

nShield Security World

nCore v13.9.6 Developer Tutorial

03 September 2026

© Entrust Corporation. All rights reserved.

Table of Contents

1. Read this guide if ... 2
2. Further information 3
3. Security advisories 4
    3.1. Contacting Entrust nShield Support 4
4. nCore architecture 5
    4.1. Programming environment architecture 5
    4.2. Generating a key 5
    4.3. Loading a key 6
    4.4. Transacting a command 7
5. C tutorial 9
    5.1. Overview 9
        5.1.1. nCore API functionality used in this tutorial 10
        5.1.2. Variables used in this tutorial 11
    5.2. Before connecting to the hardserver 12
        5.2.1. Declaring a call context 12
        5.2.2. Declaring memory allocation upcalls 12
        5.2.3. Declaring threading upcalls 13
        5.2.4. Initializing the nFast application handle 13
    5.3. Connecting to the hardserver 14
        5.3.1. Getting Security World information 14
        5.3.2. Setting up the authorization mechanism 14
    5.4. Generating a symmetric key 16
        5.4.1. Obtaining authorization and selecting a module 17
        5.4.2. Preparing the key-generation command and ACL 18
        5.4.3. Freeing memory 21
    5.5. Generating an asymmetric key 22
        5.5.1. Obtaining authorization and selecting a module 24
        5.5.2. Preparing the key-generation command and ACL 25
        5.5.3. Freeing memory 29
    5.6. Using a key 30
        5.6.1. Finding a key 30
        5.6.2. Loading a key 31
    5.7. Encrypting a file 32
    5.8. Cleaning up resources 35
6. Java tutorial 36
    6.1. Overview 36
        6.1.1. Creating a softcard 37

6.1.2. nCore classes used in this tutorial . . . 37
6.1.3. Variables used in this tutorial . . . 38
6.2. Before connecting to the hardserver . . . 39
6.3. Connecting to the hardserver . . . 39
6.4. Generating a key . . . 40
6.4.1. Methods used in generate_key() . . . 43
6.5. Using a key . . . 44
6.6. Signing a file . . . 44
6.7. Cleaning up resources . . . 47
7. Python 3 tutorial . . . 48
7.1. Prerequisites . . . 48
7.2. Set up the environment for nfpython . . . 48
7.3. Create and configure the virtualenv . . . 49
7.4. nfpython connections and commands . . . 50
7.5. Worked nfpython example for hash, sign, and verify . . . 51
8. Java examples . . . 56
8.1. Extract and compile the Java examples . . . 56
8.2. Java key management example utilities . . . 56
8.2.1. AppKeyGen.java . . . 57
8.2.2. GenerateExport.java . . . 57
8.2.3. KMJavaFloodTest.java . . . 57
8.2.4. NFKMInfo.java . . . 57
8.2.5. NVRamRTCUtil.java . . . 57
8.2.6. SimpleCrypt.java . . . 57
8.2.7. SlotPoller.java . . . 57
8.3. Java JCE/CSP example utilities . . . 58
8.3.1. AsymmetricEncryptionExample.java . . . 58
8.3.2. DK_ECDHKAExample.java . . . 58
8.3.3. ECDHExample.java . . . 58
8.3.4. ECIESExample.java . . . 58
8.3.5. EdDSAExample.java . . . 59
8.3.6. JCEChanTest.java . . . 59
8.3.7. JCEFloodTest.java . . . 59
8.3.8. JCEKDFExample.java . . . 59
8.3.9. JCESigTest.java . . . 59
8.3.10. MLDSAExample.java . . . 60
8.3.11. KeyLoadTimer.java . . . 60
8.3.12. KeyStorageExample.java . . . 61
8.3.13. NCipherLibraryInteropExample.java . . . 61

8.3.14. SignaturesExample.java . . . 61
8.3.15. SslClientExample.java . . . 61
8.3.16. SslServerExample.java . . . 61
8.3.17. SymmetricEncryptionExample.java . . . 61
8.3.18. SignatureTest.java . . . 62
8.4. Java generic stub examples . . . 62
8.4.1. BlobInfo.java . . . 62
8.4.2. Channel.java . . . 62
8.4.3. CheckMod.java . . . 62
8.4.4. CrypTest.java . . . 62
8.4.5. DesKat.java . . . 63
8.4.6. DKTest.java . . . 63
8.4.7. EasyConnection.java . . . 63
8.4.8. Enquiry.java . . . 63
8.4.9. FloodTest.java . . . 63
8.4.10. GenCert.java . . . 63
8.4.11. InitUnit.java . . . 64
8.4.12. NFEnum.java . . . 64
8.4.13. ReportVersion.java . . . 64
8.4.14. ScoreKeeper.java . . . 64
8.4.15. SigTest.java . . . 64
9. Key structures . . . 65
9.1. Mechanisms . . . 65
9.1.1. Mech_Any . . . 66
9.2. Key Types . . . 69
9.2.1. Random . . . 72
9.2.2. ArcFour . . . 72
9.2.3. Blowfish . . . 73
9.2.4. CAST . . . 74
9.2.5. CAST256 . . . 74
9.2.6. DES . . . 74
9.2.7. DES2 . . . 76
9.2.8. Triple DES . . . 77
9.2.9. Rijndael . . . 78
9.2.10. SEED . . . 79
9.2.11. Serpent . . . 80
9.2.12. Twofish . . . 81
9.2.13. Diffie-Hellman and ElGamal . . . 81
9.2.14. DSA . . . 84

9.2.15. ML-DSA . . . 89
9.2.16. SLH-DSA . . . 90
9.2.17. ML-KEM . . . 93
9.2.18. Elliptic Curve ECDH and ECDSA . . . 94
9.2.19. KCDSA . . . 96
9.2.20. RSA . . . 101
9.2.21. DeriveKey . . . 105
9.3. Hash functions . . . 112
9.3.1. SHA-1 . . . 112
9.3.2. Tiger . . . 112
9.3.3. SHA-224 . . . 113
9.3.4. SHA-256 . . . 113
9.3.5. SHA-384 . . . 113
9.3.6. SHA-512 . . . 114
9.3.7. MD2 . . . 114
9.3.8. MD5 . . . 114
9.3.9. RIPEMD 160 . . . 115
9.3.10. HAS160 . . . 115
9.4. HMAC signatures . . . 116
9.5. ACLs . . . 117
9.6. Use limits . . . 120
9.7. Actions . . . 123
9.8. Action types . . . 124
9.8.1. OpPermissions . . . 124
9.8.2. MakeBlob . . . 125
9.8.3. MakeArchiveBlob . . . 127
9.8.4. NSO . . . 128
9.8.5. NVRAM . . . 129
9.8.6. ReadShare . . . 130
9.8.7. SendShare . . . 130
9.8.8. FileCopy . . . 131
9.8.9. UserAction . . . 131
9.8.10. DeriveKey and DeriveKeyEx . . . 131
9.8.11. Using DeriveKey — an example . . . 133
9.9. Certificates . . . 142
9.9.1. Using a certificate to authorize an action . . . 143
9.9.2. Generating a certificate to authorize another operation . . . 144
10. NFKM Functions . . . 147
10.1. Debugging NFKM functions . . . 147

10.2. Functions . . . 147
10.2.1. NFKM_changepp . . . 147
10.2.2. NFKM_checkconsistency . . . 148
10.2.3. NFKM_checkpp . . . 148
10.2.4. NFKM_cmd_generaterandom . . . 149
10.2.5. NFKM_cmd_destroy . . . 149
10.2.6. NFKM_cmd_loadblob . . . 149
10.2.7. NFKM_cmd_getkeyplain . . . 150
10.2.8. NFKM_erasecard . . . 150
10.2.9. NFKM_erasemodule . . . 150
10.2.10. NFKM_hashpp . . . 151
10.2.11. NFKM_initworld_* . . . 151
10.2.12. NFKM_loadadminkeys_* . . . 154
10.2.13. NFKM_loadcardset_* . . . 159
10.2.14. NFKM_loadworld_* . . . 161
10.2.15. NFKM_makecardset_* . . . 163
10.2.16. NFKM_newkey_* . . . 167
10.2.17. NFKM_operatorcard_changepp . . . 172
10.2.18. NFKM_operatorcard_checkpp . . . 173
10.2.19. NFKM_recordkey . . . 173
10.2.20. NFKM_recordkeys . . . 173
10.2.21. NFKM_replaceacs_* . . . 174
11. OpenSSL with NFKM Engine . . . 178
11.1. Quick usage . . . 178
11.2. Testing with a self-signed certificate . . . 178
11.3. Private key operations . . . 179
11.3.1. Signing/verifying (DSA/ECDSA/RSA) . . . 179
11.3.2. Encryption/decryption (RSA) . . . 180
11.3.3. Key type and mechanism support . . . 180
11.4. Common problems . . . 182
11.4.1. invalid engine "nfkm" . . . 182
11.4.2. unable to load server certificate private key file . . . 182
12. nCore API commands . . . 183
12.1. Basic commands . . . 183
12.1.1. ClearUnit . . . 184
12.1.2. ClearUnitEx . . . 185
12.1.3. ModExp . . . 186
12.1.4. ModExpCrt . . . 186
12.2. Key-management commands . . . 187

12.2.1. ChangeSharePIN . . . 187
12.2.2. ChannelOpen . . . 188
12.2.3. ChannelUpdate . . . 191
12.2.4. Decrypt . . . 192
12.2.5. DeriveKey . . . 193
12.2.6. Destroy . . . 195
12.2.7. Duplicate . . . 196
12.2.8. Encrypt . . . 197
12.2.9. Export . . . 198
12.2.10. FirmwareAuthenticate . . . 198
12.2.11. FormatToken . . . 198
12.2.12. GenerateKey and GenerateKeyPair . . . 199
12.2.13. GenerateLogicalToken . . . 205
12.2.14. GetChallenge . . . 206
12.2.15. GetKML . . . 206
12.2.16. GetTicket . . . 207
12.2.17. Hash . . . 209
12.2.18. ImpathKXBegin . . . 210
12.2.19. ImpathKXFinish . . . 212
12.2.20. ImpathReceive . . . 213
12.2.21. ImpathSend . . . 213
12.2.22. InitialiseUnit . . . 214
12.2.23. LoadBlob . . . 215
12.2.24. LoadLogicalToken . . . 216
12.2.25. MakeBlob . . . 217
12.2.26. MergeKeyIDs . . . 220
12.2.27. ReadShare . . . 221
12.2.28. RedeemTicket . . . 223
12.2.29. RemoveKM . . . 224
12.2.30. RSAImmedSignDecrypt . . . 224
12.2.31. RSAImmedVerifyEncrypt . . . 225
12.2.32. SetACL . . . 226
12.2.33. SetKM . . . 228
12.2.34. SetNSOPerms . . . 228
12.2.35. SetRTC . . . 231
12.2.36. Sign . . . 232
12.2.37. SignModuleState . . . 233
12.2.38. StaticFeatureEnable . . . 235
12.2.39. UpdateMergedKey . . . 236

12.2.40. Verify . . . 237
12.2.41. WriteShare . . . 238
12.3. Commands used by the generic stub only . . . 239
12.3.1. ExistingClient . . . 239
12.3.2. NewClient . . . 240
13. Transaction IDs . . . 242
13.1. Introduction . . . 242
13.2. Limitations . . . 242
13.3. Unicode Notes . . . 243
13.4. Setting Transaction IDs . . . 243
13.4.1. nCore C (Generic Stub) . . . 243
13.4.2. SEElib (CodeSafe CSEE) . . . 244
13.4.3. nCore Python (nfpython) . . . 244
13.4.4. nCore Java (nfjava) . . . 244
13.4.5. Higher-level APIs . . . 245
13.5. Transaction ID logging . . . 245
13.5.1. Client debug logs . . . 245
13.5.2. nCore audit logs . . . 246

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 Entrust 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)
- Linux: `/opt/nfast/document/ncore/html/index.html` (C) and `/opt/nfast/java/docs/index.html` (Java).

1. 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, use the appropriate nShield API 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

2. Further information

This guide forms one part of the information and support provided by Entrust.

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`
- Linux:
  - 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.

