns the current flags; then clears the bits in **bic** and toggles the bits in **xor**.

The flags wanted are the **Card_flags_*** ones.

Note: It is best to avoid using this function; instead, pass appropriate **CardSet_flags_** to **NFKM_makecardset_begin** and it will automatically set appropriate share flags.

NFKM_makecardset_setshareacl

Sets the ACL to be set on subsequent shares of this card set.

```
void NFKM_makecardset_setshareacl(
                                        NFKM_MakeCSHandle mch,
                                        M_ACL *acl
);
```

The ACL is not copied: the pointer must remain valid. The initial state is that no ACL is set for shares; to return to this state, pass a null pointer.

Note: It is best to avoid using this function; instead, pass appropriate **CardSet_flags_** to **NFKM_makecardset_begin** and it will construct and use an appropriate ACL.

NFKM_newkey_*

NFKM_newkey_makeacl

This function creates the ACL for a new key:

```
M_Status NFKM_newkey_makeacl(
                                    NFast_AppHandle app,
                                    NFastApp_Connection conn,
                                    const NFKM_WorldInfo *world
                                    const NFKM_CardSet *cardset
                                    M_Word flags,
                                    M_Word opperms_base,
                                    M_Word opperms_maskout,
                                    M_ACL *acl
                                    struct NFast_Call_Context *cctx
);
```

1. **`const NFKM_WorldInfo *world`** must be non-**`NULL`**.
2. **`const NFKM_CardSet *cardset`** must be **`NULL`** for module-only protection, or non-**`NULL`** for Operator Card Set protection.

3. The following **flags** are defined:
   a. **NFKM_NKF_IKWID**
      If this flag is set, **NFKM_makeacl** does not perform its standard checks. This lets you create keys with esoteric ACLs. **IKWID** stands for 'I know what I'm doing'. You should not set this flag unless you are sure this is true.
   b. **NFKM_NKF_NVMemBlob**
      If this flag is set, **NFKM_makeacl** creates an NVRAM key blob, using the standard ACL options.
   c. **NFKM_NKF_NVMemBlobX**
      If this flag is set, **NFKM_makeacl** creates an NVRAM key blob, using the extended options.
   d. **NFKM_NKF_PerAuthUseLimit**
      If this flag is set, **NFKM_makeacl** creates an ACL with a per auth use limit.
   e. **NFKM_NKF_Protection_mask**
   f. **NFKM_NKF_ProtectionCardSet**
   g. **NFKM_NKF_ProtectionModule**
      It is not necessary to set this flag in conjunction with **NFKM_makeacl** or **NFKM_makeblobs**.
   h. **NFKM_NKF_ProtectionNoKey**
      This flag can be used when generating only public keys.
   i. **NFKM_NKF_ProtectionUnknown**
      It is not necessary to set this flag in conjunction with **NFKM_makeacl** or **NFKM_makeblobs**.
   j. **NFKM_NKF_PublicKey**
      If this flag is set, **NFKM_makeacl** creates the ACL for the public half of a key.
   k. **NFKM_NKF_Recovery_mask**
   l. **NFKM_NKF_RecoveryDefault**, **NFKM_NKF_RecoveryRequired**, **NFKM_NKF_RecoveryDisabled**, **NFKM_NKF_RecoveryForbidden**
      If any of these flags are returned by **NFKM_findkey**, it indicates that recovery is enabled.

Result for a new key if the security world has recovery:	enabled	disabled
NFKM_NKF_RecoveryDefault	enabled	disabled
NFKM_NKF_RecoveryRequired	enabled	**InvalidACL**
NFKM_NKF_RecoveryDisabled	disabled	disabled
NFKM_NKF_RecoveryForbidden	**InvalidACL**	disabled

   m. **NFKM_NKF_RecoveryNoKey**
      If this flag is returned by **NFKM_findkey**, it indicates that there is no private key.
   n. **NFKM_NKF_RecoveryUnknown**
      If this flag is returned by **NFKM_findkey**, it indicates that recovery is unknown.
   o. **NFKM_NKF_SEEAppKey**
      If this flag is set, **NFKM_makeacl** creates an ACL with a certifier for a SEE World. It has been superseded by **NFKM_NKF_SEEAppKeyHashAndMech**
   p. **NFKM_NKF_SEEAppKeyHashAndMech**
      If this flag is set, **NFKM_makeacl** creates an ACL with a certifier for a SEE World specifying the key hash and signing mechanism.
   q. **NFKM_NKF_TimeLimit**

If this flag is set, **NFKM_makeacl** creates an ACL with a time limit

r. **NFKM_NKF_HasCertificate**

4. **M_ACL *acl** —the ACL will be overwritten and, therefore, should not contain any pointers to memory that has been operated on by **malloc**.
   Set to have **oppermissions** values like **_Sign**, **_Decrypt**, **_UseAsBlobKey**, **_UseAsCertificate**, or similar. In many cases, you can set **oppermissions** to be one or more of the following macros, depending on the capabilities of the key:
   - **NFKM_DEFOPPERMS_SIGN**
   - **NFKM_DEFOPPERMS_VERIFY**
   - **NFKM_DEFOPPERMS_ENCRYPT**
   - **NFKM_DEFOPPERMS_DECRYPT**

   You can also use some combination of those macros for keys that can do both, such as RSA and symmetric keys:

```
#define NFKM_DEFOPPERMS_SIGN
(Act_OpPermissions_Details_perms_Sign|Act_OpPermissions_Details_perms_UseAsCertificate
|Act_OpPermissions_Details_perms_SignModuleCert)
#define NFKM_DEFOPPERMS_VERIFY (Act_OpPermissions_Details_perms_Verify)
#define NFKM_DEFOPPERMS_ENCRYPT
(Act_OpPermissions_Details_perms_Encrypt|Act_OpPermissions_Details_perms_UseAsBlobKey)
#define NFKM_DEFOPPERMS_DECRYPT
(Act_OpPermissions_Details_perms_Decrypt|Act_OpPermissions_Details_perms_UseAsBlobKey)
```

   If you wish to modify the default ACL, you may do so after calling this function. In such a case, the ACL will be allocated dynamically.
   The **Protection** flags either must be **Unknown** or they must be **NFKM_Module** or **NFKM_CardSet** and correspond to whether **cardset** is non-**NULL**. In any case, **NFKM_CardSet** determines the protection. You must free the ACL at some point, either by using **NFastApp_FreeACL** or, if the ACL was part of a command, as part of a call to **NFastApp_Free_Command**.

NFKM_newkey_makeaclx

This is an alternative to **NFKM_newkey_makeacl** which enables you to define more complex ACLs by defining input in the **NFKM_MakeACLParams structures**.

```
M_Status NFKM_newkey_makeaclx(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        const NFKM_WorldInfo *w,
                                        const NFKM_MakeACLParams *map,
                                        M_ACL *acl,
                                        struct NFast_Call_Context *cc
);
```

```
typedef struct NFKM_MakeACLParams {
M_Word f;
M_Word op_base, op_bic;
const NFKM_CardSet *cs;
const M_Hash *seeinteg;                          SEEAppKey
M_Word timelimit;                                TimeLimit
const M_KeyHashAndMech *seeintegkham;            SEEAppKeyHashAndMech
M_Word pa_uselimit;                              PerAuthUseLimit
NFKM_FIPS140AuthHandle fips;                     NVMemBlob, maybe others later
const M_Hash *hknvacl;                           NVMemBlobX
} NFKM_MakeACLParams;
```

The values for **NFKM_WorldInfo** and **NFKM_CardSet** are the same as for **NFKM_newkeymakeacl**.

Note: If you are creating a key for a SEE application, specify the application signing key using a **M_KeyHashAndMech**. Use of an **M_Hash** is deprecated.

NFKM_newkey_makeblobs

This function creates the working and recovery blobs for a newly generated key:

```
M_Status NFKM_newkey_makeblobs(
                                        NFast_AppHandle app,
                                        const NFKM_WorldInfo *world,
                                        M_KeyID privatekey,
                                        M_KeyID publickey,
                                        const NFKM_CardSet *cardset,
                                        M_KeyID logtokenid,
                                        M_Word flags,
                                        NFKM_Key *newkeydata_io,
                                        struct NFast_Call_Context *cctx
);
```

- **world** must be non-NULL.
- One or both of **privatekey** and **publickey** may be 0 if only one-half, or possibly even neither, is to be recorded. If the key is a symmetric key, supply it as **privatekey**.
- **cardset** and **logtokenid** must be set consistently; either both must be **NULL** or both must be non-**NULL**, depending on whether **cardset** was 0 in **NFKM_makeacl**.
- **flags** should be as in **NFKM_makeacl** for the private half (**_PublicKey** must not be specified).

This call overwrites the previous contents of **newkeydata_io** members **privblob**, **-pubblob** and **privblobrecov**, so these should not contain pointers to any memory that has been operated on by **malloc**. This call also fills in the **hash** member. It does not change the other members, which must be set appropriately before the caller uses **NFKM_recordkey**.

NFKM_newkey_makeblobsx

This function creates the working and recovery blobs for a newly generated key— it offers more functionality than **NFKM_newkey_makeblobs** as you can specify details for the blobs in a parameters structure. In particular it may be used to create a key blob stored in NVRAM.

```
M_Status NFKM_newkey_makeblobsx(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        const NFKM_WorldInfo *w,
                                        const NFKM_MakeBlobsParams *mbp,
                                        NFKM_Key *k,
                                        struct NFast_Call_Context *cc
);
```

```
typedef struct NFKM_MakeBlobsParams {
M_Word f;
M_KeyID kpriv, kpub, lt;
const NFKM_CardSet *cs;
NFKM_FIPS140AuthHandle fips;              NVMemBlob, maybe others later
M_KeyID knv;                              NVMemBlob[X]
M_KeyID knvacl;                           NVMemBlobX
} NFKM_MakeBlobsParams;
```

NFKM_newkey_writecert

Sets up the key generation certificate information for a new key.

```
M_Status NFKM_newkey_writecert(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        const NFKM_ModuleInfo *m,
                                        M_KeyID kpriv,
                                        M_ModuleCert *mc,
                                        NFKM_Key *k,
                                        struct NFast_Call_Context *cctx
);
```

The argument **mc** should be the key generation certificate for a symmetric or private key.

To free the data stored in the Key structure, call **NFKM_freecert**.

NFKM_operatorcard_changepp

Note: This function has been superseded by the **NFKM_changepp** function, see *NKFM_changepp* on page 134.

This function changes the pass phrase on an operator card:

```
M_Status NFKM_operatorcard_changepp(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        const NFKM_SlotInfo *slot,
                                        const M_Hash *oldpp,
                                        const M_Hash *newpp,
                                        struct NFast_Call_Context *cctx
);
```

Either **oldpp** or **newpp** may be **NULL** to indicate the absence of a pass phrase.

NFKM_operatorcard_checkpp

Note: This function has been superseded by the **NFKM_checkpp** function, see *NFKM_checkpp* on page 135.

This function checks the pass phrase on an operator card:

```
M_Status NFKM_operatorcard_checkpp(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        const NFKM_SlotInfo *slot,
                                        const M_Hash *pp,
                                        struct NFast_Call_Context *cctx
);
```

pp may be **NULL** to indicate the absence of a pass phrase.

NFKM_recordkey

This function writes the key blobs to the kmdata area of the host computer's hard disk:

```
M_Status NFKM_recordkey(
                                        NFast_AppHandle app,
                                        NFKM_Key *key,
                                        struct NFast_Call_Context *cctx
);
```

NFKM_recordkey does *not* take over any of the memory in the key.

Whether the key is module protected, smart-card protected, or has some other kind of protection is inferred from the **privblob** details.

The **NFKM_Key** block should be cleared to all-bits-zero before use. If you use any advanced features, set the version field (member **v**) to the correct value before calling **recordkey**.

NFKM_recordkeys

NFKM_recordkeys does the same job as NFKM_recordkey for multiple keys.

```
M_Status NFKM_recordkeys(
                                    NFast_AppHandle app,
                                    NFKM_Key **k,
                                    size_t n,
                                    struct NFast_Call_Context *cc
);
```

Either all the keys are written or none are.

NFKM_replaceacs_*

NFKM_replaceacs_abort

Destroys an admin card replacement context.

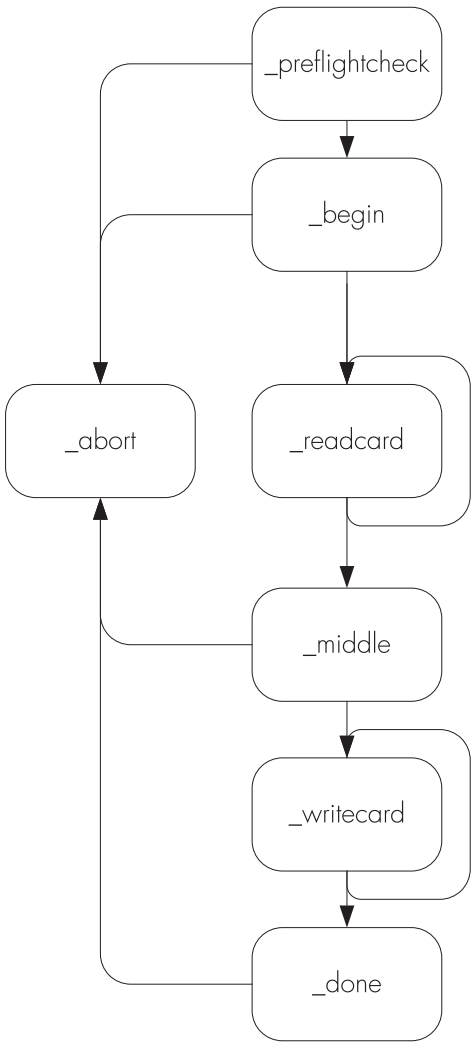
```
void NFKM_replaceacs_abort(
                                    NFKM_ReplaceACSHandle rah
);
```

`NFKM_ReplaceACSHandle rah` is the job handle returned by **`NFKM_replaceacs_begin`**

NFKM_replaceacs_begin

Starts a job to replace the Administrator Card Set. Figure 21 illustrates the paths through the **`NFKM_replaceacs`** process.