3. Security advisories

If Entrust becomes aware of a security issue affecting nShield HSMs, Entrust will publish a security advisory to customers. The security advisory will describe the issue and provide rec ommended actions. In some circumstances the advisory may recommend you upgrade the nShield firmware and or image file. In this situation you will need to re-present a quorum of administrator smart cards to the HSM to reload a Security World. As such, deployment and maintenance of your HSMs should consider the procedures and actions required to upgrade devices in the field.



The Remote Administration feature supports remote firmware upgrade of nShield HSMs, and remote ACS card presentation.

We recommend that you monitor the Announcements & Security Notices section on Entrust nShield, https://trustedcare.entrust.com/, where any announcement of nShield Security Advisories will be made.

3.1. Contacting Entrust nShield Support

To obtain support for your product, contact Entrust nShield Support, https://trustedcare.entrust.com/.

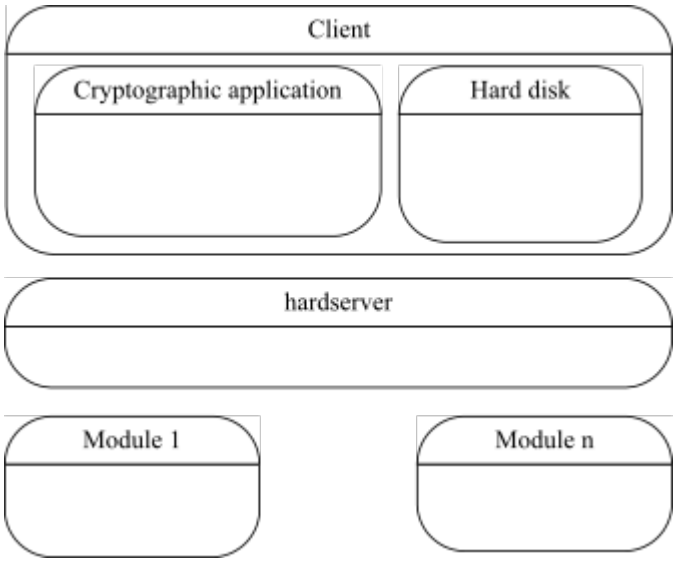
4. nCore architecture

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

4.1. Programming environment architecture

The following diagram illustrates typical architecture in which one would use the nCore API:



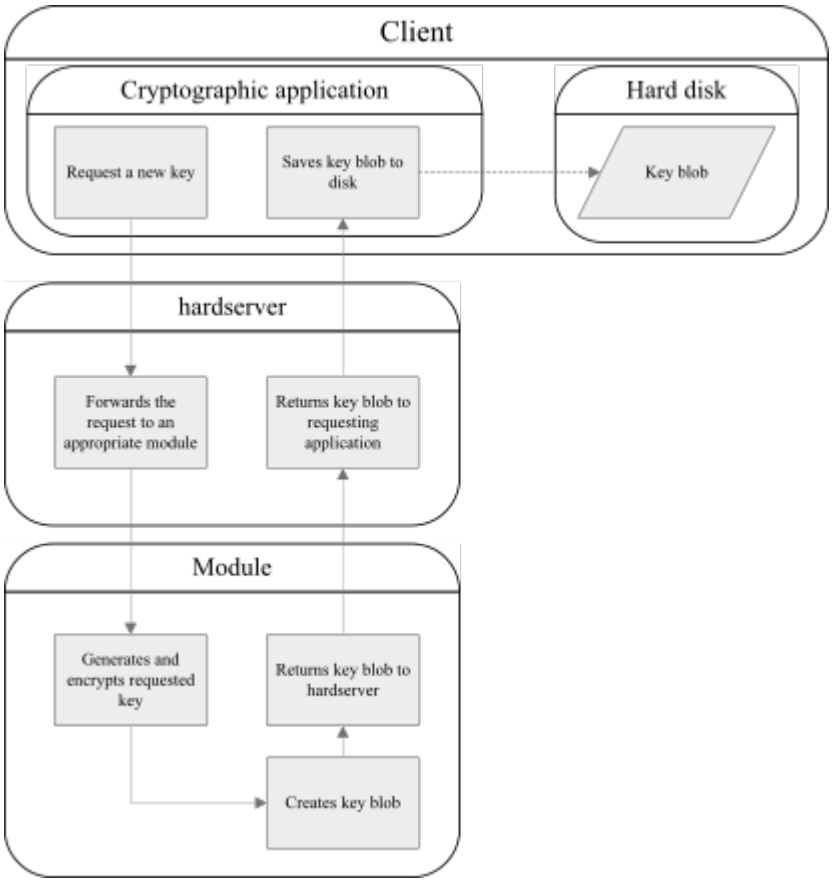
In the typical programming environment architecture diagram:

- **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.

4.2. 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.

The following diagram illustrates the interaction between your cryptographic application and the Security World that occurs during the key-generation process:



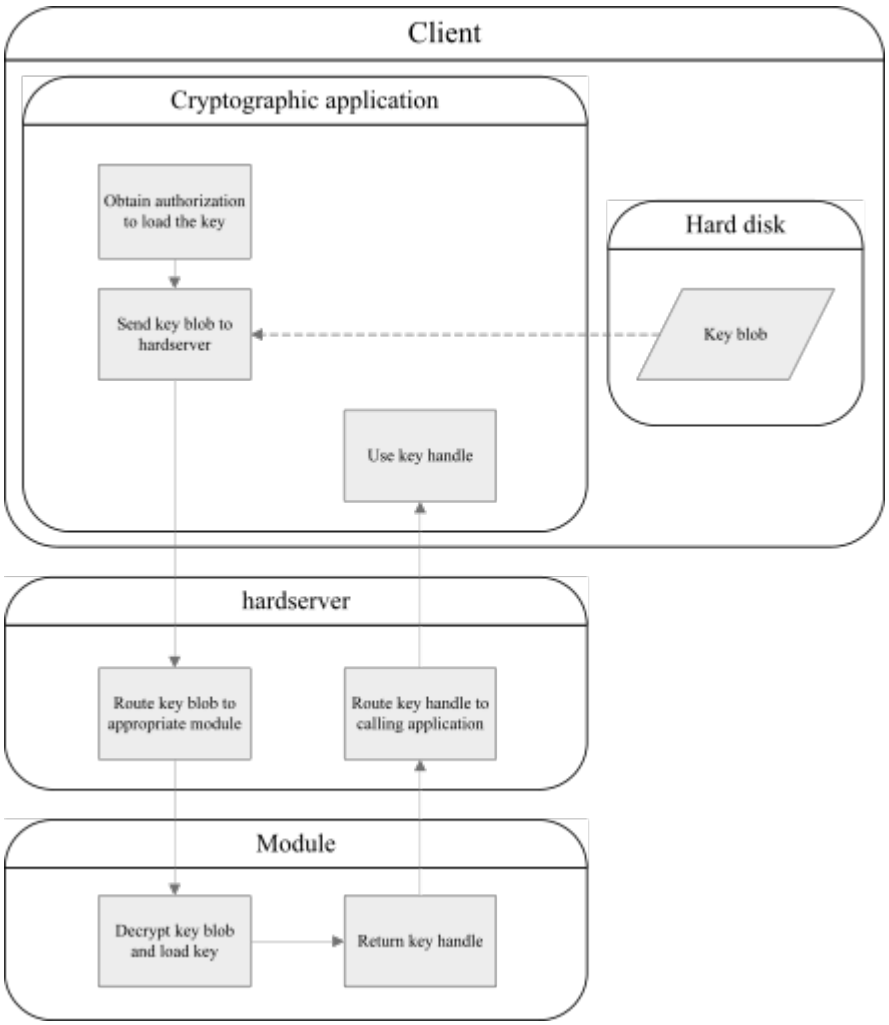
A key blob can only be decrypted by a module that has a record of the key that was used to encrypt the information in the key blob. A key blob contains key information and an Access Control List (ACL) which defines who can use the key and what operations the key can be used for.

4.3. 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. The key blob is decrypted using a key stored on the mod ule, and a handle or object reference to the key is returned to your application.

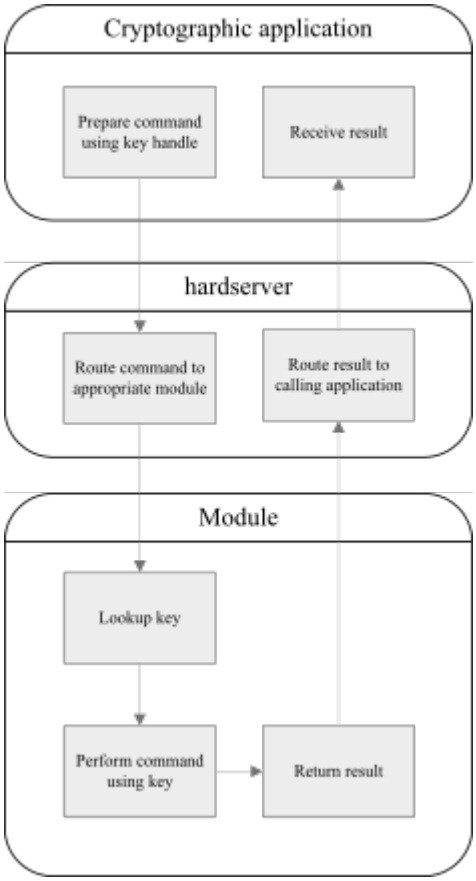
In most cases it is necessary to provide authentication in the form of a smart card and/or a passphrase before using a key. The user interaction that prompts for authentication to be provided is handled by the nCore API.

The following diagram illustrates the key loading process:



4.4. 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. The following diagram illustrates the process of transacting a command.



C tutorial explains how to write a C application that:

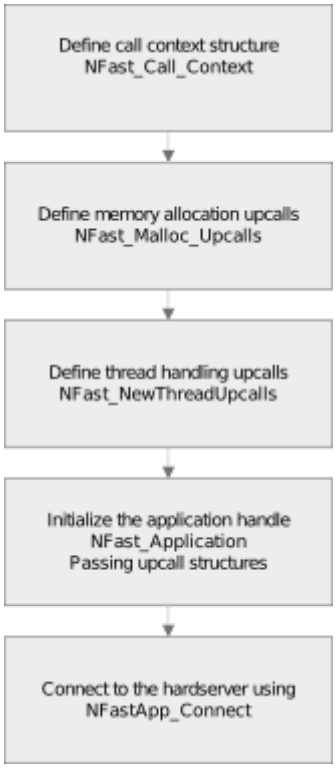
- creates a connection to the hardserver
- generates a key
- loads a key onto a module
- transacts a command with the module to use the key to encrypt a file. Java tutorial explains how to write a similar Java application which signs a file.

5. C tutorial

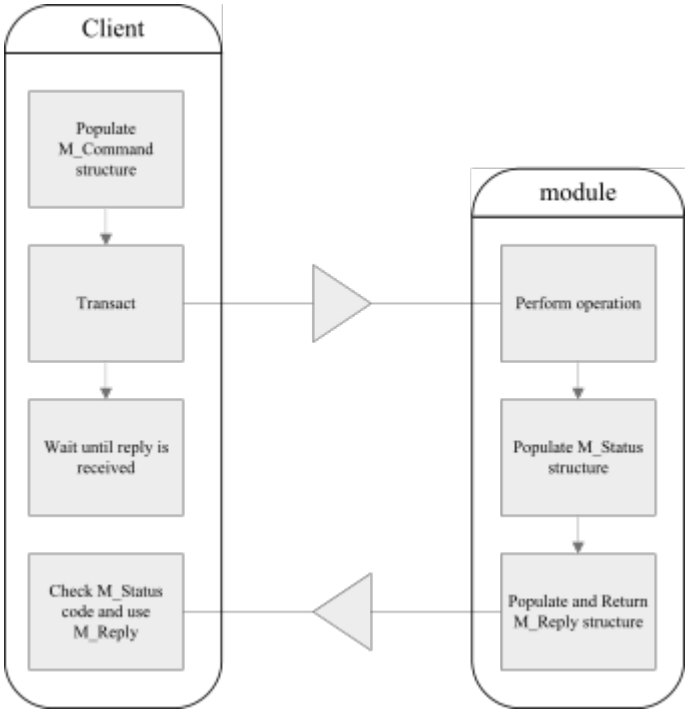
5.1. Overview

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. The following diagram illustrates the steps required to create a connection to a hardserver running on Entrust hardware:



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. The following diagram illustrates the process of transacting a cryptographic operation with the module:



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.

5.1.1. 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 Level 3 (Federal Information Processing Standards) environment.

5.1.2. 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

Variable Name	Variable Type	Description
`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 Level 3 environment

5.2. 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.

5.2.1. 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;
};
```

5.2.2. 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);
}
```

5.2.3. 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;
}
```

5.2.4. 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);
```

5.3. 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;
}
```

5.3.1. Getting Security World information

The following code reads in the Security World information that is associated with the appli cation 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;
}
```

5.3.2. Setting up the authorization mechanism

The nCore API supports three types of key protection:

- module protection
- passphrase protection
- card set protection.

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

5.3.2.1. 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;
```

5.3.2.2. Obtaining additional FIPS authorization

FIPS 140 Level 3 mode requires authorization for key-generation, which can be obtained from either an Operator Card or an Administrator Card. The following code initializes the FIPS 140 Level 3 code library, which seeks FIPS 140 Level 3 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;
```

5.3.2.3. 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;
}
```

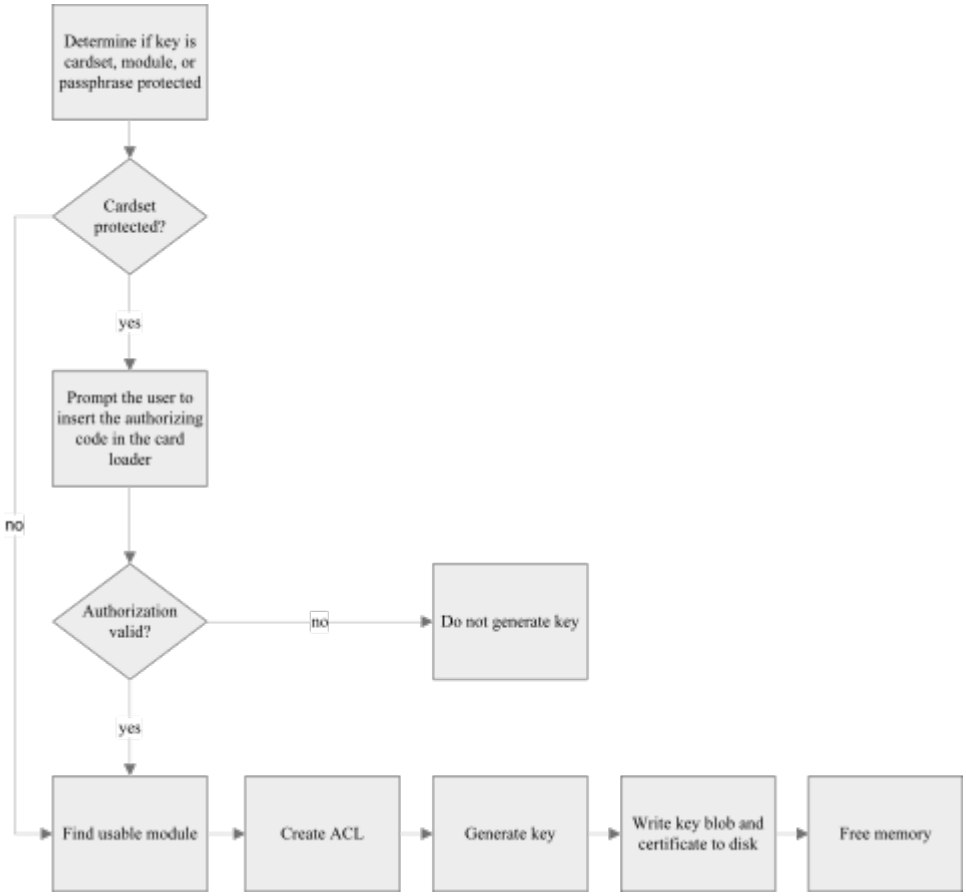
5.4. Generating a symmetric key

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.



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

The following diagram illustrates 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

Variable Name	Variable Type	Description
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

5.4.1. 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.

5.4.1.1. 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;
}
```

5.4.1.2. 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;
}
```



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.

5.4.2. 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

- passphrase 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.

```
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.

```
blob_params.f = acl_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 FIPS140 Level 3 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 FIPS 140 Level 3 authorization (or can celled the operation if the correct authorization could not be obtained), it can use the autho rization 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 nec essary 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 func tion 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;
```

5.4.3. 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 FIPS 140 Level 3 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.

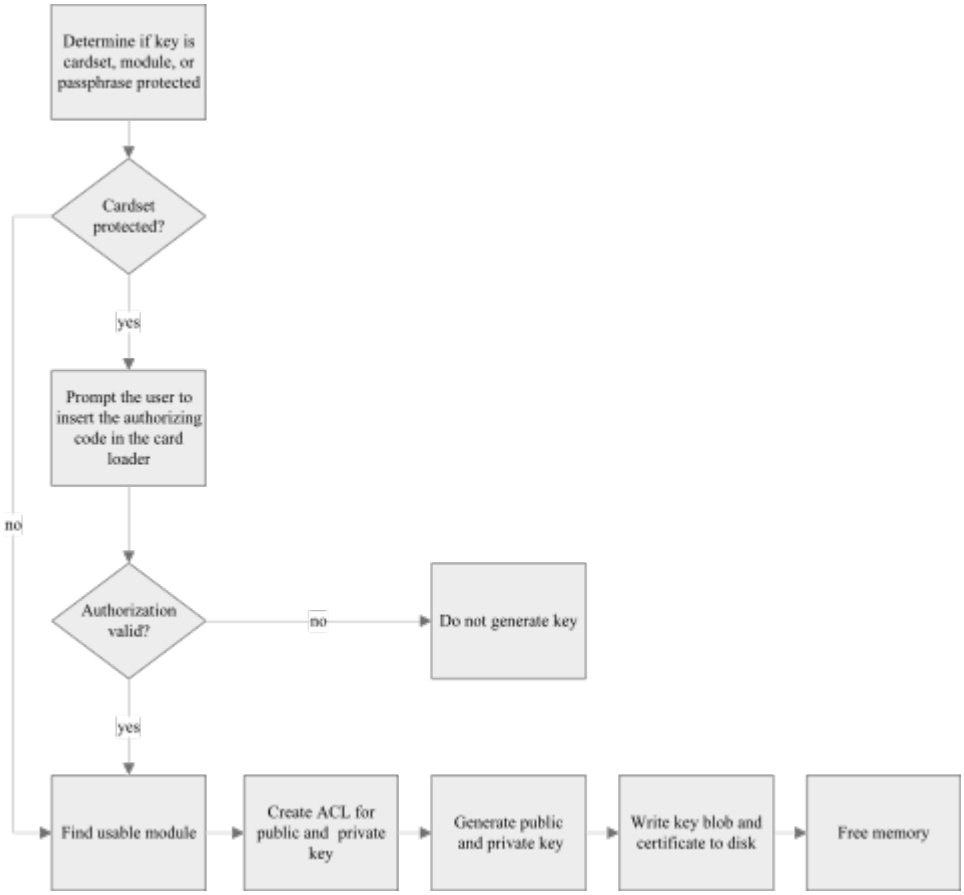
5.5. Generating an asymmetric key

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.

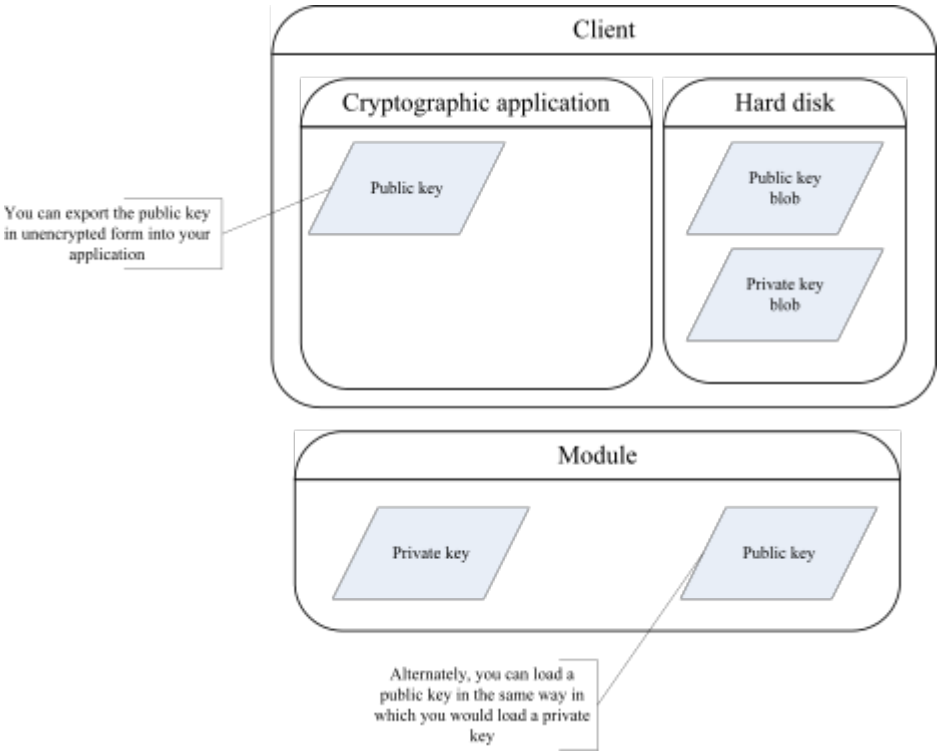


This section does not explain how to use softcards to protect keys. Softcards can be listed with `NFKM_listsoftcards()` and loaded with `NFKM_loadsoftcard()`. See the nCore API documentation of `nfkm.h` for more information about using softcards.

The following diagram illustrates the key-generation process:



The following diagram illustrates how the programming environment architecture stores generated asymmetric keys. See nCore architecture for more information about the programming environment architecture.



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

5.5.1. 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.

5.5.1.1. 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;
}
```

5.5.1.2. 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;
}
```



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.

5.5.2. 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 cre ate 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
- passphrase 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.

```
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_on-ly_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 fol lowing 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 FIPS 140 Level 3 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 FIPS 140 Level 3 authorization (or can

celled the operation if the correct authorization could not be obtained), it can use the autho rization 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 nec essary 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 func tion 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;
```

5.5.3. 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 FIPS 140 Level 3 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);

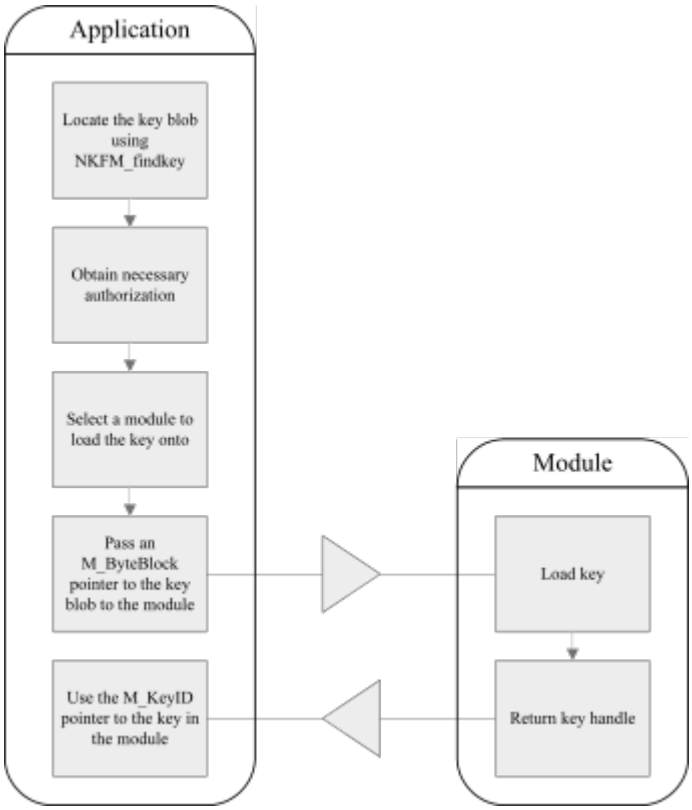
NFastApp_Free_Reply(app, cctx, 0, &reply);
keyinfo->appname = 0;
keyinfo->ident = 0;
NFKM_freekey(app, keyinfo, cctx);
NFKM_freecardset(app, cardset, cctx);
```

This concludes the explanation of asymmetric key-generation.

5.6. 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 illustrates the process of loading a key:



5.6.1. 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;
  }
}
```

5.6.2. Loading a key

Before a key can be loaded into a module, an application must obtain the appropriate autho rization. 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 pro- vide 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.