Figure 21. NFKM_replaceacs_*




```
M_Status NFKM_replaceacs_begin(
                                        NFast_AppHandle app,
                                        NFastApp_Connection conn,
                                        NFKM_ReplaceACSHandle *rah,
                                        const NFKM_ModuleInfo *m,
                                        struct NFast_Call_Context *cc
);
```

- **`NFKM_ReplaceACSHandle *rah`** is a pointer to the address to which the function will write the job handle
- **`const NFKM_ModuleInfo *m`** is a pointer to the module to use for the transfer

If this function fails, there is nothing else to do; if it succeeds, you must either go all the way through **`NFKM_replaceacs_done`** or call **`NFKM_replaceacs_abort`** to throw away all of the state.

NFKM_replaceacs_done

Wraps up an admin card replacement job.

```
M_Status NFKM_replaceacs_done(
                                        NFKM_ReplaceACSHandle rah
);
```

- **`NFKM_ReplaceACSHandle rah`** is the job handle returned by **`NFKM_replaceacs_begin`**

NFKM_replaceacs_gethash

Fetches the identifying hash for new administrator cards created by this job.

```
void NFKM_replaceacs_gethash(
                                       NFKM_ReplaceACSHandle rah,
                                       M_Hash *hh
);
```

- **`NFKM_ReplaceACSHandle rah`** is the job handle returned by **`NFKM_replaceacs_begin`**
- **`M_Hash *hh`** is a pointer to the address to write the hash

NFKM_replaceacs_middle

Does the work in the middle of an admin card set replacement job.

```
M_Status NFKM_replaceacs_middle(
                                          NFKM_ReplaceACSHandle rah
);
```

`NFKM_ReplaceACSHandle rah` is the job handle returned by **`NFKM_replaceacs_begin`**

NFKM_replaceacs_preflightcheck

Verifies that a **`replaceacs`** operation is safe.

```
int NFKM_replaceacs_preflightcheck(
                                            NFast_AppHandle app,
                                            const NFKM_WorldInfo *w,
                                            int *unsafe,
                                            struct NFast_Call_Context *cc
);
```

- **`NFKM_ReplaceACSHandle rah`** is the job handle returned by **`NFKM_replaceacs_begin`**
- **`const NFKM_WorldInfo *w`** is a pointer to the world information
- **`int *unsafe`** is cleared if safe, nonzero if not

If the operation is safe, **`*unsafe`** is cleared; otherwise it will contain a nonzero value. Later, this might explain in more detail what the problem is. Currently, the only check is for world file entries which aren't understood (and therefore might be blobs of keys which would need to be replaced).

NFKM_replaceacs_readcard

Reads an administrator card, with a view to replacing it.

```
M_Status NFKM_replaceacs_readcard(
                                                NFKM_ReplaceACSHandle rah,
                                                const NFKM_SlotInfo *s,
                                                const M_Hash *pp,
                                                int *left
);
```

- **`NFKM_ReplaceACSHandle rah`** is the job handle returned by **`NFKM_replaceacs_begin`**
- **`const NFKM_SlotInfo *s`** is a pointer to the slot containing the admin card
- **`const M_Hash *pp`** is a pointer to the passphrase hash for the card
- **`int *left`** is a pointer to the address to store number of cards remaining

NFKM_replaceacs_writecard

Writes a replacement administrator card.

```
M_Status NFKM_replaceacs_writecard(
                                               NFKM_ReplaceACSHandle rah,
                                               NFKM_SlotInfo *s, const M_Hash *pp,
                                               int *left
);
```

- **`NFKM_ReplaceACSHandle rah`** is the job handle returned by **`NFKM_replaceacs_begin`**
- **`NFKM_SlotInfo *s`** is a pointer to the slot containing admin card
- **`const M_Hash *pp`** is a pointer to the passphrase hash for the card
- **`int *left`** is a pointer to the address to store number of cards remaining

Appendix D: nCore API commands

This chapter describes the complete nShield command set. It is divided into the following sections:

- **Basic commands**
  These commands are available on all nShield modules. They do not offer any key-management functionality
- **Key-management commands**
  These commands are only available on nForce and nShield modules.
- **Commands used only by the generic stub**
  These commands are included for information only. You should not need to call them directly.

Commands are listed alphabetically within each section. For each command, the following information is listed:

- the command name
- the states in which the command can be issued
- the required inputs
- the expected output

Note: If the module is unable to complete a requested command due to a non-fatal condition, such as lack of memory or an unknown command, the module sends a response with no reply data. The reply's `cmd` value is sent to `Cmd_ErrorReturn` with the condition indicated by the status word that was returned in the header.

Note: Unless specified otherwise, there is a limit of 8K on the total message that can be sent to the nShield server for each command, or in reply. This means that the maximum length of any byteblock sent for processing must be somewhat less that 8K.

Basic commands

The following basic commands, described in this section, are available on all nShield modules:

- `ClearUnit`
- `ClearUnitEx`
- `ModExp`
- `ModExpCrt`

These commands perform cryptographic acceleration without key management.

These commands are intended for use by applications that manage their own keys.

ClearUnit

All non-error states	"Privileged" users only

This command resets a module, returning it to the same mode that it was previously in. The module and server negotiate to enable the module to be reset without disturbing the host's PCI subsystem.

When the module is cleared:

- all object handles (ID_{KA}, ID_{KT}, etc.) are invalidated
- any share reassembly process that is currently active is aborted
- the module enters the self-test state.

ClearUnit does not destroy:

- module keys K_M
- module signing key K_{ML}
- long-term fixed signing key K_{LF}
- nShield Security Officer's key K_{NSO}.

Arguments

```
struct M_Cmd_ClearUnit_Args {
    M_ModuleID module;                                  ModuleID
};
```

Reply

The reply structure for this command is empty.

The status is **Status_OK** or, if the unit is already being reset, **Status_UnitReset**. The reply is sent immediately (that is, before the unit is actually cleared).

Notes

In versions of the server prior to 1.40, the **ClearUnit** command caused a hard reset. In release 1.40, the **ClearUnit** command was given a new command number, and the old command number was renamed **OldClearUnit**, which is included for backward compatibility only. From release 1.40, servers interpret **ClearUnit** and **OldClearUnit** as **ClearUnit**. The **ClearUnit** command fails with **Status_UnknownCommand** on servers older than release 1.40.

ClearUnitEx

All non-error states	"Privileged" users only

This command resets a module, and optionally enables you to change the mode as required. **ClearUnitEx** is implemented entirely by the hardserver, which:

- Checks and sets the scratchpad registers
- Sets a **want clear** state on the command target

Further behavior is identical to the **ClearUnit** command, including sending **ClearUnit** (*not* **ClearUnixEx**) to the module. See *ClearUnit* on page 165 for more about the **ClearUnit** command.

Arguments

```
bitmap:                         flags
       harmless: 16-
ModuleID                        module        [module to be reset]
ModuleMode                      mode          [desired module mode]
```

- Flags are not currently used.

Module mode settings

The following desired module mode settings are available:

```
ModuleMode Default =0
ModuleMode Maintenance =1
ModuleMode Operational =2
ModuleMode Initialisation =3
```

Reply

- The `ModuleMode` value in the reply corresponds to the bit field in scratchpad 0.
- If the module is already part way through a reset, then `@ref Status_UnitReset` is returned.
- If the request cannot be completed because the main application of the module does not support software mode changes, then `@ref Status_ModuleApplicationNotSupported` is returned.
- If the request cannot be completed because the module monitor does not support software mode changes, then `@ref Status ModuleMonitorNotSupported` is returned.

Note: Firmware releases prior to v12 do not support changing the mode without use of the MOI switch. The **mode** argument must be 0. With the appropriate firmware, the **mode** argument can be used to change the mode.

ModExp

Operational state	initialization state

This command performs modular exponentiation on parameters passed by the client.

Arguments

```
struct M_Cmd_ModExp_Args {
   M_Bignum a;                          A base
   M_Bignum p;                          P power
   M_Bignum n;                          N modulus
};
```

Reply

```
struct M_Cmd_ModExp_Reply {
M_Bignum r;
};
```

where **r = A^P mod N**

ModExpCrt

Operational state	initialization state

This command performs modular exponentiation on parameters passed by the client. **ModExpCrt** uses the Chinese Remainder Theorem to reduce the time it takes to perform the operation.

Arguments

```
struct M_Cmd_ModExpCrt_Args {
    M_Bignum a;                     A base
    M_Bignum p;                     P modulus larger factor
    M_Bignum q;                     Q modulus smaller factor
    M_Bignum dmp1;                  D mod (P-1)
    M_Bignum dmq1;                  D mod (Q-1)
    M_Bignum iqmp;                  Q^-1 mod P
};
```

Reply

Uses **M_Cmd_ModExp_Reply**.

Notes

It is assumed that **P >= Q**.

Key-management commands

The commands described in this section, are only available on key-management modules.

If you send any of these commands to an acceleration-only module, it fails with the status value **Status_InvalidState**.

ChangeSharePIN

Operational state	initialization state

This command enables a PIN that protects a single share to be changed. The old PIN must be provided unless the share has no PIN. Likewise, the new PIN must be provided unless the PIN is being removed.

The module decrypts the share using the old PIN and the K_M associated with the token. If the share is decrypted correctly, the module encrypts it using the new PIN and the K_M. It then writes the newly encrypted share to the smart card or software token.

This operation can be performed regardless of whether or not the logical token associated with this share is "present". The only requirement is that both the smart card with the share and the K_M associated with the token be present within the module.

Arguments

```
struct M_Cmd_ChangeSharePIN_Args {
   M_Cmd_ChangeSharePIN_Args_flags flags;
   M_PhysToken token;
   M_KMHash hkm;
   M_ShortHash hkt;
   M_Word i;
   M_PIN *oldpin;
   M_PIN *newpin;
};
```

- The following **`flags`** are defined:
  - **`Cmd_ChangeSharePIN_Args_flags_oldpin_present`**
    Set this flag if the input contains the old PIN. The old PIN must be specified unless the share was previously encrypted without a PIN or if the share uses the protected PIN path.
  - **`Cmd_ChangeSharePIN_Args_flags_newpin_present`**
    Set this flag if the input contains the new PIN. The new PIN must be specified unless the share is to be encrypted without a PIN or if the share uses the protected PIN path.
  - **`Cmd_ChangeSharePIN_Args_flags__allflags`**
- **`M_KMHash hkm`** is Module key hash H_{KM}
- **`M_ShortHash hkt`** is a short, 10-byte token hash, such as returned by GetSlotInfo.
- **`M_Word i`** is the share number
- **`M_PIN *oldpin`** is the old PIN, or **`NULL`**
- **`M_PIN *newpin`** is the new PIN, or **`NULL`**

Reply

The reply structure for this command is empty.

ChannelOpen

Operational state, initialization state	Requires a ClientID

This command opens a communication channel that can be used for bulk encryption. Data can then be transferred over this channel by using the **`ChannelUpdate`** command.

Note: Channel operations are only available for symmetric algorithms.

Arguments

```
typedef struct {
   M_ModuleID module;
   M_ChannelType type;
   M_Cmd_ChannelOpen_Args_flags flags;
   M_ChannelMode mode;
   M_Mech mech;
   M_KeyID *key;
   M_IV *given_iv;
} M_Cmd_ChannelOpen_Args;
```

- **M_ChannelType type** is the data transfer mechanism for the channel. At present, only **ChannelType_Simple** is supported. Alternatively, **ChannelType_Any** can be used to let the module pick the"best" channel type that it supports.

  ```
  ChannelType_Any
  ChannelType_Simple
  ```

- **M_Cmd_ChannelOpen_Args_flags flags** The following flags are defined:
  - **Cmd_ChannelOpen_Args_flags_key_present**
    Set this flag if the command contains a **KeyID**. The command must include a **KeyID** unless you are using a hashing mechanism.
  - **Cmd_ChannelOpen_Args_flags_given_iv_present**
    Set this flag if the command designates which initialization vector to use. For encryption and signature mechanisms, if this flag is not set and the mechanism requires an initialization vector, the module will create a random **iv** and return it in the reply. For decryption and verification mechanisms, this flag must be set and the **M_IV** must be specified or **Status_InvalidParameter** will be returned.
- **M_ChannelMode mode** determines the operation to perform on this channel. The following modes are defined:
  - **ChannelMode_Encrypt**
  - **ChannelMode_Decrypt**
  - **ChannelMode_Sign**
  - **ChannelMode_Verify**
- **M_Mech mech** is the mechanism to use. See *Mechanisms* on page 64 for information on supported mechanisms.
- **M_KeyID *key** is the **KeyID** of the key to use on the channel. The key must have the appropriate **Encrypt**, **Decrypt**, **Sign**, or **Verify** permissions in its ACL. It must also be an appropriate type for the given mechanism. In order to use unkeyed hash mechanisms, this key field must be absent.
- **M_IV *given_iv** is the initialization vector to use on the channel. This field is optional for the **Encrypt** and **Sign** modes, but it must be given for the **Decrypt** and **Verify** modes. **Status_InvalidParameter** is returned if this field is not present when it is required or if it has an incorrect mechanism.