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;
}
```

5.7. 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 (`input-stream`) 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);
```

5.8. 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.

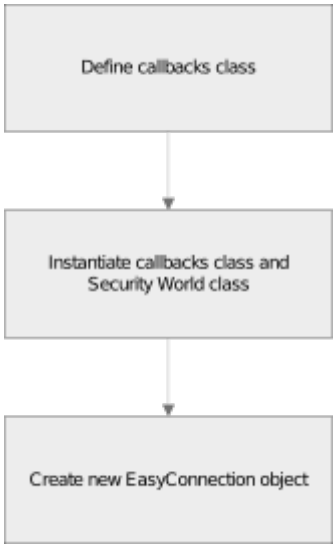
```
free(sigbytes.ptr);
if(keyid) NFKM_cmd_destroy(app, conn, 0, keyid, "keyid", cctx);
if(idch) NFKM_cmd_destroy(app, conn, 0, idch, "idch", cctx);
NFastApp_Free_Reply(app, cctx, 0, &reply);
if(rqcard_fips_initialized) RQCard_fips_free(&rqcard, &rqcard_fips);
if(rqcard_initialized) RQCard_destroy(&rqcard);
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);
```

6. Java tutorial

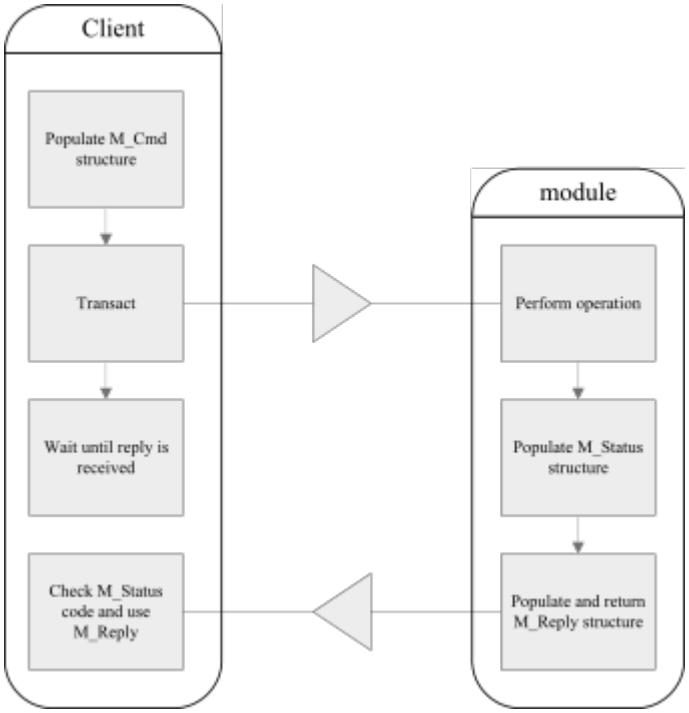
6.1. Overview

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. The following diagram illustrates the steps required to create a connection to a hardserver running on Entrust hardware:



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. The following diagram illustrates the process of transacting a cryptographic operation with a module:



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.

6.1.1. Creating a softcard

This tutorial demonstrates how to protect a key using a softcard. Use the command line util ity `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 passphrase. Type a passphrase and press **Enter**.

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

6.1.2. 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 Entrust.

- `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_dir>/java/examples/connutils`.

6.1.3. 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

Variable name	Variable type	Variable description
`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

6.2. 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 passphrase. 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 passphrase: ");
    } 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);
```

6.3. 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());
```

6.4. 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.

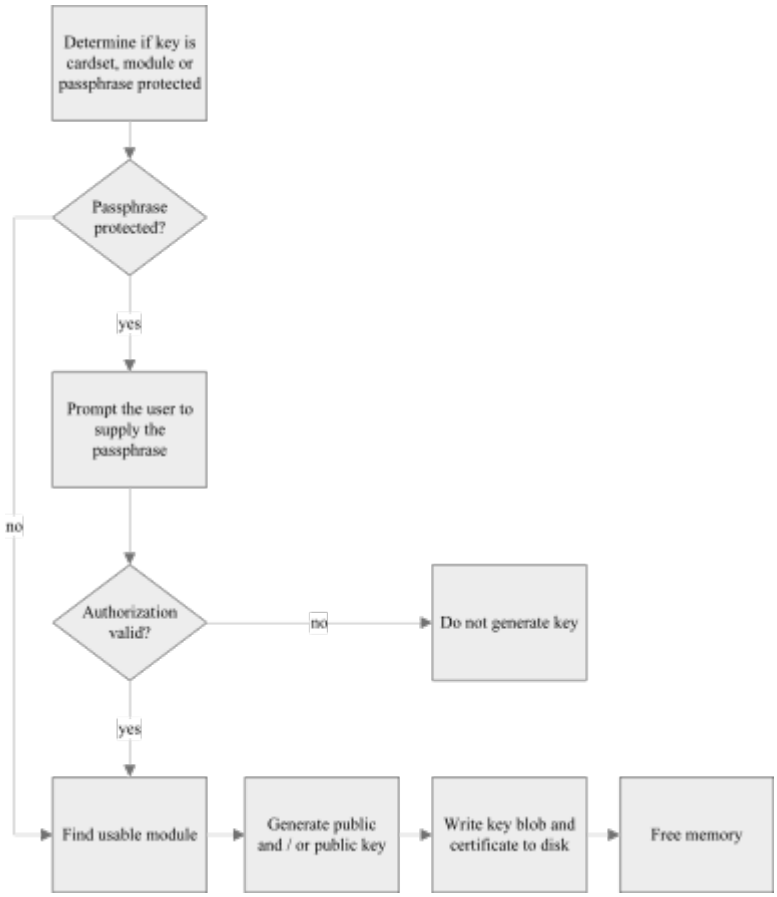
```
if(k == null) {
  if(softcard_name != "") {
    k = generate_key(wcb, world, type, size,
                     NFKM_Key_flags.f_ProtectionPassPhrase,
                     softcard_name,
                     appname, ident);
  } else {
     k = generate_key(wcb, world, type, size,
                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

ters 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. The following diagram illustrates the process of generating a key:





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.

Parameter name	Parameter Type	Parameter description
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.
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();
```

6.4.1. 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);
}
```

6.5. 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);
}
```

6.6. 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` com-

mand 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 `FileOutput-Stream` 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();
```

6.7. Cleaning up resources

Finally, unload the keys in the module memory.

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

7. Python 3 tutorial

7.1. Prerequisites

Operating systems

- Linux on x86_64
- Windows on x86_64

nShield software

- Security World 13.6 or later.

User permissions

- A user permitted to connect to the local hardserver and read the Security World key management data (kmdata) files.

Supported Python version

- Python 3.11

7.2. Set up the environment for nfpython



To use nShield Python 3 support with another version of Python 3, contact Entrust Support. Other Python versions are not covered by this guide.

The recommended way of developing and deploying an nShield Python 3 application is using the Python virtualenv created from the nShield Python 3 containing the packages your project requires.

Before you begin, install the optional "CipherTools" (Windows) or "ctd" (Linux) component as this contains the Python wheel files you require.

Entrust recommends the following directory layout:

```
application
  venv
  mypackage
    __init__.py
    code.py
  tests
    test_mypackage.py
```

script.py
setup.py

7.3. Create and configure the virtualenv

You do not need administrator access to create a virtualenv. When launching production applications, you must use the virtualenv that was created for them.

1. Start a command-line shell.
2. Change to the directory where you want to store your application files.
3. Run:

   Linux

   ```
   /opt/nfast/python3/bin/python3 -m venv venv
   ```

   Windows

   ```
   c:\Program Files\nCipher\nfast\python3\python --copies -m venv venv
   ```

4. Install all required Python packages into the virtualenv, for example:

   Linux

   ```
   . venv/bin/activate
   pip install /opt/nfast/python3/additional-packages/nfpython*.whl
   ```

   Windows (PowerShell)

   ```
   venv\Scripts\activate.ps1
   pip install c:\Program Files\nCipher\nfast\python3\additional-packages\nfpython*.whl
   ```



Entrust recommendeds that you use a `requirements.txt` file or `setup.py`/`setup.cfg` to define your dependencies, including the nfpython package.

To run your application:

- If your application uses `setup.py` entrypoint scripts, execute them directly.
- If you do not use entrypoints, you will need create your own scripts:



These examples assume that your program is called `application.py` and your virtualenv is in the `venv` directory.

Linux - application.sh

```
#!/bin/sh
HERE=$(dirname $(readlink -f $0))
. ${HERE}/venv/bin/activate
python3 ${HERE}/application.py
```

Windows - application.ps1

```
$PSScriptRoot\venv\Scripts\activate.ps1
python $PSScriptRoot\application.py
```

7.4. nfpython connections and commands

Send nCore commands to the hardserver or attached HSMs:

1. Import the nShield python module.

```
import nfpython
```

2. Set up a connection

```
conn = nfpython.connection()
```

3. Construct an nCore command:

```
c = nfpython.Command()
c.cmd = "NewEnquiry"
c.args.module = 1
c.args.version = 1
```

The easiest way to define a new command to send is to create a `Command()` object, set the attributes starting with the `cmd` name, then set the `args` attributes as required.

4. Send the command, wait for the reply, and then print it:

```
reply = conn.transact(c)
print(reply.reply.data.one)
```

This command prints a reply similar to the following output.

```
EnquiryDataOne.releasemajor= 13
              .releaseminor= 6
              .releasepatch= 5
              .checkintimehigh= 0
              .checkintimelow= 1620223158
              .flags= Hardware|HasTokens|SupportsCommandState
```

```
            .speedindex= 14200
            .recommendedminq= 9
            .recommendedmaxq= 152
            .hardwareserial= ABCD-8287-C172
            .softwaredetails= 13.6.5-120-a44e176921
```

If the command results in a reply with a status value other than `OK`, the `transact()` connection method raises an `NFStatusError` exception.

To suppress the exception and obtain the original reply regardless of status, use the `ignorestatus=True` keyword argument:

```
import nfpython
conn = nfpython.connection()
c = nfpython.Command()
c.cmd = "NewEnquiry"
c.args.module = 1000
reply = conn.transact(c, ignorestatus=True)
print(reply)
```

This prints:

```
Reply.cmd= ErrorReturn
     .status= InvalidModule
     .flags= 0x0
```

7.5. Worked nfpython example for hash, sign, and verify

nShield Security World software includes several Python 3 example files.

The default location for these files is:

- Linux: `/opt/nfast/python3/examples`
- Windows: `C:\Program Files\nCipher\nfast\python3\examples`

The files `hashing.py`, `keys.py` and `signing.py` make up a sample application that signs data using a previously generated RSA key. The application performs the following steps: load keys, digest data, and perform a sign and verify operation using an attached HSM.

1. Generate a module-protected RSA key:

   Linux

   ```
   /opt/nfast/bin/generatekey -b simple protect=module type=RSA size=2048 ident=signer
   ```

   Windows

   ```
   c:\Program Files\nCipher\nfast\bin\generatekey -b simple protect=module type=RSA size=2048 ident=signer
   ```

2. Find and load the keys

   Finding saved keys requires the `nfpython` and `nfkm` modules.

   To load an existing key, you need to know the `appname` and `ident` of the saved key, and you need the `nfpython` connection.

```
import nfpython
import nfkm

appname = "simple"
ident = "signer"
module = 1

conn = nfpython.connection(needworldinfo=True)
appident = nfkm.KeyIdent(appname=appname, ident=ident)
keydata = nfkm.findkey(conn, appident)

# load the private key from keydata.privblob
# or public key from keydata.pubblob
cmd = nfpython.Command(cmd="LoadBlob")
cmd.args.blob = keydata.privblob
cmd.args.module = module

# load the blob and get a Key ID
rep = conn.transact(cmd)
keyid = rep.reply.idka
```

3. Process the data.

   For large amounts of data, nShield software provides channels to perform crypto operations in an incremental stream.

   This example uses the `ChannelOpen` and `ChannelUpdate` commands to compute a SHA256 hash with the HSM.

   Entrust typically recommends a `ChannelUpdate` size of around 8000 bytes. However, you might find that larger or smaller sizes give better results depending on network conditions or HSM speed ratings.