Reply

```
typedef struct {
M_Cmd_ChannelOpen_Reply_flags flags;
M_KeyID idch;
M_IV *new_iv;
M_ChannelOpenInfo openinfo;
} M_Cmd_ChannelOpen_Reply;
```

- **`M_Cmd_ChannelOpen_Reply_flags flags`**
  The following flag is defined: **`Cmd_ChannelOpen_Reply_flags_new_iv_present`**. This flag is set if the **`new_iv`** field is present.
- **`M_KeyID idch`** is the ID of the Channel. It is like a **`KeyID`**; it may be used to refer to the channel and can be destroyed with the **`Destroy`** command after use. However, it will be different to the **`KeyID`**.
  Note: The server will destroy the channel automatically when the last connection associated with the application that created it closes.
- **`M_IV *new_iv`** is an initialization vector for the channel. It is returned only if the channel mode is **`Encrypt`** or **`Sign`** and no **`given_iv`** has been sent with the command.
- **`M_ChannelOpenInfo openinfo`** is extra information about the channel:

```
struct M_ModuleChannelOpenInfo {
M_ChannelType type;
union M_ChannelType__ExtraMCOI info;
};
```

- **`M_ChannelType type`** is the channel type used.
- **`union M_ChannelType__ExtraMCOI info`** is extra information that is dependent on the channel type. It allows the client to access a device driver, if necessary, in order to perform data transfer.

ChannelUpdate

Operational state, initialization state	Requires a ClientID

This command transfers data over a communication channel for bulk encryption. Such a channel must be opened with the **`ChannelOpen`** command before the **`ChannelUpdate`** command can be used.

Note: Channel operations are only available for symmetric algorithms.

Data is streamed into an open channel by giving one or more **`Update`** commands. The last data block to be processed should have the final flag set. This final block does not have to contain any input data (except in **`Verify`** mode; see below). Input data does not have to be multiples of the block size for block ciphers; the module will buffer the data internally as necessary. In general, the output block will contain all the data that can be encrypted/decrypted unambiguously given the input so far. However, PKCS #5 padding usually lags behind by a block when decrypting.

For decryption—and for encryption in non-padding modes—you must have supplied a whole number of input blocks. Otherwise, a status of **Status_EncryptFailed** or **Status_DecryptFailed** will be returned. **Status_DecryptFailed** is also used if unpadding fails during decryption.

For signing modes, no output will be generated until the final bit is set, in which case the signature or hash will be output as the byte block.

For verification modes, no output is generated. Instead, the plain text message must be input by **ChannelUpdate** commands with their final bit clear, then a **ChannelUpdate** with the final bit set is given, with the signature/hash bytes given as the input block. This will return a status of **OK** or **VerifyFailed**, as appropriate.

Arguments

```
struct M_Cmd_ChannelUpdate_Args {
   M_Cmd_ChannelUpdate_Args_flags flags;
   M_KeyID idch;
   M_ByteBlock input;
};
```

- The following **flag** is defined: **Cmd_ChannelUpdate_Args_flags_final**. This flag indicates the last block of input data.
- **M_KeyID idch** is the **ChannelID** returned by **ChannelOpen**.
- **M_ByteBlock input** is a byte block of input data (it may be of zero length)

Reply

```
struct M_Cmd_ChannelUpdate_Reply {
    M_ByteBlock output;
};
```

M_ByteBlock output is a byte block containing output data from the channel. This block may be of zero length.

Decrypt

Operational state, initialization state	Requires a ClientID

This command takes a cipher text and decrypts it with a previously stored key.

The limit of 8K does not apply to data decrypted by this command. This is because the Generic Stub library splits the command into a **ChannelOpen** command followed by a number of **ChannelUpdate** commands. Only symmetric mechanisms use channels; asymmetric mechanisms cannot.

For information on formats, see *Encrypt* on page 176.

Arguments

```
struct M_Cmd_Decrypt_Args {
   M_Cmd_Decrypt_Args_flags flags;
   M_KeyID key;
   M_Mech mech;
   M_CipherText cipher;
   M_PlainTextType reply_type;
};
```

- No **flags** are defined.
- **M_KeyID key** is $\mathbf{ID}_{KA}$.
- **M_Mech mech**: See *Mechanisms* on page 64 for information on supported mechanisms. If **mech** is not **Mech_Any**, then it must match the mechanism of the ciphertext, **cipher.mech**. If it does not match, then a **MechanismNotExpected** error is returned.

Reply

```
struct M_Cmd_Decrypt_Reply {
   M_PlainText plain;
 }
```

DeriveKey

Operational state, initialization state	Requires a ClientID

This command creates a new key object from a number of other keys that have been stored already on the module. Then, **DeriveKey** returns a **KeyID** for the new key.

There are two special key types used by **DeriveKey**:

- a **template key** — the template is used to provide the ACL and application data for the output key
- a **wrapped key** — a key type for holding encrypted keys.

Arguments

```
struct M_Cmd_DeriveKey_Args {
   M_Cmd_DeriveKey_Args_flags flags;
   M_DeriveMech mech;
   int n_keys;
   M_vec_KeyID keys;
   union M_DeriveMech__DKParams params;
} M_Cmd_DeriveKey_Args;
```

- No **flags** are defined.
- **M_DeriveMech mech**
  See *Derive Key Mechanisms* on page 97 for information on supported mechanisms.

- **`int n_keys`**
  This value is the number of keys that have been supplied in the key table. For all currently supported mechanisms, this value must be 3.
- **`M_vec_KeyID keys`**
  This is a table containing the **`KeyIDs`** of the keys that are to be used. You must enter the **`KeyIDs`** of these keys in the following order:
  a. template key
  b. base key
  c. wrapping key(s)

  Note: For all currently supported mechanisms, there is at most 1 wrapping key.
  Each key must be of the correct type for the mechanism.
  Each of these keys must have an ACL that permits them to be used for **`DeriveKey`** operations in this role.

  Note: Any of the keys may have an ACL that requires a certificate. If more than one of the keys requires a certificate, then all the certificates must have the same signing key.
- **`union M_DeriveMech__DKParams params`**
  Parameters for the specific wrapping mechanism. See *Derive Key Mechanisms* on page 97.

```
union M_DeriveMech__DKParams {
 M_DeriveMech_ConcatenationKDF_DKParams concatenationkdf;
M_DeriveMech_PKCS8Encrypt_DKParams pkcs8encrypt;
M_DeriveMech_PKCS8Decrypt_DKParams pkcs8decrypt;
M_DeriveMech_RawDecrypt_DKParams rawdecrypt;
M_DeriveMech_AESKeyWrap_DKParams aeskeywrap;
M_DeriveMech_RSAComponents_DKParams rsacomponents;
M_DeriveMech_AESKeyUnwrap_DKParams aeskeyunwrap;
M_DeriveMech_RawDecryptZeroPad_DKParams rawdecryptzeropad;
M_DeriveMech_SignedKDPKeyWrapDES3_DKParams signedkdpkeywrapdes3;
M_DeriveMech_ECCMQV_DKParams eccmqv;
M_DeriveMech_KDPKeyWrapDES3_DKParams kdpkeywrapdes3;
M_DeriveMech_SSL3withRSA_DKParams ssl3withrsa;
M_DeriveMech_ConcatenateBytes_DKParams concatenatebytes;
M_DeriveMech_RawEncrypt_DKParams rawencrypt;
M_DeriveMech_SSL3withDH_DKParams ssl3withdh;
M_DeriveMech_NISTKDFmCTRpRijndaelCMACr32_DKParams nistkdfmctrprijndaelcmacr32;
M_DeriveMech_RawEncryptZeroPad_DKParams rawencryptzeropad;
};
```

Reply

```
struct M_Cmd_DeriveKey_Reply {
   M_KeyID key;
};
```

The **`M_KeyID`** points to the derived key. The ACL and application data for this key are the ACL and application data that have been stored as the key data of the template key. The key type is defined by the mechanism used. The key data is determined by the base key, the wrapping key (or wrapping keys), and the mechanism.

Notes

The key derivation mechanisms provide a means of converting keys of many different types into **KeyType_Wrapped** and then back again. The type of the original key is usually *not* preserved in the **Wrapped** data format (the **EncryptMarshalled** mechanism *does* preserve type).

Therefore, one key may be converted to another of a different type by unwrapping it with a different mechanism. Indeed, the key data itself may be modified by unwrapping it with a different key.

This feature is provided to increase flexibility and interoperability, which is a major goal of the **DeriveKey** command. However, it can be a potential weak point in security. Therefore, nShield recommends that whenever a base key is turned into a **Wrapped** key type, if the new key is to be used within the nShield environment, the ACL for the new key be set only to allow decoding back to the original key. This is done by setting the **DeriveKey** ACL entry in the wrapped key so that:

- the **mech** field identifies the correct decoding mechanism
- the **otherkeys** table identifies the correct unwrapping key in the right role.

Destroy

Operational state, initialization state	Requires a ClientID

This removes a key object from memory and zeroes any storage associated with it.

This command can be used to destroy:

- a key object by specifying an $\mathbf{ID}_{KA}$
- a logical token by specifying an $\mathbf{ID}_{KT}$
- a **ModuleSEEWorld** by specifying a **KeyID**
- an impath by specifying an **ImpathID**
- an **FTSessionID** or **FileTransferID**
- a channel
- a foreign token lock
- multiple objects that were previously merged by means of **MergeKeyIDs**. Only the merged **KeyID** is removed; the underlying keys remain loaded.

When an object has multiple **KeyIDs**, **Destroy** only removes the **KeyID** for the current **ClientID** or **SEEWorld**. The underlying object is removed when the last **KeyID** for the object is destroyed.

It is an error to **Destroy** an $\mathbf{ID}_{KA}$ that has not been issued previously by the nShield server or that has already been destroyed.

Note: An $\mathbf{ID}_{KA}$ may be reused for a new object after the current object is destroyed.

A key that forms part of a merged set made with **MergeKeyIDs** (see *MergeKeyIDs* on page 197) cannot be destroyed. Attempts to do so will return an **ObjectInUse** error. Destroy the merged **KeyID** first.

Arguments

```
struct M_Cmd_Destroy_Args {
M_KeyID key;
};
```

M_KeyID key can be any object with an **M_KeyID**, such as an $\mathbf{ID}_{KA}$, an $\mathbf{ID}_{KT}$, or the SEE World's *KeyID*.

Reply

The reply structure for this command is empty.

Duplicate

Operational state, initialization state	Requires a ClientID

This command duplicates a key object within module memory and returns a new handle to it. The new key object can then be manipulated independently of the original key object.

The new key inherits its ACL from the original key.

Arguments

```
struct M_Cmd_Duplicate_Args {
   M_KeyID key;
};
```

M_KeyID key is $\mathbf{ID}_{KA}$.

Reply

```
struct M_Cmd_Duplicate_Reply {
   M_KeyID newkey;
};
```

M_KeyID newkey is $\mathbf{ID}_{KA2}$.

Encrypt

Operational state, initialization state	Requires a ClientID

This command encrypts data by using a previously loaded key. It returns the cipher text.

The limit of 8K does not apply to data encrypted by this command. This is because the Generic Stub library splits the command into a **ChannelOpen** command followed by a number of **ChannelUpdate** commands. Only symmetric mechanisms use channels; asymmetric mechanisms cannot.

Arguments

```
struct M_Cmd_Encrypt_Args {
   M_Cmd_Encrypt_Args_flags flags;
   M_KeyID key;
   M_Mech mech;
   M_PlainText plain;
   M_IV *given_iv;
 };
```

- The following **flag** is defined:
  Cmd_Encrypt_Args_flags_given_iv_present
  This flag must be set if the command includes the initialization vector. If this flag is not set, the module will generate a random initialization vector if one is required by this mechanism.
- **M_KeyID key** is $\mathbf{ID}_{KA}$.
- **M_Mech mech**
  See *Mechanisms* on page 64 for information on supported mechanisms. If **Mech_Any** is specified and an IV is given, the mechanism is taken from that IV. Otherwise, if **Mech_Any** is *not* specified, the given mechanism is used. Moreover, if an IV is given, its mechanism must match the given mechanism, otherwise **Status_MechanismNotExpected** will be returned.
- **M_IV *given_iv**
  This can be either the IV to use or otherwise **NULL** if no IV is defined or if you prefer that the module choose an IV on its own.

Reply

```
struct M_Cmd_Encrypt_Reply {
   M_CipherText cipher;
};
```

Export

Operational state, initialization state	Requires a ClientID

This command is used to extract key material in plain text.

Note: Most private key objects should have an ACL (or ACLs) that forbid the reading of this data in plain text.

Arguments

```
struct M_Cmd_Export_Args {
   M_KeyID key;
};
```

Reply

```
struct M_Cmd_Export_Reply {
   M_KeyData data;
};
```

FirmwareAuthenticate

Operational state, initialization state, maintenance state

This command is used to authenticate the firmware in a module by comparing it to a firmware image on the host. If performed in the maintenance state it can be used to authenticate the monitor.

Use the **fwcheck** command-line utility to perform this operation.

FormatToken

Operational state, initialization state	May require a KNSO certificate

This command initializes a smart card.

Arguments

```
struct M_Cmd_FormatToken_Args {
M_Cmd_FormatToken_Args_flags flags;
M_PhysToken token;
M_KMHash *auth_key;
};
```

- The following **flag** is defined:
  Cmd_FormatToken_Args_flags_auth_key_present
  Set this flag if the input includes a module key hash to use for challenge-response authentication. This flag can only be used if the smart card supports authentication.
- **M_KMHash *auth_key** is the H_{KM} of a module key or a **NULL** pointer. The module key is combined with the unique identity of the token to produce the key to be used for challenge-response authentication.

Reply

The reply structure for this command is empty.

GenerateKey and GenerateKeyPair

Operational state, initialization state	Requires a ClientID May require a KNSO certificate

The **GenerateKey** command randomly generates a key object of the given type and with the specified ACL (or ACLs) and stores it in internal RAM.

The **GenerateKeyPair** command randomly generates a matching public and private key pair.

Use **GenerateKey** for symmetric algorithms.

For public-key algorithms, use **GenerateKeyPair**.

Arguments

```
struct M_Cmd_GenerateKey_Args {
   M_Cmd_GenerateKey_Args_flags flags;
   M_ModuleID module;
   M_KeyGenParams params;
   M_ACL acl;
   M_AppData *appdata;
};
```

```
struct M_Cmd_GenerateKeyPair_Args {
   M_Cmd_GenerateKeyPair_Args_flags flags;
   M_ModuleID module;
   M_KeyGenParams params;
   M_ACL aclpriv;
   M_ACL aclpub;
   M_AppData *appdatapriv;
   M_AppData *appdatapub;
} M_Cmd_GenerateKeyPair_Args;
```

- The following **flags** are defined:
  - **Cmd_GenerateKey_Args_flags_Certify**
    If this flag is set, the reply will contain a certificate of data type **ModuleCert** that describes the security policy for this key or key pair. This certificate enables an observer, such as an organization's Security Officer or a certificate authority, to check that the key or key pair was generated in compliance with a stated security policy before they allow the key to be used. The certificate contains:
    - H_{KA} for the key
    - the application data field or fields
    - the ACL (or ACLs)
    - The certificate is signed by the module's private key. K_{ML}^{-1}
  - **Cmd_GenerateKey_Args_flags_appdata_present**
    You must set this flag if the request contains application data for the symmetric key.
  - **Cmd_GenerateKey_Args_flags_PairwiseCheck**
    If this flag is set, the module performs a consistency check on the key by creating a random message, then encrypting and decrypting this message. The test fails if the encrypted message is the same as the plain text or if the encrypted message fails to decrypt to the plain text.
  - **Cmd_GenerateKeyPair_Args_flags_Certify**
  - **Cmd_GenerateKeyPair_Args_flags_appdatapriv_present**
    You must set this flag if the request contains application data for the private key.
  - **Cmd_GenerateKeyPair_Args_flags_appdatapub_present**
    You must set this flag if the request contains application data for the public key.
  - **Cmd_GenerateKeyPair_Args_flags_PairwiseCheck**
- **M_ModuleID module**
  If the module ID is nonzero, the key is loaded onto the specified module. If the module ID is 0, the key is loaded onto the first available module. You can use the **GetWhichModule** command to determine which modules contain which keys).
- **M_KeyGenParams params**
  The key type and required parameters needed to generate this key or key pair are as follows:

```
struct M_KeyGenParams {
   M_KeyType type;
   union M_KeyType__GenParams params;
};
```

- The following **key types** are defined:
  - **KeyType_ArcFour** Use **GenerateKey**
  - **KeyType_Blowfish**
  - **KeyType_CAST** Use **GenerateKey**
  - **KeyType_CAST256**
  - **KeyType_DES** Use **GenerateKey**
  - **KeyType_DES2** Use **GenerateKey**
  - **KeyType_DES3** Use **GenerateKey**
  - **KeyType_DHPrivate** Use **GenerateKeyPair**
  - **KeyType_DHPublic** Do not use for key generation
  - **KeyType_DKTemplate**
  - **KeyType_DSAComm** Use **GenerateKey**
  - **KeyType_DSAPrivate** Use **GenerateKeyPair**
  - **KeyType_DSAPublic** Do not use for key generation
  - **KeyType_HMACMD2**
  - **KeyType_HMACMD5**
  - **KeyType_HMACRIPEMD160**
  - **KeyType_HMACSHA1**
  - **KeyType_HMACSHA256**
  - **KeyType_HMACSHA384**
  - **KeyType_HMACSHA512**
  - **KeyType_HMACTiger**
  - **KeyType_IDEA**
  - **KeyType_KCDSAComm**
  - **KeyType_KCDSAPrivate**
  - **KeyType_KCDSAPublic**
  - **KeyType_Random** Use **GenerateKey**
  - **KeyType_RC2**
  - **KeyType_RC5**
  - **KeyType_Rijndael**
  - **KeyType_RSAPrivate** Use **GenerateKeyPair**
  - **KeyType_RSAPublic** Do not use for key generation
  - **KeyType_SEED**
  - **KeyType_Serpent**
  - **KeyType_Skipjack**
  - **KeyType_SSLMasterSecret KeyType_Template_Data** Do not use for key generation
  - **KeyType_Twofish**
  - **KeyType_Void KeyType_Wrapped** Created by **DeriveKey**
  - **KeyType_Any** Do not use for key generation
  - **KeyType_None** Do not use for key generation

  Note: When generating a key pair, you must specify the key type for the private half of the key pair.

  Note: The following key types have **key generation parameters**:

```
union M_KeyType__GenParams {
   M_KeyType_RSAPrivate_GenParams rsaprivate;
   M_KeyType_DSAPrivate_GenParams dsaprivate;
   M_KeyType_Random_GenParams random;
   M_KeyType_DSAComm_GenParams dsacomm;
   M_KeyType_DHPrivate_GenParams dhprivate;
   M_KeyType_Wrapped_GenParams wrapped;
};
```

- **`M_KeyType_RSAPrivate_GenParams rsaprivate`**. See *RSA* on page 93.
- **`M_KeyType_DSAPrivate_GenParams dsaprivate`**. See *DSA* on page 83.
- **`M_KeyType_Random_GenParams random`**. See *Random* on page 69.
- **`M_KeyType_DSAComm_GenParams dsacomm`**. See *DSA* on page 83.
- **`M_KeyType_DHPrivate_GenParams dhprivate`**. See *Diffie-Hellman and ElGamal* on page 80.
- **`M_KeyType_Wrapped_GenParams wrapped`**. Generating a wrapped key creates a random key block — this may be useful in some key derivation schemes.

DES and Triple DES do not have any key generation parameters. ArcFour and CAST use the same parameters as the key type RANDOM. ElGamal uses key type Diffie-Hellman.

- **`M_ACL acl`**
  See *ACLs* on page 107.
- **`M_AppData *appdata`**
- This is application data. If the command contains application data, the appropriate flag must be set. If no **`appdata`** is provided, the **`appdata`** stored with the key is set to all-bits-zero.
- **`M_ACL aclpriv`**
  ACL for private half
- **`M_ACL aclpub`**
  ACL for public half
- **`M_AppData *appdatapriv`**
  appdata for private half.
- **`M_AppData *appdatapub`**
  appdata for public half.

Reply

```
struct M_Cmd_GenerateKey_Reply {
   M_Cmd_GenerateKey_Reply_flags flags;
   M_KeyID key;
   M_ModuleCert *cert;
};
```

```
struct M_Cmd_GenerateKeyPair_Reply {
   M_Cmd_GenerateKeyPair_Reply_flags flags;
   M_KeyID keypriv;
   M_KeyID keypub;
   M_ModuleCert *certpriv;
   M_ModuleCert *certpub;
};
```

- The following **flags** are defined:
  - **Cmd_GenerateKey_Reply_flags_cert_present**
  - **Cmd_GenerateKeyPair_Reply_flags_cert_present**
    These flags are set if the reply contains a certificate or a certificate pair.
- **M_KeyID key** is $\mathbf{ID}_{KA}$.
- **M_ModuleCert *cert** is a certificate that describes how the key was generated.

```
struct M_ModuleCert {
   M_CipherText signature;
   M_ByteBlock modcertmsg;
};
```

```
struct M_ModCertMsg {
   M_ModCertType type;
   union M_ModCertType__ModCertData data;
};
```

```
union M_ModCertType__ModCertData {
   M_ModCertType_KeyGen_ModCertData keygen;
};
```

```
struct M_ModCertType_KeyGen_ModCertData {
   M_ModCertType_KeyGen_ModCertData_flags flags:
   M_KeyGenParams genparams;
   M_ACL acl;
   M_Hash hka;
};
```

- **M_ModCertType type** From release 1.67.15 and later, this should be type **KeyGen** with code 2. The previous type, now called **OldKeyGen**, did not distinguish between public and private keys and should no longer be used
  The following **flag** is defined:
- **ModCertType_KeyGen_ModCertData_flags_public**
  Set this flag if this is the public half of a key pair.
- **M_KeyGenParams genparams**
  These are the key generation parameters to be used to generate this key.
- **M_ACL acl**
  This is the ACL that was applied to this key when it was created.
- **M_Hash hka**
  This is the SHA-1 hash of the key value.

Notes

If the **Strict_FIPS140** flag was set in the **SetKNSO** command, **GenerateKey** or **GenerateKeyPair** will fail with status **Status_StrictFIPS140** if you attempt to generate a secret key that can be exported as plain

text. A secret key is any key that can have **Sign** or **Decrypt** permissions.

GenerateLogicalToken

Operational state, initialization state	Requires a ClientID May require a KNSO certificate

This command generates a random token key K_T, associates it with the given properties and secret-sharing parameters (**n** and **t**), and encrypts it with the given module key that is identified by its hash, H_{KM}.

The result is stored internally, and an identifier ID_{KT} and a hash H_{KT} are returned. The token is referred to by its identifier in commands and by its hash in ACLs.

Arguments

```
struct M_Cmd_GenerateLogicalToken_Args {
   M_ModuleID module;
   M_KMHash hkm;
   M_TokenParams params;
};
```

- **M_ModuleID module**
  If the module ID is nonzero, the key is loaded onto the specified module. If the module ID is 0, the token is generated on the first available module.
- **M_KMHash hkm** is the H_{KM} of the module key to use to protect this token. If you supply an all zero H_{KM}, the module will use the null module key.

Reply

```
struct M_Cmd_GenerateLogicalToken_Reply {
   M_KeyID idkt;
   M_TokenHash hkt;
};
```

- **M_KeyID idkt** is ID_{KT}
- **M_TokenHash hkt** is H_{KT}

GetChallenge

Operational state, initialization state	Requires a ClientID

The **GetChallenge** command returns a nonce that is used to build a fresh certificate. See *Certificates on page 129*. **GetChallenge** is also used during impath setup.

Arguments

```
struct M_Cmd_GetChallenge_Args {
   M_ModuleID module;
};
```

Reply

```
struct M_Cmd_GetChallenge_Reply {
   M_KMHash nonce;
};
```

GetKML

Operational state, initialization state

This command is used to retrieve a **KeyID** for the module's long-term public key. This key is generated by **InitialiseUnit** and is held internally. $\mathbf{K}_{ML}$ has ACL permissions that allow it to be extracted as plain text, to be used to verify signatures, to view its own ACL, and to extend its ACL.

Arguments

```
struct M_Cmd_GetKML_Args {
   M_ModuleID module;
};
```

Reply

```
struct M_Cmd_GetKML_Reply {
   M_KeyID idka;
};
```

M_KeyID idka is $\mathbf{ID}_{KA}$ for $\mathbf{K}_{ML}$

GetTicket

Operational state, initialization state

Requires a ClientID

This command gets a ticket for a specific **KeyID**. The ticket can then be passed to another client or to an SEE application, which can redeem the ticket for a **KeyID** in its name space.

Tickets can be single-use or permanent, and they can specify the destination.

Note: The program should treat tickets as opaque objects. nShield reserves the right to change the structure of tickets at any time.

Arguments

```
struct M_Cmd_GetTicket_Args {
M_Cmd_GetTicket_Args_flags flags;
M_KeyID obj;
M_TicketDestination dest;
union M_TicketDestination__TicketDestSpec destspec;
};
```

- The following **flags** are defined:
  - **Cmd_GetTicket_Args_flags_Reusable**
    If this flag is set, the ticket can be used multiple times. Otherwise, the ticket can only be used once.
  - **Cmd_GetTicket_Args_flags_HarmlessInfoFlags**
    Set if the nShield server understands new destinations, **TicketDestination_AnyKernelClient** and later. The nShield will set this flag automatically.
- **M_KeyID obj**
  The object for which a ticket is required. This may be any object with a **KeyID**, for example a key, token or SEEWorld.
- **M_TicketDestination dest** are destinations at which this ticket can be redeemed:

```
typedef enum M_TicketDestination {
   TicketDestination_Any =
   TicketDestination_AnyClient =
   TicketDestination_NamedClient =
   TicketDestination_AnySEEWorld =
   TicketDestination_NamedSEEWorld =
   TicketDestination_AnyKernelClient
   TicketDestination__Max =
} M_TicketDestination;
```

- **TicketDestination_Any**
  This specifies any destination. If the nShield server has not set **Cmd_GetTicket_Args_flags_HarmlessInfoFlags** this will not include **TicketDestination_AnyKernelClient** or later destinations.
- **TicketDestination_AnyClient**
  This specifies any client connected to this server.
- **TicketDestination_NamedClient**
  This is the specific client that is named in the **M_TicketDestination__TicketDestSpec**.
- **TicketDestination_AnySEEWorld**
  This specifies any SEEWorld loaded on this module.
- **TicketDestination_NamedSEEWorld**
  This is the specific SEEWorld that is named in the **M_TicketDestination__TicketDestSpec**
- **TicketDestination_AnyKernelClient**
  This specifies any client operating in kernel space. This can only be used if the nShield server reports that the module offers the kernel interface.
- **union M_TicketDestination__TicketDestSpec destspec**

This specifies a specific destination:

```
union M_TicketDestination__TicketDestSpec {
   M_TicketDestination_NamedSEEWorld_TicketDestSpec namedseeworld;
   M_TicketDestination_NamedClient_TicketDestSpec namedclient;
};
```

- **`M_TicketDestination_NamedSEEWorld_TicketDestSpec namedseeworld`**
  This is the **`KeyID`** of the **`SEEWorld`**:

```
struct M_TicketDestination_NamedSEEWorld_TicketDestSpec {
   M_KeyID world;
};
```

- **`M_TicketDestination_NamedClient_TicketDestSpec namedclient`**
  This is the SHA-1 hash of the **`ClientID`**:

```
struct M_TicketDestination_NamedClient_TicketDestSpec {
   M_Hash hclientid;
};
```

Reply

```
struct M_Cmd_GetTicket_Reply {
   M_nest_Ticket ticket;
};
```

`M_nest_Ticket ticket` is a ticket for this object to pass to the destination.