```
message = b"hello world" * 10240
chunksize = 8000

conn = nfpython.connection(needworldinfo=True)
c = nfpython.Command()
c.cmd = "ChannelOpen"
c.args.type = "simple"
c.args.mode = "sign"
c.args.mech = "SHA256Hash"

rep = conn.transact(c)
channel = rep.reply.idch

c = nfpython.Command()
c.cmd = "ChannelUpdate"
# split the message up into small chunks and transmit each in sequence
for chunk in (message[i:i+chunksize] for i in range(0, len(message), chunksize)):
```

```
    c.args.idch = channel
    c.args.input = nfpython.ByteBlock(chunk, fromraw=True)
    conn.transact(c)

# obtain the hash value by setting the final flag
c.args.input = nfpython.ByteBlock()
c.args.flags |= "final"
rep = conn.transact(c)
digest = rep.reply.output
```

4. Sign the digest using the loaded private key:

```
# compute the hash (either using nfpython or hashlib)
digest = digest_message(conn, message)

# perform a signature
plain = nfpython.Hash32(digest)
c = nfpython.Command()
c.cmd = "Sign"
c.args.mech = "RSAhSHA256pPKCS1"
c.args.key = privkey
c.args.plain.type = "Hash32"
c.args.plain.data.data = plain

rep = conn.transact(c)
signature = rep.reply.sig
```

5. Verify the signature.

   Verification requires a public key, the signature plaintext, and the signature data.



If the signature is invalid, `transact(cmd)` raises an `NFStatusError` exception with its status attribute set to `"VerifyFailed"`.

```
digest = nfpython.Hash32(hashbytes)

c = nfpython.Command()
c.cmd = "Verify"
c.args.key = pubkey
c.args.plain.type = "Hash32"
c.args.plain.data.data = digest
c.args.sig = signature

conn.transact(c)
```

The full example would use the following code:

keys.py

```
import nfpython
import nfkm


def load_key(conn, appname: str, ident: str, module=0, private=True) -> nfpython.KeyID:
    """
    Load a key given an appname and ident
    :param conn:
```

```
    :param appname: Key appname, eg "simple"
    :param ident: Key ident
    :param module: module to load the key on (default 0 = any)
    :param private: load the private blob if true
    :return: the loaded key
    """
    appident = nfkm.KeyIdent(appname=appname, ident=ident)
    keydata = nfkm.findkey(conn, appident)

    cmd = nfpython.Command(cmd="LoadBlob")
    if private:
        cmd.args.blob = keydata.privblob
    else:
        cmd.args.blob = keydata.pubblob
    cmd.args.module = module

    rep = conn.transact(cmd)
    keyid = rep.reply.idka
    return keyid
```

hashing.py

```
import nfpython


def digest_message(conn, message: bytes, module=0, mech="Sha256Hash", chunksize=8000) -> bytes:
    """
    Hash a binary string using the HSM and ChannelUpdate commands
    :param conn:
    :param message: binary message to hash.
    :param module: HSM to sign with (default 0 = any)
    :param mech: hash mechanism name
    :param chunksize: digest block size
    :return:
    """
    c = nfpython.Command()
    c.cmd = "ChannelOpen"
    c.args.type = "simple"
    c.args.mode = "sign"
    c.args.mech = mech
    c.args.module = module

    rep = conn.transact(c)
    channel = rep.reply.idch

    c = nfpython.Command()
    c.cmd = "ChannelUpdate"
    # split the message up into small chunks and transmit each in sequence
    for chunk in (message[i:i+chunksize] for i in range(0, len(message), chunksize)):
        c.args.idch = channel
        c.args.input = nfpython.ByteBlock(chunk, fromraw=True)
        conn.transact(c)

    # obtain the hash value by setting the final flag
    c.args.input = nfpython.ByteBlock()
    c.args.flags |= "final"
    rep = conn.transact(c)
    digest = rep.reply.output
    return digest
```

signing.py

```
#!/usr/bin/env python3
```

```
import nfpython
from keys import load_key
from hashing import digest_message


def sign_message(conn, privkey: nfpython.KeyID, message: bytes) -> (nfpython.Hash32, nfpython.CipherText):
    """
    Hash and sign a binary string using the HSM and a loaded RSA private key
    :param conn:
    :param privkey: KeyID of loaded key
    :param message: bytes to sign
    :return: the digest hash and signature
    """
    digest = digest_message(conn, message)
    plain = nfpython.Hash32(digest)
    c = nfpython.Command()
    c.cmd = "Sign"
    c.args.mech = "RSAhSHA256pPKCS1"
    c.args.key = privkey
    c.args.plain.type = "Hash32"
    c.args.plain.data.data = plain

    rep = conn.transact(c)
    signature = rep.reply.sig
    return plain, signature


def verify_signature(conn, pubkey: nfpython.KeyID, digest, signature) -> bool:
    """
    Verify a signature using the HSM and a loaded public key
    :param conn:
    :param pubkey:
    :param digest:
    :param signature:
    :return:
    """
    cmd = nfpython.Command()
    cmd.cmd = "Verify"
    cmd.args.key = pubkey
    cmd.args.plain.type = "Hash32"
    cmd.args.plain.data.data = digest
    cmd.args.sig = signature

    conn.transact(cmd)


def run():
    conn = nfpython.connection(needworldinfo=True)
    privkey = load_key(conn, appname="simple", ident="signer")
    pubkey = load_key(conn, appname="simple", ident="signer", private=False)

    message_bytes = b"hello world" * 1024
    print(f"Hash and Sign {len(message_bytes)} bytes..")
    digest, signature = sign_message(conn, privkey, message_bytes)
    print("Verifying..")
    verify_signature(conn, pubkey, digest, signature)
    print("Done.")


if __name__ == "__main__":
    run()
```

8. 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` direc tory.

8.1. Extract and compile the Java examples

The Java example files are in subdirectories of the `%NFAST_HOME%\java\examples` (Windows) or `/opt/nfast/java/examples` (Linux) directory.

1. Extract the example files:

   ```
   jar xf <path-to-examples-jar-file>
   ```

   The JCE-related examples extract into the `com/ncipher/provider/examples` subtree.

2. Compile the examples:

   a. If using Java 8 or earlier (using "--class-path or -cp")

   ```
   javac -cp <fully-qualified-path-to-JCE-provider-jar-file> *.java
   ```

   For example:

   ```
   javac -cp /opt/nfast/java/classes/nCipherKM.jar *.java
   javac -cp /opt/nfast/java/classes/nCipherKM.jar com/ncipher/provider/examples/*.java
   ```

   b. If using Java 9 or later (using "--module-path or -p")

   ```
   javac -p <fully-qualified-path-to-JCE-provider-jar-file> --add-modules ALL-MODULE-PATH *.java
   ```

   For example:

   ```
   javac -p /opt/nfast/java/classes --add-modules ALL-MODULE-PATH *.java
   javac -p /opt/nfast/java/classes --add-modules ALL-MODULE-PATH com/ncipher/provider/examples/*.java
   ```

8.2. Java key management example utilities

8.2.1. AppKeyGen.java

This example utility demonstrates application key generation and import.

8.2.2. 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.

8.2.3. 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.

8.2.4. 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.

8.2.5. 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.

8.2.6. 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.

8.2.7. 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.

8.3. Java JCE/CSP example utilities

8.3.1. AsymmetricEncryptionExample.java

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

8.3.2. DK_ECDHKAExample.java

This example utility demonstrates:

- Creation of two ECDH key pairs.
- Key agreement using ECDHWITHSHA1KDF between two parties.
- Encryption/Decryption using the shared secret key.

8.3.3. ECDHExample.java

This example utility demonstrates:

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

8.3.4. ECIESExample.java

This example utility demonstrates:

- Creation of an ECDH key pair by the receiver.
- Key wrapping by the sender using the agreed ECIES parameters and the public half of receiver's ECDH key pair.
- Key unwrapping by the receiver using the agreed ECIES parameters.
- Encryption/Decryption using the shared secret key.

8.3.5. EdDSAExample.java

This example utility demonstrates how to generate and store key for use in Ed25519 and Ed25519ph operations.

The example generates an Ed25519 key pair, creates a KeyStore and stores both halves of the key pair.



This example may require `sudo` permissions on Linux machines.

8.3.6. 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.

8.3.7. JCEFloodTest.java

This example utility does performance testing for RSA, DSA, ECDSA and Ed25519 private key operations.

It demonstrates:

- RSA/DSA/ECDSA/Ed25519 Key Pair generation.
- RSA/DSA/ECDSA/EdDSA 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.

8.3.8. JCEKDFExample.java

This example utility demonstrates:

- Generation and saving of an HMAC key to be used with the KDF.
- Derivation of a key using the KDF.
- Implementation of a Known Answer Test for the KDF.

8.3.9. 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.

8.3.10. MLDSAExample.java

This example demonstrates the following:

- ML-DSA key pair generation.
- ML-DSA signing and verification.
- ML-DSA parameter set selection using MLDSAParameterSpec.
- Hedged ML-DSA signing using the standard ML-DSA signature algorithms.
- Deterministic signing option using the `-DET` signature variants.
- Pre-hash signature option using `SHA512withML-DSA`.
- Selection between parameter-set-specific signature names, such as ML-DSA-65, and parameter-set-independent signature names, such as ML-DSA.
- Optional ML-DSA context usage with MLDSAAlgorithmParameterSpec.

The example uses standard Java key interfaces, PrivateKey and PublicKey.

By default, the example generates an ML-DSA-65 key and uses the ML-DSA-65 signature algorithm.



Change the example constants and recompile to try different ML-DSA options.

MLDSAParameterSpec and MLDSAAlgorithmParameterSpec are provided by nCipherKM.

SHA512withML-DSA performs SHA-512 pre-hashing as part of the signature operation.

The ML-DSA context used for verification must match the context used for signing.

8.3.11. 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.



This example may require `sudo` permissions on Linux machines.

8.3.12. 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.



This example may require `sudo` permissions on Linux machines.

8.3.13. 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.

8.3.14. SignaturesExample.java

This example generates RSA, DSA, ECDSA and Ed25519 key pairs. For the associated mechanism of each key type, it performs signing and verification of random plain text.

8.3.15. 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 Java JCE/CSP example utilities

8.3.16. 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 Java JCE/CSP example utilities

8.3.17. 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.

8.3.18. 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.



This example requires the `Bouncy Castle` security provider to be loaded and configured to run properly.

8.4. Java generic stub examples



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.

8.4.1. 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`.

8.4.2. Channel.java

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

8.4.3. CheckMod.java

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

8.4.4. CrypTest.java

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

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

8.4.5. DesKat.java

This example utility is for DES known answer tests.

It demonstrates simple nCore key management usage.

8.4.6. DKTest.java

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

8.4.7. EasyConnection.java

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

8.4.8. Enquiry.java

This example utility displays enquiry information.

It demonstrates:

- simple nCore usage
- the `enquiry` command.

8.4.9. FloodTest.java

This example utility does performance testing for modexp code.

It demonstrates:

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

8.4.10. GenCert.java

This example utility generates a certificate.

It demonstrates the use of the `BuildCmdCert` class.

8.4.11. InitUnit.java

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

8.4.12. 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.

8.4.13. ReportVersion.java

This example utility reports the embedded version information from the current `nfjava` com ponent. `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.

8.4.14. 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`.

8.4.15. SigTest.java

This example utility does signature performance testing.

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



Java is not a high-performance language. On slow host systems or sys- tems 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 ver- sion of `SigTest` instead.

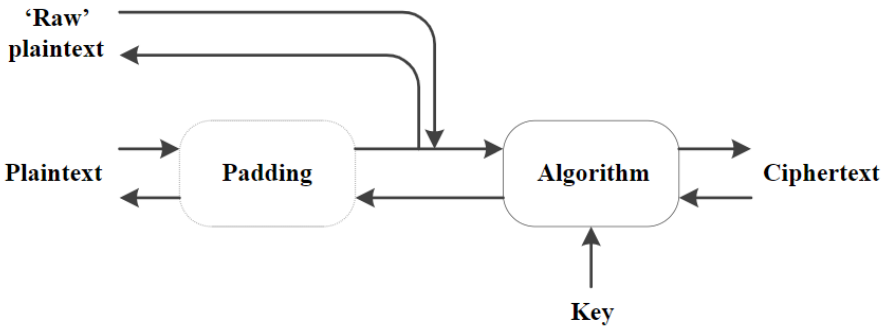
9. Key structures

This chapter describes the data structures used by the nShield 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 nShield 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.

9.1. 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).



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.

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_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_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_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;
};
```

If you require the enum or `#define` value for a mechanism, refer to the nCore API documentation. See nShield Security World: nCore v13.9.6 Developer Tutorial for the API documentation locations.