Hash

Operational state, initialization state

This command hashes a message.

There is no limit to the size of the plaintext. This is because the Generic Stub library splits the command into a **`ChannelOpen`** command followed by a number of **`ChannelUpdate`** commands. Only symmetric mechanisms use channels; asymmetric mechanisms cannot.

Arguments

```
struct M_Cmd_Hash_Args {
   M_Cmd_Hash_Args_flags flags;
   M_Mech mech;
   M_PlainText plain;
};
```

- No **flags** are defined.
- **M_Mech mech** - see *Mechanisms* on page 64.
- **M_PlainText plain** This must be in the format **M_PlainTextType_Bytes_Data**.

Reply

```
struct M_Cmd_Hash_Reply {
   M_CipherText sig; Hash
};
```

ImpathKXBegin

Operational state, initialization state	Requires a ClientID

This command creates a new *intermodule path* (impath) and returns a key-exchange message that is to be sent to the peer module.

An impath is a cryptographically secure channel between two nShield nC-series hardware security modules. Data sent through such a channel is secure against both eavesdroppers and active adversaries. The channel can carry arbitrary user data as well as module-protected secrets, such as share data, to be passed directly between modules.

Modules are identified by means of **M_RemoteModule** structures. The elements of a **M_RemoteModule** describe a specific module or a set of modules—for example, those modules that know a particular module key—as well as information about how modules must prove their identity. The **M_RemoteModule** structures are the primary means for describing security policy decisions about impaths.

Note: In many cases you do not need to define the impath yourself. If you use the nCore remote slot commands, the nShield server will create the required impaths automatically.

Arguments

```
struct M_Cmd_ImpathKXBegin_Args {
    M_Cmd_ImpathKXBegin_Args_flags flags;
    M_ModuleID module;
    M_RemoteModule me;
    M_RemoteModule him;
    M_ImpathKXGroupSelection hisgroups;
    M_Nonce n;
    int n_keys;
    M_vec_KeyID keys;
};
```

- No **flags** are defined.
- **M_ModuleID module**
  The module ID of the module which is to be the local end of the impath.
- **M_RemoteModule me**
  This is an **M_RemoteModule** structure describing the local module. It must exactly match the him structure being used at the other end of the impath.
- **M_RemoteModule him**
  This is an **M_RemoteModule** structure describing the peer module. It must exactly match the **me** structure being used at the other end of the impath.
- **M_ImpathKXGroupSelection hisgroups**
  This is the peer module's list of supported key-exchange groups. This list can be obtained, for example, by using the **NewEnquiry** command on the remote module. The list is used to select the key-exchange group that is to be used when setting up the impath.
- **M_Nonce n**
  This is a challenge obtained from the remote module by using the **GetChallenge** command.
- **int n_keys** is the size of the **keys** table
- **M_vec_KeyID keys**
  This is a table of **KeyIDs** for the user keys whose hashes are listed in **me.hks**. The keys must have the **SignModuleCert** permission enabled. User keys may be either private or symmetric.

Reply

```
struct M_Cmd_ImpathKXBegin_Reply {
   M_ImpathID imp;
   M_ByteBlock kx;
};
```

- **M_ImpathID imp**
  This is the ID for this impath. After the impath is no longer required, it can be disposed of by using the **Destroy** command.
- **M_ByteBlock kx**
  This is a key-exchange message that is to be transmitted to the peer module. (See *ImpathKXFinish* on page 190.)

ImpathKXFinish

Operational state, initialization state	Requires a ClientID

This command completes an impath (intermodule path) key exchange. It leaves the impath ready for data transmission and receipt.

Arguments

```
struct M_Cmd_ImpathKXFinish_Args {
   M_Cmd_ImpathKXFinish_Args_flags flags;
   M_ImpathID imp;
   M_NetworkAddress *addr;
   int n_keys;
   M_vec_KeyID keys;
   M_ByteBlock kx;
};
```

- The following **flag** is defined:
  - **Cmd_ImpathKXFinish_Args_flags_addr_present**
    Indicates whether the **M_NetworkAddress *addr** is present.
- **M_ImpathID imp** is the ID for the impath
- **M_NetworkAddress *addr**
  This is the network address of the peer host. If supplied, this is compared against the **addr** field in the **him** structure given to the **ImpathKXBegin** command.
- **int n_keys** is the size of the **keys** table.
- **M_vec_KeyID keys**
  This is a table of **KeyIDs** for the user keys, public or symmetric, whose hashes were listed in the **hks** table in the **him** structure given to the **ImpathKXBegin** command.
- **M_ByteBlock kx**
  This is the key-exchange message returned by **ImpathKXBegin** on the peer module.

Reply

The reply structure for this command is empty.

ImpathReceive

Operational state, initialization state	Requires a ClientID

This command decrypts a user-data message that was encrypted using an impath.

Arguments

```
struct M_Cmd_ImpathReceive_Args {
   M_ImpathID imp;
   M_ByteBlock cipher;
};
```

- **`M_ImpathID imp`** is the ID for the impath.
- **`M_ByteBlock cipher`** is the cipher text emitted by an **`ImpathSend`** command issued to the peer module. Each cipher text message can be received once only, in order to prevent replay attacks.

Reply

```
struct M_Cmd_ImpathReceive_Reply {
   M_ByteBlock data;
};
```

`M_ByteBlock data` is a recovered plain text message.

ImpathSend

Operational state, initialization state	Requires a ClientID

This command encrypts a user message using an impath's keys, ready for transmission to the peer host.

Arguments

```
struct M_Cmd_ImpathSend_Args {
   M_Cmd_ImpathSend_Args_flags flags;
   M_ImpathID imp;
   M_ByteBlock data;
};
```

- No **`flags`** are defined.
- **`M_ImpathID imp`** is the ID for the impath.
- **`M_ByteBlock data`** is the message to be sent.

Reply

```
struct M_Cmd_ImpathSend_Reply {
   M_ByteBlock cipher;
};
```

M_ByteBlock cipher is the cipher text corresponding to the given plain text data. The plain text can be recovered by issuing an **ImpathReceive** command to the peer module.

InitialiseUnit

Pre-initialization state, initialization state	"Privileged" users only

This command causes a module in the pre-initialization state to enter the initialization state.

When the module enters the initialization state, it erases all module keys $\mathbf{K}_{M}$, including $\mathbf{K}_{M0}$. It also erases the module's signing key, $\mathbf{K}_{ML}$, and the hash of the Security Officer's keys, $\mathbf{H}_{KNSO}$. It does not erase the long-term $\mathbf{K}_{LF}$ key. It then generates a new $\mathbf{K}_{ML}$ and $\mathbf{K}_{M0}$.

In order to use the module after it has been initialized, you must set a new Security Officer's key.

Note: When the module is in the pre-initialization state, you cannot obtain a ClientID. In order to use commands that require a ClientID, use the **NewClient** command after the module enters the Initialization state.

Arguments

```
struct M_Cmd_InitialiseUnit_Args {
   M_ModuleID module;
};
```

Reply

The reply structure for this command is empty.

LoadBlob

Operational state, initialization state	Requires a ClientID

This command allows a key blob to be loaded into the module. If this process is successful, a new $\mathbf{ID}_{KA}$ handle will be generated and returned.

For $\mathbf{K}_{M}$ blobs, the required $\mathbf{K}_{M}$ value must be present in the module's internal storage.

For $\mathbf{K}_{T}$ blobs, the logical token containing $\mathbf{K}_{T}$ must be "present". This is not possible if the $\mathbf{K}_{M}$ associated with that $\mathbf{K}_{T}$ is not present in the module. See *GenerateLogicalToken* on page 184 and *LoadLogicalToken* on page 193.

For the archival key blobs $\mathbf{K}_{i}$ or $\mathbf{K}_{AR}$, the appropriate key object must be loaded.

Arguments

```
struct M_Cmd_LoadBlob_Args {
   M_Cmd_LoadBlob_Args_flags flags;
   M_ModuleID module;
   M_ByteBlock blob;
   M_KeyID *idkb;
} M_Cmd_LoadBlob_Args;
```

- The following **flag** is defined:
  Cmd_LoadBlob_Args_flags_idkb_present
  See ***idkb** below.
- **M_ModuleID module** is the module id.
- **M_ByteBlock blob** is a key blob.
- **M_KeyID *idkb**
  In order to load a blob encrypted under a token or recovery key, set the **idkb_present** flag and include the identifier of either the token or the recovery key (**ID**$_{KT}$ for tokens, **ID**$_{KAR}$ for recovery keys) in the data as idkb. Otherwise, do not set **idkb_present**, and set **idkb** to **NULL**.

Reply

```
struct M_Cmd_LoadBlob_Reply {
    M_KeyID idka;
};
```

M_KeyID idka is **ID**$_{KA}$.

LoadLogicalToken

Operational state, initialization state	Requires a ClientID May require a KNSO certificate

This command is used to initiate loading a token from shares.

The command returns an **ID**$_{KT}$. The token and any loaded shares can be removed by issuing the **Destroy** command with this identifier.

When this command is issued, the module allocates space for a share-reassembly process. In order to assemble the token, the application must issue one or more **ReadShare** commands (see *ReadShare* on page 198).

Arguments

```
struct M_Cmd_LoadLogicalToken_Args {
   M_ModuleID module;
   M_TokenHash hkt;
   M_KMHash hkm;
   M_TokenParams params;
};
```

- **`M_ModuleID module`** is the module ID of the module. If you enter a module ID of 0, the command returns with status **`InvalidParameter`**.
- **`M_TokenHash hkt`** is $\mathbf{H}_{KT}$
- **`M_KMHash hkm`** is the $\mathbf{H}_{KM}$ of the module key that is to be used to protect this token. If you supply an all-zero HKM, the module will use the null module key.
- **`M_TokenParams params`**
  The shares information must match that which was given when the token was generated. The flags and time limit are read from the token, and values set in the command are ignored.

Reply

```
struct M_Cmd_LoadLogicalToken_Reply {
   M_KeyID idkt;
};
```

`M_KeyID idkt` is the $\mathbf{ID}_{KT}$.

MakeBlob

Operational state, initialization state	Requires a ClientID

This command requests that the module generate a key blob using a key whose identifier is given. The ACL for the key must allow the key to be exported as a blob, otherwise the command will fail.

The ACL for the key $\mathbf{ID}_{KA}$ must have a **`MakeBlob`** entry (for Module and Token blobs) or **`MakeArchiveBlob`** entry (for Direct or Indirect blobs) which permits making a blob with the requested parameters.

For a $\mathbf{K}_{M}$ key, the relevant key must be stored internally within the module.

For a $\mathbf{K}_{T}$ key, the logical token containing this key must be "present". Otherwise, the handle of another key object can be given to encrypt the blob. To succeed, the key object needs a **`UseAsBlobKey`** permission.

Arguments

```
struct M_Cmd_MakeBlob_Args {
   M_Cmd_MakeBlob_Args_flags flags;
   M_BlobFormat format;
   M_KeyID idka;
   union M_BlobFormat__MkBlobParams blobkey;
   M_ACL *acl;
   M_MakeBlobFile *file;
};
```

- The following **flags** are defined:
  - **Cmd_MakeBlob_Args_flags_acl_present**
    Set this flag if the command contains a new ACL.
  - **Cmd_MakeBlob_Args_flags_file_present**
    Set this flag to store the blob in an NVRAM or smart card file, defined by the **M_MakeBlobFile**.
- **M_BlobFormat format**
  The following formats are defined:
  - **BlobFormat_Module**
    Blob encrypted by a module key.
  - **BlobFormat_Token**
    Blob encrypted by a Logical Token.
  - **BlobFormat_Direct**
    Blob encrypted by a symmetric archiving key. Currently only Triple DES keys may be used.
  - **BlobFormat_Indirect**
    Blob encrypted by an public archiving key, this requires the private key to decrypt. Currently only RSA keys may be used.
  - **BlobFormat_UserKey**
    Not yet supported.
- **union M_BlobFormat__MkBlobParams blobkey**
  The following MKBlobParams are defined for the four different blob types:

```
struct M_BlobFormat_Direct_MkBlobParams {
   M_KeyID idki;
};
```

```
struct M_BlobFormat_Indirect_MkBlobParams {
   M_KeyID idkr;
   M_Mech mech;
};
```

```
struct M_BlobFormat_Module_MkBlobParams {
   M_KMHash hkm;
};
```

```
struct M_BlobFormat_Token_MkBlobParams {
   M_KeyID idkt;
};
```

```
struct M_BlobFormat_UserKey_MkBlobParams {
   M_KeyID idkr;
   M_Mech mech;
 };
```

```
union M_BlobFormat__MkBlobParams {
   M_BlobFormat_Module_MkBlobParams module;
   M_BlobFormat_Token_MkBlobParams token;
   M_BlobFormat_Direct_MkBlobParams direct;
   M_BlobFormat_Indirect_MkBlobParams indirect;
   M_BlobFormat_UserKey_MkBlobParams userkey;
};
```

- **`M_KeyID idki`**
  This is the **`KeyID`** of a Triple DES key that is to be used to encrypt the blob.
- **`M_KeyID idkr`**
  This is the **`KeyID`** of the public key that is to be used to encrypt the blob.
- **`M_Mech mec`**
  This is the public key mechanism that is to be used to encrypt the blob.
- **`M_KMHash hkm`**
  This is the hash of the module key that is to be used to encrypt the blob.
- **`M_KeyID idkt`**
  This is the **`KeyID`** of the token that is to be used to encrypt the blob.
- **`M_ACL *acl`**
  This is either an ACL to be used for the key blob or **`NULL`**. If no ACL is specified, the loaded key's existing ACL is recorded in the blob. See *ACLs* on page 107.
  The ACL created for the blob does not include permission groups that have global limits (as opposed to per-authorization limits).
  The permissions of the new ACL must be a subset of those specified by the existing ACL. For more information, see *SetACL* on page 202.
- **`M_MakeBlobFile *file`**
  A structure defining the file in which to store the blob.

```
struct M_MakeBlobFile {
   M_MakeBlobFile_flags flags;
   M_KeyID kacl;
   M_PhysToken file;
};
```

- No **`flags`** are defined.
- **`M_KeyID kacl`**

The **KeyID** of a template key defining the ACL for this file. This ACL must contain the **LoadBlob** permission.

- **M_PhysToken file**
  A **FileSpec** specifying the location of the file.

Reply

```
struct M_Cmd_MakeBlob_Reply {
   M_ByteBlock blob;
};
```

M_ByteBlock blob is a **KeyBlob**.

MergeKeyIDs

All non-error states Requires a ClientID

In situations where one key has been loaded onto several modules, this key will have a different **KeyID** on each module. The **MergeKeyIDs** command takes a list of **KeyIDs**, which are assumed to refer to the same key, and creates a new **KeyID** that can be used to refer to the key on any module. This facilitates load sharing and fail-over strategies.

Arguments

```
struct M_Cmd_MergeKeyIDs_Args {
   int n_keys;
   M_vec_KeyID keys;
} ;
```

- **int n_keys** is the number of keys.
- **M_vec_KeyID keys** is a list of $\mathbf{ID}_{KA}$.

Reply

```
struct M_Cmd_MergeKeyIDs_Reply {
   M_KeyID newkey;
};
```

M_KeyID newkey is $\mathbf{ID}_{KA}$

Notes

MergeKeyIDs does not check to see whether the supplied **KeyIDs** actually refer to the same key.

Merged **KeyIDs** may not themselves be supplied to **MergeKeyIDs**.

A merged **KeyID** will continue to work even if some of the modules containing the component **KeyIDs** are reset or fail, though performance may be reduced in such cases. The merged **KeyID** will only stop working after all the modules containing the component **KeyIDs** are reset or fail.

MergeKeyIDs can be used to group keys, logical tokens, SEE Worlds, and any other objects that are referred to by a **KeyID** and destroyed by **Destroy**.

The server does not attempt to ensure that the merged **KeyIDs** refer to the same underlying data, or even to the same types of objects.

ReadShare

Operational state, initialization state	Requires a ClientID

This command is used to assemble a logical token from shares.

Note: The smart card architecture keeps public data storage areas separate from the areas that are used to store logical token shares. Specifically, if a given piece of information can be read or written with **ReadShare** or **WriteShare**, then it cannot be read or written with **ReadFile** or **WriteFile**. The converse is also true.

Arguments

```
struct M_Cmd_ReadShare_Args {
   M_Cmd_ReadShare_Args_flags flags;
   M_PhysToken token;
   M_KeyID idkt;
   M_Word i;
   M_PIN *pin;
};
```

- The following **flags** are defined:
  - **Cmd_ReadShare_Args_flags_pin_present**
    This flag must be set if the input includes a PIN.
    Note: If the slot uses the ProtectedPINPath, do not include the PIN with the command.
  - **Cmd_ReadShare_Args_flags_UseLimitsUnwanted**
    If this flag is set the module does not allocate Per-Authorisation Use limits to this logical token. Keys protected by the assembled local token will only be permitted to perform actions that do not have use limits. Per authorisation use limits can only be allocated to one logical token for each insertion of the card. However, it is possible that the logical token is required on several modules, or by several clients on one module. Therefore, you should set this flag, if you are aware that you do not need the **uselimits** and wish to make them available elsewhere.
- **M_KeyID idkt** is the $\mathbf{ID}_{KT}$.
- **M_Word i** is share number **i**.
- **M_PIN *pin**
  If the share is protected by a PIN, this must be specified in order to successfully decrypt the share, otherwise **pin** must be a **NULL** pointer. If the input includes a PIN, the **pin_present** flag must be set.

Reply

```
struct M_Cmd_ReadShare_Reply {
   M_Word sharesleft;
};
```

M_Word sharesleft is the number of shares that are still required in order to recreate the token. You can issue further **ReadShare** commands when the shares are present.

A **sharesleft** value of 0 indicates that all shares are present. At that point, the module will automatically assemble the token.

Notes

If an error occurs during an individual share reading operation (because of, for example, an incorrect PIN or the wrong token), the current state of the logical token is retained, and the operation can simply be repeated.

If an error occurs during the final share reassembly process (implying that the shares have been corrupted in some way), the logical token is invalidated, and **Status_TokenReassemblyFailed** is returned. The token must then be destroyed, and the whole operation must be restarted.

At any time during the share reassembly sequence, the host can abort it (and clear the reassembly buffer) by calling **Destroy** with the given $\mathbf{ID}_{KT}$. If the client closes before the token has been assembled, the server automatically issues the **Destroy** command.

RedeemTicket

Operational state, initialization state	Requires a ClientID

This command gets a **KeyID** in return for a key ticket.

Arguments

```
struct M_Cmd_RedeemTicket_Args {
   M_Cmd_RedeemTicket_Args_flags flags;
   M_ModuleID module;
   M_nest_Ticket ticket;
};
```

- No **flags** are defined.
- **M_ModuleID module**
  This specifies the module ID of the module that contains this object.
- **M_nest_Ticket ticket**
  This is the ticket that is supplied by **GetTicket**.

Reply

```
struct M_Cmd_RedeemTicket_Reply {
   M_KeyID obj;
};
```

`M_KeyID obj` is the new KeyID for this object.

RemoveKM

	Requires a ClientID
Operational state, initialization state	May require a KNSO certificate
	"Privileged" users only

This command deletes a given K_M value from permanent storage. The deletion process overwrites the value in order to ensure its destruction.

Note: K_{M0} cannot be deleted.

Arguments

```
struct M_Cmd_RemoveKM_Args {
   M_ModuleID module;
   M_Cmd_RemoveKM_Args_flags flags;
   M_KMHash hkm;
};
```

- **`M_ModuleID module`** is the **`ModuleID`**.
- No **`flags`** are defined.
- **`M_KMHash hkm`** is H_{KM}.

Reply

The reply structure for this command is empty.

RSAImmedSignDecrypt

Operational state, initialization state

This command performs RSA decryption by using an RSA private key that is provided in plain text.

Arguments

```
struct M_Cmd_RSAImmedSignDecrypt_Args {
   M_Bignum m;
   M_Bignum k_p;
   M_Bignum k_q;
   M_Bignum k_dmp1;
   M_Bignum k_dmq1;
   M_Bignum k_iqmp;
};
```

- **`M_Bignum m`** Ciphertext
- **`M_Bignum k_p`** P modulus first factor
- **`M_Bignum k_q`** Q modulus first factor
- **`M_Bignum k_dmp1`** D MOD_{P-1}
- **`M_Bignum k_dmq1`** D MOD_{Q-1}
- **`M_Bignum k_iqmp`** Q^{-1} MOD_{P}

Reply

```
struct M_Cmd_RSAImmedSignDecrypt_Reply {
   M_Bignum r;
};
```

`M_Bignum r` is plain text .

Notes

The plain text and cipher text are in the nShield bignum format.

No padding is done.

RSAImmedVerifyEncrypt

Operational state, initialization state

This command performs RSA encryption with an RSA public key provided in plain text.

Arguments

```
struct M_Cmd_RSAImmedVerifyEncrypt_Args {
   M_Bignum a;
   M_Bignum k_e;
   M_Bignum k_n;
};
```

- **M_Bignum a** Message
- **M_Bignum k_e** Key exponent
- **M_Bignum k_n** Key modulus

Reply

Uses **M_Cmd_RSAImmedSignDecrypt_Reply** .

Notes

The plain text and cipher text are in nShield bignum format.

No padding or unpadding is performed.

SetACL

Operational state, initialization state	Requires a ClientID

This command replaces the ACL of a given key object with a new ACL. The existing ACL must have either **ExpandACL** or **ReduceACL** permission. If the existing ACL only includes the **ReduceACL** permission, you must set the **Cmd_SetACL_Args_flags_reduce** flag, and also the new ACL must be a subset of the existing ACL.

Arguments

```
struct M_Cmd_SetKM_Args {
   M_Cmd_SetKM_Args_flags flags;
   M_KeyID idka;
   M_ACL *acl;
};
```

- The following **flag** is defined:
  - **Cmd_SetACL_Args_flags_reduce**
    If this flag is not set, the command checks the **ExpandACL** permission in the existing ACL.
    However, if this flag is set:
    - the command checks the **ReduceACL** permission in the existing ACL
    - the new ACL must be a subset of the existing ACL
- **M_KeyID idka** is $\mathbf{ID}_{KA}$.
- **M_ACL *acl** is the new ACL for the key.

Reply

The reply structure for this command is empty.

Notes

The new ACL will be a subset of the original ACL if for every action in the new ACL there exists an entry in the existing ACL in a permission group with:

- the same certifier or no certifier
- the same or more restrictive **FreshCerts** flag
- use limits that are at least as permissive as those in the new ACL

The use limits are considered to be as permissive as those in the new ACL if for each limit in the original ACL there is a limit in the new ACL:

- of the same type, global or per-authorization
- with the same limit ID
- with a use count and a time limit that are no greater than those in the original.

The following changes count as reducing an ACL:

- adding a certifier or **NSOCertified** to a group
- adding **UseLimits** to a group that did not have them previously
- adding a time limit or a use count to a use limit that did not have one previously
- reducing an existing use count or time limit
- adding a module serial number to a group.

The following changes do *not* count as reducing an ACL:

- changing the certifier for a group
- changing the module serial number for a group
- changing a use count to a time limit or changing a time limit to a use count
- changing from **NSOCertified** to a specific certifier or changing from a specific certifier to **NSOCertified**.

Note: If the **Strict_FIPS140** flag was set in the **SetKNSO** command, then **SetACL** will fail with status **Status_FIPS_Compliance** if you attempt to add **ExportAsPlain** to the ACL of a secret key. A secret key is any key that can have **Sign** or **Decrypt** permissions.

If you want to record the new ACL permanently, you must make a new blob of the key.

SetKM

	Requires a ClientID
Operational state, initialization state	May require a KNSO certificate
	"Privileged" users only

This command allows a key object to be stored internally as a module key ($\mathbf{K}_M$) value. The $\mathbf{K}_M$ value is derived from the key material given by $\mathbf{ID}_{KA}$. The ACL and other information associated with $\mathbf{ID}_{KA}$ are not stored.

Arguments

```
struct M_Cmd_SetKM_Args {
   M_ModuleID module;
   M_Cmd_SetKM_Args_flags flags;
   M_KeyID idka;
};
```

- **`M_ModuleID module`**
- No **`flags`** are defined.
- **`M_KeyID idka`** is $\mathbf{ID}_{KA}$.
  $\mathbf{K}_{A}$ must be a DES3 key with **`UseAsKM`** permission.

Reply

The reply structure for this command is empty.

Notes

If you attempt to set as a $\mathbf{K}_{M}$ a key that has the same hash as an existing $\mathbf{K}_{M}$, then **`SetKM`** will overwrite the existing module key with the new key. If you are attempting to overwrite $\mathbf{K}_{M0}$, the command will return **`Status_AccessDenied`**.

SetNSOPerms

Initialization state only	Requires a ClientID "Privileged" users only

The **`SetNSOPerms`** command stores the key hash $\mathbf{H}_{KA}$, which is returned by **`GetKeyInfo`** as the new Security Officer's key.

It also determines which operations require a KNSO certificate.

Note: The **`SetNSOPerms`** command requires you to set a flag if you want an operation to be allowed without a certificate. This is the opposite behavior to the **`SetKNSO`** command.

This command may only be called once after each use of **`InitialiseUnit`** (see *InitialiseUnit* on page 192). After it is set, the Security Officer's key can only be changed by completely reinitializing the module.