9.1.1. Mech_Any

Instead of explicitly specifying a mechanism, you can let the module select the mechanism by specifying `Mech_Any`. The nShield 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. If an IV is given, the mechanism specified in the IV is used.

Key Type	Encryption mechanism	Signing mechanism	Verification mechanism
RSAPublic	Mech_RSApPKCS1OAEP, Mech_RSApPKCS1OAEPhSHA224, Mech_RSApPKCS1OAEPhSHA256, Mech_RSApPKCS1OAEPhSHA384 or Mech_RSApPKCS1OAEPhSHA512 chosen based on size of key.		
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.	
MLDSAPublic			Mech_MLDSA, Mech_MLDSAhSHA256, Mech_MLDSAhSHA256det, Mech_MLDSAhSHA512, Mech_MLDSAhSHA512det, Mech_MLDSAhSHAKE128b256 or Mech_MLDSAhSHAKE128b256det.
MLDSAPrivate		Mech_MLDSA, Mech_MLDSAhSHA256, Mech_MLDSAhSHA256det, Mech_MLDSAhSHA512, Mech_MLDSAhSHA512det, Mech_MLDSAhSHAKE128b256 or Mech_MLDSAhSHAKE128b256det.	
SLHDSAPublic			Mech_SLHDSA, Mech_SLHDSAdet, Mech_SLHDSAhSHA256, Mech_SLHDSAhSHA256det, Mech_SLHDSAhSHA512, Mech_SLHDSAhSHA512det, Mech_SLHDSAhSHAKE128b256, Mech_SLHDSAhSHAKE128b256det, Mech_SLHDSAhSHAKE256b512 or Mech_SLHDSAhSHAKE256b512det.

Key Type	Encryption mechanism	Signing mechanism	Verification mechanism
SLHDSAPrivate		Mech_SLHDSA, Mech_SLHDSAdet, Mech_SLHDSAhSHA256, Mech_SLHDSAhSHA256det, Mech_SLHDSAhSHA512, Mech_SLHDSAhSHA512det, Mech_SLHDSAhSHAKE128b256, Mech_SLHDSAhSHAKE128b256det, Mech_SLHDSAhSHAKE256b512 or Mech_SLHDSAhSHAKE256b512det. If no mechanism is specified then Mech_SLHDSA is used as the default. The SLHDSA scheme name is retrieved from the SLHDSA private key data.	
ECDSAPrivate		Mech_ECDSA, Mech_ECDSAhSHA224, Mech_ECDSAhSHA256, Mech_ECDSAhSHA384, or Mech_ECDSAhSHA512 chosen based on size of key.	
DES *(not available in FIPS 140 Level 3 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 Type	Encapsulation mechanism	Decapsulation mechanism
MLKEMPublic	DeriveMech_MLKEMEncaps	

Key Type	Encapsulation mechanism	Decapsulation mechanism
MLKEMPrivate		DeriveMech_MLKEMDecaps

9.2. Key Types

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

Key type	Block size	Encrypt	Decrypt	Sign	Verify	Encapsulation	Decapsulation
ArcFour	-	Y	Y	-	-	-	-
Blowfish	64	-	-	-	-	-	-
-	-	Y	Y	-	-	-	-
CBC MAC	-	-	-	Y	Y	-	-
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	-	-	-	-

Key type	Block size	Encrypt	Decrypt	Sign	Verify	Encapsulation	Decapsulation
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	-	-	-	-
Rijndael	128	-	-	-	-	-	-
CBC	-	Y	Y	-	-	-	-
CBC MAC	-	-	-	Y	Y	-	-
ECB	-	Y	Y	-	-	-	-
GCM	-	Y	Y	-	-	-	-
Twofish	128	-	-	-	-	-	-
CBC	-	Y	Y	-	-	-	-
CBC MAC	-	-	-	Y	Y	-	-
ECB	-	Y	Y	-	-	-	-
Diffie-Hellman	-	-	-	-	-	-	-
Key Exchange	-	-	Y	-	-	-	-
ElGamal	-	Y	Y	-	-	-	-
DSA	-	-	-	Y	Y	-	-
MLDSA	-	-	-	Y	Y	-	-
SLHDSA	-	-	-	Y	Y	-	-
MLKEM	-	-	-	-	-	Y	Y

Key type	Block size	Encrypt	Decrypt	Sign	Verify	Encapsulation	Decapsulation
ECDSA	-	-	-	Y	Y	-	-
ECDH	-	-	-	-	-	-	-
Key Exchange	-	-	Y	-	-	-	-
KCDSA	-	-	-	Y	Y	-	-
RSA	-	-	-	Y	Y	-	-
HMAC	-	-	-	-	-	-	-
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	-	-
HMACSHA3b224	-	-	-	Y	Y	-	-
HMACSHA3b256	-	-	-	Y	Y	-	-
HMACSHA3b384	-	-	-	Y	Y	-	-
HMACSHA3b512	-	-	-	Y	Y	-	-
HMACTiger	-	-	-	Y	Y	-	-
Random	-	-	-	-	-	-	-
Template	-	-	-	-	-	-	-
Wrapped	-	-	-	-	-	-	-

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;
```

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.

9.2.1. Random

9.2.1.1. Key data

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

9.2.1.2. Key generation parameters

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

9.2.1.3. 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`.

9.2.2. 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`.

9.2.2.1. Key data

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

9.2.2.2. Key generation parameters

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

9.2.2.3. Mechanisms

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

9.2.3. 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.

9.2.3.1. Key data

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

9.2.3.2. Key generation parameters

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

9.2.3.3. Mechanisms

- `Mech_BlowfishmECBpNONE`: ECB
- `Mech_BlowfishmCBCpNONE`: CBC
- `Mech_BlowfishmCBCi64PKCS5`: CBC
- `Mech_BlowfishmCBCMACi64PKCS5`: CBC MAC
  This mechanism is deprecated and may be withdrawn in future firmware.
- `Mech_BlowfishmECBpPKCS5`: ECB
- `Mech_BlowfishmCBCMACi0PKCS5`: CBC MAC

9.2.4. 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.

9.2.4.1. Mechanisms

- `Mech_CASTmCBCi64pPKCS5`: CBC
- `Mech_CASTmCBCMACi64pPKCS5`
  This mechanism is deprecated and may be withdrawn in future firmware.
- `Mech_CASTmECBpPKCS5`: ECB
- `Mech_CASTmCBCMACi0pPKCS5`: CBC MAC

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

9.2.5. 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.

9.2.5.1. 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`
  This mechanism is deprecated and may be withdrawn in future versions.
- `Mech_CAST256mCBCMACi0pPKCS5`: CBC MAC

9.2.6. DES

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

9.2.6.1. 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;
```

56 bits plus 8 parity bits

9.2.6.2. Key generation parameters

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

9.2.6.3. 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 par ity set correctly, the module will return `Status_InvalidData`.

9.2.6.4. Mechanisms

- `Mech_DESmCBCpNONE`: CBC no padding
- `Mech_DESmCBCi64pPKCS5`: CBC with PKCS5 padding
- `Mech_DESmCBCMACi64pPKCS5`
  This mechanism is deprecated and may be withdrawn in future versions.
- `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

9.2.6.5. CBC

9.2.6.5.1. Cipher text

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

9.2.6.5.2. IV

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

9.2.6.6. CBC MAC

9.2.6.6.1. Cipher text

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

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.

9.2.7. DES2

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

9.2.7.1. 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;
```

112 bit plus 16 parity bits.

9.2.7.2. Key generation parameters

There are no key generation parameters.

9.2.7.3. 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`.

9.2.7.4. Mechanisms

- `Mech_DES2mCBCpNONE`: CBC no padding
- `Mech_DES2mCBCi64pPKCS5`: CBC with PKCS5 padding
- `Mech_DES2mCBCMACi64pPKCS5`
  This mechanism is deprecated and may be withdrawn in future versions.
- `Mech_DES2mECBpNONE`: ECB no padding
- `Mech_DES2mECBpPKCS5`: ECB with PKCS5 padding
- `Mech_DES2mCBCMACi0pPKCS5`: CBCMAC with PKCS5 padding
- `Mech_DES2mCBCMACi0pNONE`: CBC MAC with no padding

9.2.7.5. CBC

9.2.7.5.1. Cipher text

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

9.2.7.5.2. IV

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

9.2.8. Triple DES

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

9.2.8.1. 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;
```

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

9.2.8.2. Key generation parameters

There are no key generation parameters.

9.2.8.3. Mechanisms

- `Mech_DES3mCBCi64pPKCS5`: CBC with PKCS #5 padding
- `Mech_DES3mCBCMACi64pPKCS5`
  This mechanism is deprecated and may be withdrawn in future versions.
- `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

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

nShield uses outer CBC.

9.2.9. Rijndael

Rijndael is now FIPS approved as the AES. The 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`.

9.2.9.1. Mechanisms

- `Mech_RijndaelmCBCpNONE`: CBC
- `Mech_RijndaelmCBCi128pPKCS5`: CBC with PKCS5 padding
- `Mech_RijndaelmCBCMACi128pPKCS5`
  This mechanism is deprecated and may be withdrawn in future versions.
- `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

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

9.2.9.2. 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).

9.2.10. 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.

If you wish to use the SEED algorithm, you must order and enable it as part of the nShield `KISAAlgorithms` feature.

9.2.10.1. Key data

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

9.2.10.2. Key generation parameters

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

9.2.10.3. 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`
  This mechanism is deprecated and may be withdrawn in future versions.
- `Mech_SEEDmCBCMACi0pPKCS5`: CBCMAC

9.2.11. 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.

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.

9.2.11.1. Key data

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

9.2.11.2. Key generation parameters

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

9.2.11.3. Mechanisms

- `Mech_SerpentmECBpNONE`: ECB with no padding
- `Mech_SerpentmCBCpNONE`: CBC with no padding
- `Mech_SerpentmCBCi128PKCS5`: CBC with PKCS #5 padding
- `Mech_SerpentmCBCMACi128PKCS5`
  This mechanism is deprecated and may be withdrawn in future versions.
- `Mech_SerpentmECBpPKCS5`: ECB with PKCS #5 padding
- `Mech_SerpentmCBCMACi0PKCS5`: CBCMAC

9.2.12. 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.

9.2.12.1. Key data

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

9.2.12.2. Key generation parameters

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

9.2.12.3. Mechanisms

- `Mech_TwofishmECBpNONE`: ECB with no padding
- `Mech_TwofishmCBCpNONE`: CBC with no padding
- `Mech_TwofishmCBCi128PKCS5`: CBC with PKCS #5 padding
- `Mech_TwofishmCBCMACi128PKCS5`
  This mechanism is deprecated and may be withdrawn in future versions.
- `Mech_TwofishmECBpPKCS5`: ECB with PKCS #5 padding
- `Mech_TwofishmCBCMACi0PKCS5`: CBCMAC

9.2.13. Diffie-Hellman and ElGamal

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

decryption.

9.2.13.1. 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.

9.2.13.2. 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.

9.2.13.3. 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.