Arguments

```
struct M_Cmd_SetNSOPerms_Args {
   M_ModuleID module;
   M_Cmd_SetNSOPerms_Args_flags flags;
   M_KeyHash hknso;
   M_NSOPerms publicperms;
};
```

- **M_ModuleID module** is the module id
- The following **flag** is defined:
  - **Cmd_SetNSOPerms_Args_flags_FIPS140Level3**
    If this flag is set, the module adopts a security policy that complies with FIPS 140-2 level 3. This enforces the following restrictions:
    - the **Import** command fails if you attempt to import a key of a type that can be used to sign or decrypt messages
      Note: Use of the **Import** command for other key types requires a K_{NSO} certificate.
    - **GenerateKey** and **GenerateKeyPair** require K_{NSO} certificates
    - **GenerateKey** and **GenerateKeyPair** fail if you attempt to generate a key of a type that can be used to sign or decrypt messages with an ACL that allows **ExportAsPlain**
    - **SetACL** fails if you attempt to add the **ExportAsPlain** action to the ACL of a key of a type that can be used to sign or decrypt messages.
      All cryptographic mechanisms which do not use a FIPS-approved algorithm are disabled. (This restriction is new for firmware versions 2.18.13 and later).
      Cryptographic algorithms which are *disabled* are: ArcFour, Blowfish, CAST, CAST256, HAS160, KCDSA, MD2, MD5, RIPEMD160, SEED, Serpent, SSLMasterSecret mechanisms, Tiger, Twofish.
      The following algorithms are *unaffected*: DES, DES2, DES3, Diffie-Hellman, DSA, Rijndael (AES), RSA, SHA-1, SHA-256, SHA-384 and SHA-512

    Note: In order to fully comply with FIPS 140-2 level 3 you must also ensure that none of the following are set: **NSOPerms_ops_ReadFile, NSOPerms_ops_WriteFile, NSOPerms_ops_EraseShare, NSOPerms_ops_EraseFile, NSOPerms_ops_FormatToken, NSOPerms_ops_GenerateLogToken, NSOPerms_ops_SetKM, NSOPerms_ops_RemoveKM**.
- **M_KeyHash hkns** is H_{KA} to set as H_{KNSO}
- **M_NSOPerms publicperms**
  The **NSOPerms** word is a bit map that determines which operations do *not* require a certificate from the nShield Security Officer. These certificates can be reusable. The following flags are defined:
  - **NSOPerms_ops_LoadLogicalToken**
  - **NSOPerms_ops_ReadFile**
  - **NSOPerms_ops_WriteShare**
  - **NSOPerms_ops_WriteFile**
  - **NSOPerms_ops_EraseShare**
  - **NSOPerms_ops_EraseFile**
  - **NSOPerms_ops_FormatToken**
  - **NSOPerms_ops_SetKM**
  - **NSOPerms_ops_RemoveKM**
  - **NSOPerms_ops_GenerateLogToken**
  - **NSOPerms_ops_ChangeSharePIN**
  - **NSOPerms_ops_OriginateKey** Not allowed in **SetKNSO**
  - **NSOPerms_ops_NVMemAlloc** Not allowed in **SetKNSO**
  - **NSOPerms_ops_NVMemFree** Not allowed in **SetKNSO**
  - **NSOPerms_ops_GetRTC** Not allowed in **SetKNSO**
  - **NSOPerms_ops_SetRTC** Not allowed in **SetKNSO**
  - **NSOPerms_ops_DebugSEEWorld** Not allowed in **SetKNSO**
  - **NSOPerms_ops_SendShare** Not allowed in **SetKNSO**
  - **NSOPerms_ops_ForeignTokenOpen** Not allowed in **SetKNSO**

Reply

The reply structure for this command is empty.

Notes

Note: Modules that are supplied by nShield are initialized with no operations that require $\mathbf{K}_{NSO}$ certificates. This means that the key whose hash is installed as $\mathbf{H}_{KNSO}$ is irrelevant.

SetRTC

	Requires an SEE-Ready module
Operational state, initialization state	May require a KNSO certificate
	"Privileged" users only

Arguments

```
struct M_Cmd_SetRTC_Args {
   M_ModuleID module;
   M_Cmd_SetRTC_Args_flags flags;
   M_RTCTime time;
};
```

- **`M_ModuleID module`**
  The module ID of the module. If you enter a module ID of 0, the command returns with status **`InvalidParameter`**.
- The following **`flag`** is defined:
  - **`Cmd_SetRTC_Args_flags_adjust`**
    If this flag is set, the module calculates the difference between the current time according to the RTC and the time supplied in the command. Next, it divides this difference by the length of time since the clock was last set in order to determine a drift rate. The result of all future calls to **`GetRTC`** is corrected using this drift rate. The command returns status **`OutOfRange`** if the implied drift rate is larger than the chip's guaranteed maximum drift rate. If, however, this flag is *not* set, the module will clear any current drift rate adjustment.
- **`M_RTCTime time`** is the new time.

Reply

The reply structure for this command is empty.

Sign

Operational state, initialization state	Requires a ClientID

This command signs a message with a stored key.

For information on formats, see *Encrypt* on page 176.

Sign pads the message as specified by the relevant algorithm, unless you use plaintext of the type **Bignum**.

Note: You cannot sign a message that is longer than the maximum size of an nShield command. In order to sign longer messages, use the **Hash** command first, and then call **Sign** with the appropriate **Hash** plain text type.

Arguments

```
struct M_Cmd_Sign_Args {
   M_Word flags;
   M_KeyID key;
   M_Mech mech;
   M_PlainText plain;
   M_IV *given_iv
};
```

- No **flags** are defined.
- **M_KeyID key** is the $\mathbf{ID}_{KA}$.

Reply

```
struct M_Cmd_Sign_Reply {
   M_CipherText sig;
};
```

SignModuleState

Operational state, initialization state	Requires a ClientID

SignModuleState makes the module generate a signed Module Certificate that contains data about the current state of the module. Optionally, a **challenge** value may be supplied to provide a provably fresh certificate.

Arguments

```
struct M_Cmd_SignModuleState_Args{
    M_ModuleID module;
    M_Cmd_SignModuleState_Args_flags flags;
    M_SignerType enum;
    M_Nonce challenge;
    M_wrap_vec_ModuleAttribTag *attribs;
};
```

- The following **flags** are defined:
  - **Cmd_SignModuleState_Args_flags_challenge_present**
    This flag must be set if the command contains a challenge.
  - **Cmd_SignModuleState_Args_flags_attribs_present**
    This flag must be set if the command contains Module Attribute Tags. If not set the module delivers a default set of attributes.
- **SignerType** can have the following values:
  - **KLF**: The certificate is signed by the KLF long-term key. **Status_NotAvailable** is returned if this key has not been set.
  - **KML**: The certificate is signed by the KML key. This is always available (except in pre-initialization mode, when the command is not accepted anyway).
  - **Appkey**: The certificate is signed a user key, using the given mechanism (which can be **Mech_Any**). The key must have a new OpPermission bit in its ACL, called **SignModuleCert**. **SignModuleCert** is a less generate permission than **Sign**: the module uses it only to sign well-formed messages whose content it believes to be true. **Sign** permission doesn't imply **SignModuleCert** permission.
- **M_wrap_vec_ModuleAttribTag *attribs** is a list of the attributes to include in the signed message

```
struct M_wrap_vec_ModuleAttribTag {
   int n;
   M_vec_ModuleAttribTag v;
};
```

The following attributes are defined:

- **ModuleAttribTag_None**
- **ModuleAttribTag_Challenge** (default if included in command)
- **ModuleAttribTag_ESN** (default)
- **ModuleAttribTag_KML** (default)
- **ModuleAttribTag_KLF** (default)
- **ModuleAttribTag_KNSO** (default)
- **ModuleAttribTag_KMList** (default)
- **ModuleAttribTag_PhysSerial**
- **ModuleAttribTag_PhysFIPSl3**
- **ModuleAttribTag_FeatureGoldCert**
- **ModuleAttribTag_Enquiry**
- **ModuleAttribTag_AdditionalInfo**
- **ModuleAttribTag_ModKeyInfo**

Reply

The reply structure for this command is as follows:

```
struct M_Cmd_SignModuleState_Reply {
    M_ModuleCert *cert;
};
```

M_ModuleCert *cert is a certificate that describes how the key was generated.

```
struct M_ModuleCert {
    M_CipherText signature;
    M_ByteBlock modcertmsg;
};
```

```
struct M_ModCertMsg {
    M_ModCertType type;
    union M_ModCertType__ModCertData data;
};
```

```
union M_ModCertType__ModCertData {
    M_ModCertType_KeyGen_ModCertData keygen;
};
```

```
struct M_ModCertType_KeyGen_ModCertData {
    M_ModCertType_KeyGen_ModCertData_flags flags:
    M_KeyGenParams genparams;
    M_ACL acl;
    M_Hash hka;
};
```

- **`M_ModCertType type`** is one of the following:
  - **`None`**
  - **`Challenge`**: appears if a challenge is present in the **`SignModuleState`** command
  - **`ESN`**: ASCII string
  - **`KML`** : KML key, defined with key hash and key data
  - **`KLF`**: KLF key, defined with key hash and key data
  - **`KNSO`**: not present if module is in initialization mode
  - **`KMList`**
- The following **`flag`** is defined:
  - **`ModCertType_KeyGen_ModCertData_flags_public`**
    Set this flag if this is the public half of a key pair.
- **`M_KeyGenParams genparams`**
  These are the key generation parameters to be used to generate this key.
- **`M_ACL acl`**
  This is the ACL that was applied to this key when it was created.
- **`M_Hash hka`**
  This is the SHA-1 hash of the key value.

StaticFeatureEnable

Operational state, initialization state

This command is used to enable a purchased feature. It requires a certificate signed by the nShield master feature enabling key, KSA, authorizing the feature on the specified module.

Use the **fet** command-line utility to perform this function.

Arguments

```
struct M_Cmd_StaticFeatureEnable_Args {
   M_ModuleID module; Module ID
   M_FeatureInfo info;
};
```

M_FeatureInfo info is a description of the feature to authorize

Reply

The reply structure for this command is empty.

UpdateMergedKey

All non-error states	Processed by the nShield Server.

This command allows a merged key set to be manipulated, listed, or both.

Arguments

```
struct M_Cmd_UpdateMergedKey_Args {
    M_PlainText mkey; IDKA
    M_Cmd_UpdateMergedKeys_Args_flags flags
    int n_addkeys;
    M_KeyID *addkeys;
    int n_delkeys;
    M_KeyID *delkeys;
};
```

- **M_PlainText mkey** (**ID**$_{KA}$) is a merged key set created with **MergeKeyIDs**.
- The following **flags** are defined:
  - **Cmd_UpdateMergedKey_Args_flags_ListWorking**
    If this flag is set, the keys in the resulting merged key that are in working modules are returned.
  - **Cmd_UpdateMergedKey_Args_flags_ListNonworking**
    If this flag is set, the keys in the resulting merged key that are not in working modules are returned.
    These two flags can be set together if required.
- **M_KeyID *addkeys** is a table of keys to be added to the merged key.
  Merged key IDs that currently contain no key IDs are allowed.
- **M_KeyID *delkeys** is a table of keys to be deleted from the merged key.
  Note: Including a key in this list deletes all copies of the specified key.

Reply

```
struct M_Cmd_UpdateMergedKey_Reply {
   int n_keys;
   M_KeyID *keys;
};
```

M_KeyID *keys is a table containing the merged key that results once the specified keys are added and deleted from the input merged key.

If **ListWorking** is set, keys in working modules are included; if **ListNonWorking** is set, keys not in working modules are included. If both are set, all keys are included.

Notes

You cannot add a merged key to another merged key, or delete a merged key from another merged key.

The same key can be present more than once in a merged key.

The keys specified in **addkeys** are added to the target merged key first. The keys specified in **delkeys** are then deleted. This means that if the same key is present in both **addkeys** and **delkeys**, it is not present in the resulting merged key.

Verify

Operational state, initialization state	Requires a ClientID

This command verifies a digital signature. It returns **Status_OK** if the signature verifies correctly and **Status_VerifyFailed** if the verification fails.

The limit of 8K does not apply to data signed by this command. This is because the Generic Stub library splits the command into a **ChannelOpen** command followed by a number of **ChannelUpdate** commands.

For information on formats, see *Sign* on page 206.

Arguments

```
struct M_Cmd_Verify_Args {
    M_Cmd_Verify_Args_flags flags;
    M_KeyID key;
    M_Mech mech;
    M_PlainText plain;
    M_CipherText sig;
};
```

- No **flags** are defined.
- **M_KeyID key**: ID_{KA}
- **M_Mech mech**: set **Mech_Any** in order to use the mechanism specified in the signature. If you specify a mechanism, **Verify** will compare this with the mechanism in the signature and return **Status_MechanismNotExpected** if the mechanisms do not match.
- **M_PlainText plain**: message.
- **M_CipherText sig**: signature.

Reply

The reply structure for this command is empty.

WriteShare

Operational state, initialization state	Requires a ClientID May require a KNSO certificate

This command creates one share of a logical token and writes it to a smart card identified by the **SlotID, insertion counter** pair. The **i** value identifies the share number. This command needs to be given once for each share that is to be generated.

Arguments

```
struct M_Cmd_WriteShare_Args {
    M_Cmd_WriteShare_Args_flags flags;
    M_PhysToken token;
    M_KeyID idkt;
    M_Word i;
    M_PIN *pin;
    M_ACL *acl;
};
```

- The following **flags** are defined:
  - **Cmd_WriteShare_Args_flags_pin_present**
    This flag must be set if the input includes a pass phrase.
  - **Cmd_WriteShare_Args_flags_UseProtectedPINPath**
    Set this flag if the token reads a pass phrase by means of a protected path. However, this feature is not currently implemented.
  - **Cmd_WriteShare_Args_flags_acl_present**
    Set this flag if the command contains an ACL for the share.

  Note: Setting both **pin_present** and **UseProtectedPINPath** will cause the command to fail with **InvalidParameter**.
- **M_KeyID idkt**: ID_{KT}
- **M_Word i** is the share number for the share you are writing. Share numbers start at 0. Each share in a token can only be written once.
- **M_ACL *acl** is an ACL for this share. If no ACL is specified, a default ACL is assumed, containing a single **ReadShare** action without any flags set and requiring no certification.

Note: If any shares of a logical token are to have an ACL set, you must set an ACL for all of them. Shares with ACLs cannot be read in modules running firmware earlier than version 1.75.0.

Reply

The reply structure for this command is empty.

Commands used by the generic stub only

The following commands are used by the generic stub library to connect to the module.

- **`ExistingClient`**
- **`NewClient`**

Applications usually do not have to call these commands directly.

ExistingClient

All non-error states	Connection must not be associated with a ClientID

This command identifies a connection as belonging to an existing client. There must be at least one other connection from this client still open. The **`ExistingClient`** command is called automatically by the generic stub function **`NFastApp_Connect`** as appropriate, for example when making an additional connection to an existing client.

Arguments

```
struct M_Cmd_ExistingClient_Args {
    M_Cmd_ExistingClient_Args_flags flags;
    M_ClientID client;
};
```

- No **`flags`** are defined.
- **`M_ClientID client`**: R_{SC}

Reply

```
struct M_Cmd_ExistingClient_Reply {
    M_Cmd_ExistingClient_Reply_flags flags;
};
```

No **`flags`** are defined.

NewClient

Initialization state, operational state	Connection must not be associated with a ClientID

This command asks the module for a random number to use as the **`ClientID`** for a new connection. It is called automatically by the generic stub function **`NFastApp_Connect`**.