  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;
```

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.

9.2.13.4. Mechanisms

- `Mech_DHKeyExchange`
- `Mech_ElGamal`
- `Mech_DLIESe3DEShSHA1`
- `Mech_DLIESeAEShSHA1`

9.2.13.4.1. 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:

$$(G^{X_A})^{X_B} = G(X_A X_B) \bmod P$$

4. Bob takes Alice's public key and passes it to Decrypt using his private key. This returns, in bignum format:

$$(G^{X_B})^{X_A} = G(X_B X_A) = G(X_A X_B) \bmod P$$

   This result is the same as that which Alice derived.

5. The session key can then be derived from this multi-precision number.

9.2.13.4.2. ElGamal

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

9.2.13.4.3. 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

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

9.2.13.5. Cipher text

9.2.13.5.1. Diffie-Hellman

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

9.2.13.5.2. ElGamal

```
typedef struct {
    M_Bignum a    [gk mod p]
    M_Bignum b    M * (gxk) mod p
} M_Mech_ElGamal_Cipher;
```

where k is a random integer 1 < k < (p—1)

ElGamal signature creation and verification are not currently implemented.

9.2.14. 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 `DSADis`

creteLogGroup 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.

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

9.2.14.1. DSA keys

9.2.14.1.1. DSA common key

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

9.2.14.1.2. DSA private key

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

9.2.14.1.3. 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 $h^{(p—1)/q}$, where `h < p—1` and $h^{((p—1)/q)} \bmod p > 1$.
- This is the discrete logarithm group. These values may be shared between users.
- A 160-bit number `< q`.

$g^{x} \bmod p$ (a `p`-bit number).

9.2.14.2. 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.

9.2.14.3. 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 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 $h^{((p—1)/q)} \bmod p > 1$`.

9.2.14.4. Cipher text

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

`r` is $g^k \bmod p \bmod q$

`s` is $k^{-1}(H(m)+xr)) \bmod q$

9.2.14.5. 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.

9.2.14.6. Mechanisms

- `Mech_DSA`
- `Mech_DSAhSHA224`
- `Mech_DSAhSHA256`
- `Mech_DSAhSHA384`
- `Mech_DSAhSHA512`
- `Mech_DSAhRIPEMD160`

9.2.15. ML-DSA

9.2.15.1. Keys

For ML-DSA keys, we store the keys as pure bytes without any specific details about the components forming the keys.

```
struct M_KeyType_MLDSAPrivate_Data {
  /** Private key data as defined in FIPS-204
   **
   ** - Maximum length: 4896 bytes */
  M_ByteBlock sk;
};
```

```
struct M_KeyType_MLDSAPublic_Data {
  /** Public key data as defined in FIPS-204
   **
   ** - Maximum length: 2592 bytes */
  M_ByteBlock pk;
};
```

9.2.15.2. Plain text

Mech	Unhashed plain text type	Hash used for bytes plaintext
Mech_MLDSA	Bytes or Hash64	
Mech_MLDSAdet	Bytes or Hash64	
Mech_MLDSAhSHA256	Hash32	SHA-256
Mech_MLDSAhSHA256det	Hash32	SHA-256
Mech_MLDSAhSHA512	Hash64	SHA-512
Mech_MLDSAhSHA512det	Hash64	SHA-512
Mech_MLDSAhSHAKE128b256	Hash32	SHAKE128
Mech_MLDSAhSHAKE128b256det	Hash32	SHAKE128
Mech_MLDSAhSHAKE256b512	Hash64	SHAKE256
Mech_MLDSAhSHAKE256b512det	Hash64	SHAKE256

9.2.15.3. Schemes

- MLDSAScheme_MLDSA44
- MLDSAScheme_MLDSA65

- `MLDSAScheme_MLDSA87`

9.2.15.4. Supported Key Sizes

ML-DSA Scheme	Private Key Size (bytes)	Public Key Size (bytes)
ML-DSA-44	2560	1312
ML-DSA-65	4032	1952
ML-DSA-87	4896	2592

9.2.15.5. Mechanisms

- `Mech_MLDSA`
- `Mech_MLDSAhSHA256`
- `Mech_MLDSAhSHA512`
- `Mech_MLDSAhSHA512det`
- `Mech_MLDSAhSHAKE128b256`
- `Mech_MLDSAhSHAKE128b256det`
- `Mech_MLDSAhSHAKE256b512`
- `Mech_MLDSAhSHAKE256b512det`
- `DeriveMech_MLDSAGenerateInternal`

Mechanisms with the 'det' suffix means deterministic signature mechanisms and the ones without 'det' are randomized signatures.

`DeriveMech_MLDSAAGenerateInternal` can be used for key generation, however it is unavailable in FIPS-140 Level 3 mode.

9.2.16. SLH-DSA

9.2.16.1. Keys

For SLH-DSA keys, we store the keys as pure bytes without any specific details about the components forming the keys.

```
struct M_KeyType_SLHDSAPrivate_Data {
  /** Public key data as defined in FIPS-205 */
  SpincsScheme scheme;
  /** Private key data as defined in FIPS-205
```

```
   **
   ** - Maximum length: 128 bytes */
  ByteBlock sk;
};
```

```
struct M_KeyType_SLHDSAPublic_Data {
  /** Public key data as defined in FIPS-205 */
  SpincsScheme scheme;
  /** Private key data as defined in FIPS-205
   **
   ** - Maximum length: 64 bytes */
  ByteBlock pk;
};
```

9.2.16.2. Plain text

Mech	Unhashed plain text type	Hash used for bytes plaintext
Mech_SLHDSA	Hash	
Mech_SLHDSAdet	Hash	
Mech_SLHDSAhSHA256	Hash32	SHA-256
Mech_SLHDSAhSHA256det	Hash32	SHA-256
Mech_SLHDSAhSHA512	Hash64	SHA-512
Mech_SLHDSAhSHA512det	Hash64	SHA-512
Mech_SLHDSAhSHAKE128b256	SHAKE128	SHA-256
Mech_SLHDSAhSHAKE128b256det	SHAKE128	SHA-256
Mech_SLHDSAhSHAKE256b512	SHAKE256	SHA-512
Mech_SLHDSAhSHAKE256b512det	SHAKE256	SHA-512

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

9.2.16.3. Schemes

- SphincsSchemeName_SLHDSA128FhSHA2
- SphincsSchemeName_SLHDSA128ShSHA2
- SphincsSchemeName_SLHDSA128FhSHAKE
- SphincsSchemeName_SLHDSA128ShSHAKE
- SphincsSchemeName_SLHDSA192FhSHA2

- SphincsSchemeName_SLHDSA192ShSHA2
- SphincsSchemeName_SLHDSA192FhSHAKE
- SphincsSchemeName_SLHDSA192ShSHAKE
- SphincsSchemeName_SLHDSA256FhSHA2
- SphincsSchemeName_SLHDSA256ShSHA2
- SphincsSchemeName_SLHDSA256FhSHAKE
- SphincsSchemeName_SLHDSA256ShSHAKE

9.2.16.4. Supported Key Sizes

SLH-DSA Scheme	pk bytes	sig bytes	sk bytes
SLH-DSA-SHA2-128s	32	7856	4*n
SLH-DSA-SHAKE-128s	32	7856	4*n
SLH-DSA-SHA2-128f	32	17088	4*n
SLH-DSA-SHAKE-128f	32	17088	4*n
SLH-DSA-SHA2-192s	48	16244	4*n
SLH-DSA-SHAKE-192s	48	16244	4*n
SLH-DSA-SHA2-192f	48	35664	4*n
SLH-DSA-SHAKE-192f	48	35664	4*n
SLH-DSA-SHA2-256s	64	29792	4*n
SLH-DSA-SHAKE-256s	64	29792	4*n
SLH-DSA-SHA2-256f	64	49856	4*n
SLH-DSA-SHAKE-256f	64	49856	4*n

9.2.16.5. Mechanisms

- Mech_SLHDSA
- Mech_SLHDSAdet
- Mech_SLHDSAhSHA256
- Mech_SLHDSAhSHA256det
- Mech_SLHDSAhSHA512
- Mech_SLHDSAhSHA512det
- Mech_SLHDSAhSHAKE128b256

- `Mech_SLHDSAhSHAKE128b256det`
- `Mech_SLHDSAhSHAKE256b512`
- `Mech_SLHDSAhSHAKE256b512det`
- `DeriveMech_SLHDSAGenerateInternal`

Mechanisms with the 'det' suffix means deterministic signature mechanisms and the ones without 'det' are randomized signatures.

`DeriveMech_SLHDSAGenerateInternal` can be used for key generation, however it is unavailable in FIPS-140 Level 3 mode.

9.2.17. ML-KEM

9.2.17.1. Keys

For ML-KEM keys, we store the keys as pure bytes without any specific details about the components forming the keys.

```
struct M_KeyType_MLKEMPrivate_Data {
  /** Decapsulation key data as defined in FIPS-203
   **
   ** - Maximum length: 3168 bytes */
  Byteblock dk;
  /** Seed data d || z, or 0 byte string if not present
   **
   ** - Maximum length: 64 bytes */
  ByteBlock dz;
  /** MLKEM-512/768/1024 as defined in the FIPS-203 publication */
  MLKEMScheme scheme;
};
```