Arguments

```
typedef struct M_Cmd_NewClient_Args {
   M_Cmd_NewClient_Args_flags flags;
};
```

No **`flags`** are defined.

Reply

```
struct M_Cmd_NewClient_Reply {
    M_Cmd_NewClient_Reply_flags flags;
    M_ClientID client;
};
```

- No **`flags`** are defined.
- **`M_ClientID client`**: $\mathbf{R}_{SC}$

Glossary

Authorized Card List

Controls the use of Remote Administration cards. If the serial number of a card does not appear in the Authorized Card List, it is not recognized by the system and cannot be used. The list only applies to Remote Administration cards.

Access Control List (ACL)

An Access Control List is a set of information contained within a key that specifies what operations can be performed with the associated key object and what authorization is required to perform each of those operations.

Administrator Card Set (ACS)

Part of the Security World architecture, an Administrator Card Set (ACS) is a set of smart cards used to control access to Security World configuration, as well as recovery and replacement operations.

The Administrator Cards containing share in the logical tokens that protect the Security World keys, including $\mathbf{K}_{NSO}$, the key-recovery key, and the recovery authorization keys. Each card contains one share from each token. The ACS is created using the well-known module key so that it can be loaded onto any nShield module.

See also *Security World*, *Operator Card Set (OCS)*

Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a block cipher adopted as an encryption standard by the US government and officially documented as US FIPS PUB 197 (FIPS 197). Originally only used for non-classified data, AES was also approved for use with for classified data in June 2003. Like its predecessor, the Data Encryption Standard (DES), AES has been analyzed extensively and is now widely used around the world.

Although AES is often referred to as *Rijndael* (the cipher having been submitted to the AES selection process under that name by its developers, Joan Daemen and Vincent Rijmen), these are not precisely the same cipher. Technically, Rijndael supports a larger range of block and key sizes (at any multiple of 32 bits, to a minimum of 128 bits and a maximum of 256 bits); AES has a fixed block size of 128 bits and only supports key sizes of 128, 192, or 256 bits.

See also *Data Encryption Standard (DES)* on page 216

CAST

CAST is a symmetric encryption algorithm with a 64-bit block size and a key size of between 40 bits to 128 bits (but only in 8-bit increments).

client identifier: R_{SC}

This notation represents an arbitrary number used to identify a client. In the nCore API, all client identifiers are 20 bytes long.

Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a symmetric cipher approved by NIST for use with US Government messages that are Secure but not Classified. The implementation of DES used in the module has been validated by NIST. DES uses a 64-bit block and a 56-bit key. DES keys are padded to 64 bits with 8 parity bits.

See also *Triple DES* on page 221, *Advanced Encryption Standard (AES)* on page 215

Diffie-Hellman

The Diffie-Hellman algorithm was the first commercially published public key algorithm. The Diffie-Hellman algorithm can only be used for key exchange.

Digital Signature Algorithm (DSA)

Also known as the Digital Signature Standard (DSS), the Digital Signature Algorithm (DSA) is a digital signature mechanism approved by NIST for use with US Government messages that are Secure but not Classified. The implementation of the DSA used by nShield modules has been validated by NIST as complying with FIPS 186.

Digital Signature Standard (DSS)

See *Digital Signature Algorithm (DSA)* on page 216

ECDH

A variant of the Diffie-Hellman anonymous key agreement protocol which uses elliptic curve cryptography.

See also *Diffie-Hellman* on page 216.

ECDSA

Elliptic Curve DSA: a variant of the Digital Signature Algorithm (DSA) which uses elliptic curve cryptography.

See also *Digital Signature Algorithm (DSA)* on page 216, *Diffie-Hellman* on page 216.

encryption: $\{A\}_B$

This notation indicates the result of **A** encrypted with key **B**.

Federal Information Processing Standards (FIPS)

The Federal Information Processing Standards (FIPS) were developed by the United States federal government for use by non-military government agencies and government contractors. FIPS 140 is a

series of publications intended to coordinate the requirements and standards for cryptographic security modules, including both their hardware and software components.

All Security Worlds are compliant with FIPS 140-2. By default, Security Worlds are created to comply with FIPS 140-2 at level 2, but those customers who have a regulatory requirement for compliance with FIPS 140-2 at level 3 can also choose to create a Security World that meets those requirements.

For more details about FIPS 140-2, see http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

hardserver

The hardserver software controls communication between applications and nShield modules, which may be installed locally or remotely. It runs as a service on the host computer. The behavior of the hardserver is controlled by the settings in the hardserver configuration file.

The hardserver software controls communication between the internal hardware security module and applications on the network. The module hardserver is configured using the front panel on the module or by means of uploaded configuration data. Configuration data is stored on the module and in files in a specially configured file system on each client computer.

hardware security module (HSM)

A hardware security module (commonly referred to as an HSM) is a hardware device used to hold cryptographic keys and software securely.

hash: H(X)

This notation indicates a fixed length result that can be obtained from a variable length input and that can be used to identify the input without revealing any other information about it. The nShield module uses the Secure Hash Algorithm (SHA-1) for its internal security.

identifier hash: $H_{ID}(X)$

An identifier hash is a hash that uniquely identifies a given object (for example, a key) without revealing the data within that object. The module calculates the identity hash of an object by hashing together the object type and the key material. The identity hash has the following properties:

H_{ID} is not modified by any operations on the key (for example, altering the ACL, the application data field, or other modes and flags)

H_{ID} is the same for both public and private halves of a key pair.

Unique data is added to the hash so that a H_{ID} is most unlikely to be the same as any other hash value that might be derived from the key material.

key blob

A key blob is a key object with its ACL and application data encrypted by a module key, a logical token, or a recovery key. Key blobs are used for the long-term storage of keys. Blobs are

cryptographically secure; they can be stored on the host computer's hard disk and are only readable by units that have access to the same module key.

See also *Access Control List (ACL)*.

key object: K_A

This is a key object to be kept securely by the module. A key object may be a private key, a public counterpart to a private key, a key for a symmetric cipher (MAC or some other symmetric algorithm), or an arbitrary block of data. Applications can use this last type to allow the module to protect any other data items in the same way that it protects cryptographic keys. Each key object is stored with an ACL and a 20-byte data block that the application can use to hold any relevant information.

KeyID: ID_{KA}

When a key object KA is loaded within the module's RAM, it is given a short identifier or handle that is notated as $\mathbf{ID}_{KA}$. This is a transient identifier, not to be confused with the key hash **HID(KA)**.

logical token: K_T

A logical token is a key used to protect key blobs. A logical token is generated on the nShield module and never revealed, except as shares.

MAC: MAC_{KC}

This notation indicates a MAC (Message Authentication Code) created using key **KC**.

module

See *hardware security module (HSM)*.

module key: K_M

A module key is a cryptographic key generated by each nShield module at the time of initialization and stored within the module. It is used to wrap key blobs and key fragments for tokens. Module keys can be shared across several modules to create a larger Security World.

All modules include two module keys:

- module key zero $\mathbf{K}_{M0}$, a module key generated when the module is initialized and never revealed outside the module.
- **null**, or well-known module key $\mathbf{K}_{MWK}$.

You can program extra module keys into a module.

See also: *Security World*, *hardware security module (HSM)*.

module signing key: K_{ML}

The module signing key is the module's public key. It is used to issue certificates signed by the module. Each module generates its own unique $\mathbf{K}_{ML}$ and $\mathbf{K}_{ML}^{-1}$ values when it is initialized. The private half of this key pair, $\mathbf{K}_{ML}^{-1}$, is never revealed outside the module.

nShield master feature enable key K_{SA}

Certain features of the module firmware are available as options. These features must be purchased separately from Thales. To use a feature on a specific module, you require a certificate from Thales signed by K_{SA}. These certificates include the electronic serial number for the module.

nShield Remote Administration Card

Smart cards that are capable of negotiating cryptographically secure connections with an HSM, using warrants as the root of trust. nShield Remote Administration Cards can also be used in the local slot of an HSM if required. You must use nShield Remote Administration Cards with Remote Administration.

nShield Security Officer's key: K_{NSO}^{-1}

The notation K_{NSO}^{-1} indicates the Security Officer's signing key. This key is usually a key to a public-key signature algorithm.

nShield Trusted Verification Device

A smart card reader that allows the card holder to securely confirm the Electronic Serial Number (ESN) of the HSM to which they want to connect, using the display of the device. Thales supplies and the nShield Trusted Verification Device and recommends its use with Remote Administration.

null module key: K_{MWK}

The null module key is used to create tokens that can be loaded onto any module. Such tokens are required for recovery schemes. The null module key is a Triple DES key of a value 01010101. As this value is well known, this module key does not have any security. Key blobs cannot be made directly under the null module key. To make a blob under a token protected by the null module key, the key must have the ACL entry **AllowNullKMToken**.

Operator Card Set (OCS)

Part of the Security World architecture, an Operator Card Set (OCS) is a set of smart cards containing shares of the logical tokens that is used to control access to application keys within a Security World. OCSs are protected using the Security World key, and therefore they cannot be used outside the Security World.

See also: *Security World*, *Administrator Card Set (ACS)*.

Recovery key: K_{RA}

The recovery key is the public key of the key recovery agent.

Remote Access Client, Server and solution

The remote access solution, such as SSH or a remote desktop application, which is used as standard by your organization. Enables you to to carry out Security World administrative tasks from a different location to that of an nShield Connect or nShield Solo.

For example, the remote access solution is used to run security world utilities remotely and to enter passphrases.

Note: Thales does not provide this software.

Remote Administration

An optional Security World feature that enables Remote Administration card holders to present their cards to an HSM located elsewhere. For example, the card holder may be in an office, while the HSM is in a data center. Remote Administration supports the ACS, as well as persistent and non-persistent OCS cards, and allows all smart card operations to be carried out, apart from loading feature certificates.

nShield Remote Administration Client

A GUI or command-line interface that enables you to select an HSM located elsewhere from a list provided by the Remote Administration Service, and associate a card reader attached to your computer with the HSM. Resides on your local Windows or Linux-based computer.

Remote Administration Service

Enables secure communications between an nShield Remote Administration Card and the hardserver that is connected to the appropriate HSM. Listens for incoming connection requests from nShield Remote Administration Clients. Supplies a list of available HSMs to the nShield Remote Administration Client and maintains an association between the relevant card reader and the HSM.

Dynamic Slot

Virtual card slots that can be associated with a card reader connected to a remote computer. Remote Administration Slots are in addition to the local slot of an HSM and any soft card slot that may be available. HSMs have to be configured to support between zero (default) and 16 Remote Administration Slots.

Rijndael

See *Advanced Encryption Standard (AES)* on page 215

salt: X

The random value, or salt, is used in some commands to discourage brute force searching for keys.

Security World

The Security World technology provides an infrastructure for secure lifecycle management of keys. A Security World consists of at least one hardware security module, some cryptographic key and certificate data encrypted by a Security World key and stored on at least one host computer, a set of Administrator Cards used to control access to Security World configuration, recovery and replacement operations, and optionally one or more sets of Operator Cards used to control access to application keys.

See also *Administrator Card Set (ACS)*, *Operator Card Set (OCS)*.

Security World key: K_{MSW}

The Security World key is the module key that is present on all modules in a Security World. Each Security World has a unique Security World key. This key is generated randomly when the Security World is created, and it is stored as a key blob protected by the ACS.

share: K_{Ti}

The notation K_{Ti} indicates a share of a logical token. Shares can be stored on smart cards or software tokens. Each share is encrypted under a separate share key.

share key: K_{Si}

A share key is a key used to protect an individual share in a token. Share keys are created from a Security World key, a pass phrase, and a salt value.

Standard nShield Cards

Smart cards used in the local slot of an HSM. Standard nShield cards are not supported for use with Remote Administration.

Standard card reader

A smart card reader for ISO/IEC 7816 compliant smart cards. Thales recommends that standard smart card readers are only used with the nShield Remote Administration Client command-line utility, not the GUI.

Triple DES

Triple DES is a highly secure variant of the Data Encryption Standard (DES) algorithm in which the message is encrypted three times.

See also *Data Encryption Standard (DES)* on page 216, *Advanced Encryption Standard (AES)* on page 215.

Internet addresses

Web site:	http://www.thales-esecurity.com/
Support:	http://www.thales-esecurity.com/support-landing-page
Online documentation:	http://www.thales-esecurity.com/knowledge-base
International sales offices:	http://www.thales-esecurity.com/contact

Addresses and contact information for the main Thales e-Security sales offices are provided at the bottom of the following page.

THALES

www.thales-esecurity.com

About Thales e-Security

Thales e-Security is a leading global provider of trusted cryptographic solutions with a 40-year track record of protecting the world's most sensitive applications and information. Thales solutions enhance privacy, trusted identities, and secure payments with certified, high performance encryption and digital signature technology for customers in a wide range of markets including financial services, high technology, manufacturing, and government. Thales e-Security has a worldwide support capability, with regional headquarters in the United States, the United Kingdom, and Hong Kong. www.thales-esecurity.com

Follow us on:

Americas – Thales e-Security Inc. 900 South Pine Island Road, Suite 710, Plantation, FL 33324 USA • Tel: +1 888 744 4976 or +1 954 888 6200 • Fax: +1 954 888 6211 • E-mail: sales@thalesesec.com
Asia Pacific – Unit 4101 41/F 248, Queen's Road East, Wanchai, Hong Kong, PRC • Tel: +852 2815 8633 • Fax: +852 2815 8141 • E-mail: asia.sales@thales-esecurity.com
Europe, Middle East, Africa – Meadow View House, Long Crendon, Aylesbury, Buckinghamshire HP18 9EQ • Tel: +44 (0)1844 201800 • Fax: +44 (0)1844 208550 • E-mail: emea.sales@thales-esecurity.com