```
struct M_KeyType_MLKEMPublic_Data {
  /** Encapsulation key data as defined in FIPS-203
   **
   ** - Maximum length: 2592 bytes */
  Byteblock ek;
};
```

9.2.17.2. Schemes

- `MLKEMScheme_MLKEM512`
- `MLKEMScheme_MLKEM768`
- `MLKEMScheme_MLKEM1024`

9.2.17.3. Supported Key Sizes

ML-KEM Scheme	Encapsulation Key	Decapsulation Key	CipherText	Shared Secret Key
ML-KEM-512	800	1632	768	32
ML-KEM-768	1184	2400	1088	32
ML-KEM-1024	1568	3168	1568	32

9.2.17.4. Mechanisms

- `DeriveMech_MLKEMEncaps`
- `DeriveMech_MLKEMDecaps`
- `DeriveMech_MLKEMGenerateInternal`

`DeriveMech_MLKEMGenerateInternal` can be used for key generation, however it is unavailable in FIPS-140 Level 3 mode.

9.2.18. Elliptic Curve ECDH and ECDSA

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

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

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.

9.2.18.1. Elliptic Curve keys

9.2.18.1.1. 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.

9.2.18.1.2. 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.

9.2.18.2. Key generation parameters

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

- `curve` is the curve used.

9.2.18.3. 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.

9.2.18.4. Cipher text - ECDSA

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

`r` is $x_1 \bmod n$

`s` is $s = k^{-1}(e + dr) \bmod n$.

9.2.18.5. 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.

9.2.18.6. 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

9.2.18.7. Mechanisms

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

Neither Mech_ECDSA nor Mech_ECDH handle normal representations.

9.2.19. 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.

If you wish to use the KCDSA algorithm, you must order and enable it as part of the `KISAAlgorithms` feature. If you are outside Korea, contact for information about obtaining the

appropriate export licence.

KCDSA 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.

9.2.19.1. KCDSA keys

9.2.19.1.1. 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 $h^{((p-1)/q)} \bmod p > 1$.

9.2.19.1.2. 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 $h^{((p-1)/q)} \bmod p > 1$.
- x is an arbitrary number where $0 < x < q$.
- y is $g^{(1/x \bmod q)} \bmod p$ (a number less than p).

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

9.2.19.2. Key generation parameters

9.2.19.2.1. 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 $h^{(p-1)/q}$, where $h < p-1$ and $h^{((p-1)/q)} \bmod p > 1$.

9.2.19.3. Cipher text

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

- `r` is $h(g^k \bmod p)$.
- `s` is $x(k - (r \oplus h(z||m))) \bmod q$

The symbol $\oplus$ represents a bit-wise XOR operation. The symbol || represents concatenation of `Byteblock`s

9.2.19.4. Plain text

See Key Types 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`.

9.2.19.5. Mechanisms

- `Mech_KCDSAHAS160`
- `Mech_KCDSASHA1`
- `Mech_KCDSARIPEMD160`
- `Mech_KCDSASHA224`
- `Mech_KCDSASHA256`

9.2.20. RSA

9.2.20.1. 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 nShield module have the public exponent `0x10001` by default.

9.2.20.2. 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 $D \mod P^{-1}$
- `dmq1` is $D \mod Q^{-1}$
- `iqmp` is $Q^{-1} \mod P$

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.

9.2.20.3. 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.

9.2.20.4. Mechanisms

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

- `Mech_RSApPKCS1=` (see RSA mechanisms note 1)
- `Mech_RSAhSHA1pPKCS1=` (see RSA mechanisms note 2)
- `Mech_RSAhRIPEMD160pPKCS1=` (see RSA mechanisms note 2)
- `Mech_RSApPKCS1OAEP=` (see RSA mechanisms note 3)
- `Mech_RSApPKCS1OAEPhSHA224`
- `Mech_RSApPKCS1OAEPhSHA256`
- `Mech_RSApPKCS1OAEPhSHA384`
- `Mech_RSApPKCS1OAEPhSHA512`
- `Mech_RSAhSHA1pPSS`
- `Mech_RSAhRIPEMD160pPSS`
- `Mech_RSAhSHA224pPSS`
- `Mech_RSAhSHA256pPSS`
- `Mech_RSAhSHA384pPSS`
- `Mech_RSAhSHA512pPSS`

9.2.20.4.1. RSA mechanisms note 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.

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.

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.

9.2.20.4.2. RSA mechanisms note 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.

  You must make sure 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.

9.2.20.4.3. RSA mechanisms note 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.

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.

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.

9.2.20.5. Cipher text - PKCS #11 padding

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

9.2.20.6. Cipher text - OAEP padding

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

9.2.21. DeriveKey

9.2.21.1. 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.

9.2.21.2. 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.

9.2.21.3. 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

9.2.21.4. Derive Key Mechanisms

- `DeriveMech_DESsplitXOR` (see DeriveKey mechanisms note 1).
- `DeriveMech_DESjoinXOR` (see DeriveKey mechanisms note 2)
- `DeriveMech_DES2splitXOR` (see DeriveKey mechanisms note 1).
- `DeriveMech_DES2joinXOR` (see DeriveKey mechanisms note 2)
- `DeriveMech_DES3splitXOR` (see DeriveKey mechanisms note 1).
- `DeriveMech_DES3joinXOR` (see DeriveKey mechanisms note 2)

- `DeriveMech_DESjoinXORsetParity` (see DeriveKey mechanisms note 2)
- `DeriveMech_DES2joinXORsetParity` (see DeriveKey mechanisms note 2)
- `DeriveMech_DES3joinXORsetParity` (see DeriveKey mechanisms note 2)
- `DeriveMech_RandsplitXOR` (see DeriveKey mechanisms note 1).
- `DeriveMech_RandjoinXOR` (see DeriveKey mechanisms note 2)
- `DeriveMech_CASTsplitXOR` (see DeriveKey mechanisms note 1).
- `DeriveMech_CASTjoinXOR` (see DeriveKey mechanisms note 2)
- `DeriveMech_EncryptMarshalled` (see DeriveKey mechanisms note 3)
- `DeriveMech_DecryptMarshalled` (see DeriveKey mechanisms note 3)
- `DeriveMech_PKCS8Encrypt` (see DeriveKey mechanisms note 4)
- `DeriveMech_PKCS8Decrypt` (see DeriveKey mechanisms note 4)
- `DeriveMech_RawEncrypt` (see DeriveKey mechanisms note 5)
- `DeriveMech_RawDecrypt` (see DeriveKey mechanisms note 5)
- `DeriveMech_AESsplitXOR` (see DeriveKey mechanisms note 1).
- `DeriveMech_AESjoinXOR` (see DeriveKey mechanisms note 2)
- `DeriveMech_Any`
- `DeriveMech_PublicFromPrivate` (see DeriveKey mechanisms note 6)
- `DeriveMech_ECCMQV`
- `DeriveMech_ConcatenateBytes`
- `DeriveMech_ConcatenationKDF`
- `DeriveMech_NISTKDFmCTRpRijndaelCMACr32`
- `DeriveMech_RawEncryptZeroPad`
- `DeriveMech_RawDecryptZeroPad`
- `DeriveMech_AESKeyWrap`
- `DeriveMech_AESKeyUnwrap`
- `DeriveMech_ECKA`
- `DeriveMech_ECIESKeyWrap` (see DeriveKey mechanisms note 7)
- `DeriveMech_ECIESKeyUnwrap` (see DeriveKey mechanisms note 8)

9.2.21.4.1. DeriveKey mechanisms note 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`.

9.2.21.4.2. DeriveKey mechanisms note 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.

9.2.21.4.3. DeriveKey mechanisms note 3

The EncryptMarshalled and DecryptMarshalled mechanisms are provided to allow export of keys from a module in FIPS 140 Level 3 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:

1. Marshals an M_PlainText structure that represents the base key to produce a byte string.
2. Turns the byte string into Bytes plaintext, and encrypts it with the wrapping key to produce ciphertext.
3. Marshals the ciphertext into a further byte string.
4. 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.

   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 FIPS 140 Level 3 mode. The import must be authorized by a certificate signed by the nShield 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 plain text to recover the M_PlainText structure for the output key.

An example of importing keys using the DecryptMarshalled mechanism:

1. 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.
2. 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).
3. 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.
4. Create a template key Kt that contains the desired ACL for the key to be imported, and

import it.

5. 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.

9.2.21.4.4. DeriveKey mechanisms note 4

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 nShield 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).

9.2.21.4.5. DeriveKey mechanisms note 5

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. Alternatively, for `RawEncrypt` only, a signing or hashing mechanism can be provided instead of an encrypt one. In these cases, the raw key data is signed or hashed instead.



This mechanism is not intended for secure transport of key data between nShield 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:

1. It extracts the key data of the Base key as a byte block.
2. If an encryption mechanism is specified in the `IV`, the key data is encrypted 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.
   If a signing mechanism is specified in the `IV`, the key data is signed using the Wrapping key, `IV` and the mechanism specified in the `IV`, which must be a valid mechanism for the given Wrapping Key.
   If a hashing mechanism is specified in the `IV`, the key data is hashed using the Wrap-

ping key (if the mechanism requires one), `IV` and the mechanism specified in the `IV`, which must be a valid mechanism for the given Wrapping Key. HMAC mechanisms require a wrapping key and others do not. For more information see HMAC signatures.

3. The resulting ciphertext, signature or `hash` 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. Alternatively, the specified mechanism in the `IV` does not 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 des tination 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.

9.2.21.4.6. DeriveKey mechanisms note 6

`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.

9.2.21.4.7. DeriveKey mechanisms note 7

The `DeriveMech_ECIESKeyWrap` mechanism takes a base key of the specified symmetric type and a wrapping key of type `ECDHPublic` to produce an output key of type `Wrapped`.

9.2.21.4.8. DeriveKey mechanisms note 8

The `DeriveMech_ECIESKeyUnwrap` mechanism takes a base key of type `ciphertext` and a wrapping key of type `ECDHPrivate` to produce an output key of type `keytype`.

9.3. 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.

9.3.1. SHA-1

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

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

9.3.1.1. Mechanism

```
Mech_SHA1Hash
```

9.3.1.2. Reply

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

9.3.2. 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.

9.3.2.1. Mechanism

```
Mech_TigerHash
```

9.3.2.2. Reply

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

9.3.3. SHA-224

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

9.3.3.1. Mechanism

```
Mech_SHA224
```

9.3.3.2. Reply

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

9.3.4. SHA-256

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

9.3.4.1. Mechanism

```
Mech_SHA256
```

9.3.4.2. Reply

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

9.3.5. SHA-384

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

9.3.5.1. Mechanism

```
Mech_SHA384Hash
```

9.3.5.2. Reply

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

9.3.6. SHA-512

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

9.3.6.1. Mechanism

```
Mech_SHA512Hash
```

9.3.6.2. Reply

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

9.3.7. MD2

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

9.3.7.1. Mechanism

```
Mech_MD2Hash
```

9.3.7.2. Reply

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

9.3.8. MD5

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

9.3.8.1. Mechanism

```
Mech_MD5Hash
```

9.3.8.2. Reply

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

9.3.9. 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.

9.3.9.1. Mechanism

```
Mech_RIPEMD160Hash
```

9.3.9.2. Reply

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

9.3.10. HAS160

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

If you wish to use the HAS160 hash function, you must order and enable it as part of the `KISAAlgorithms` feature.

9.3.10.1. Mechanism

```
Mech_HAS160Hash
```

9.3.10.2. Reply

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

9.4. 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.

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_HMACSHA3b224`
- `KeyType_HMACSHA3b256`
- `KeyType_HMACSHA3b384`
- `KeyType_HMACSHA3b512`
- `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_HMACSHA3b224`
- `Mech_HMACSHA3b256`
- `Mech_HMACSHA3b384`
- `Mech_HMACSHA3b512`
- `Mech_HMACTiger`

9.5. 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.
  - `PermissionGroup_flags_certifier_present` Set this flag if actions in this group must be certified with a key that matches the given hash. 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.

    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_LogKeyUsage` Set this flag if Sign, Verify, Encrypt or Decrypt (and corresponding `Cmd_ChannelOpen`) actions in this group should be logged by the nShield Audit Logging capability.

    If Audit Logging is not enabled for the module attempting to use a key with PermissionGroup_flags_LogKeyUsage set the module returns Status_InvalidACL.

- `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.
- `PermissionGroup_flags_NSOCertified` Set this flag if the actions in this group must be certified by the Security Officer's key K_{NSO}, whatever that is set to for this module at this time.

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.

  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.

`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 `NewEn-`

quiry command for the module.

9.6. 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 cre ated, it is set to 0. Any time a user performs an action that requires a use limit, the mod ule 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. Oth erwise, 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.

  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 decre ment 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 cre ated, it is set to 0. Any time a user performs an action that requires a use limit, the mod ule 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. Oth erwise, 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 `Read Share` 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.

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.

9.7. 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
- `Act_MakeBlob=` - see MakeBlob
- `Act_MakeArchiveBlob=` - see MakeArchiveBlob
- `Act_NSOPermissions=` - see NSO
- `Act_DeriveKey=` - see DeriveKey and DeriveKeyEx
- `Act_DeriveKeyEx=` - see DeriveKey and DeriveKeyEx

- `Act_NVMemOpPerms=` - see NVRAM
- `Act_FeatureEnable=` - see NVRAM
- `Act_NVMemUseLimit=`
- `Act_SendShare=` - see SendShare
- `Act_ReadShare=` - see ReadShare
- `Act_StaticFeatureEnable=`
- `Act_UserAction=` - see UserAction
- `Act_FileCopy=` - see FileCopy

`details` depend on the chosen action type:

```
union M_Act__Details {
    M_Act_FeatureEnable_Details featureenable;
    M_Act_DeriveKey_Details derivekey;
    M_Act_DeriveKeyEx_Details derivekeyex;
    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;
};
```

9.8. Action types

9.8.1. 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, 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

9.8.2. 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 K_{M0}. If this flag is *not* set, you must use K_{M0} or logical tokens based on 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`

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.

9.8.3. 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.

- `*kahash` is the key hash.
- `*blobfile` — see MakeBlob

9.8.4. 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

9.8.5. 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 oper ation.

9.8.6. 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;
```

9.8.7. 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

9.8.8. 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.

9.8.9. 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;
};
```

9.8.10. DeriveKey and DeriveKeyEx

These action types enable the key to be used in the `DeriveKey` command. They allow 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. If the `Cmd_DeriveKey_Args_flags_WorldHashMech` flag has been set in the `DeriveKey` command, then the `DeriveKeyEx` action should be used.

```
typedef struct {
    M_Word flags;
```

```
    M_DeriveRole role;
    M_DeriveMech mech;
    int n_otherkeys;
    M_KeyRoleID *otherkeys;
    M_DKMechParams *params;
} M_Act_DeriveKey_Details;
```

```
typedef struct {
    M_Act_DeriveKeyEx_Details_flags flags;
    M_DeriveRole role;
    M_DeriveMech mech;
    int n_otherkeys;
    M_vec_KeyRoleIDEx otherkeys;
    M_DKMechParams *params;
} M_Act_DeriveKeyEx_Details;
```

- The following `flags` are defined:
  - `Act_DeriveKey_Details_flags_params_present`
  - `Act_DeriveKeyEx_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.
- `n_otherkeys` - the number of keys in the `otherkeys` 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;
  ```

  ```
  typedef struct {
      M_DeriveRole role;
      M_KeyHashEx hash;
  } M_KeyRoleIDEx;
  ```

  - `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 either SHA-1 or a stronger hash determined by the

`Cmd_DeriveKey_Args_flags_WorldHashMech`, which can be obtained via the `GetKeyInfoEx` command.

- `*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`

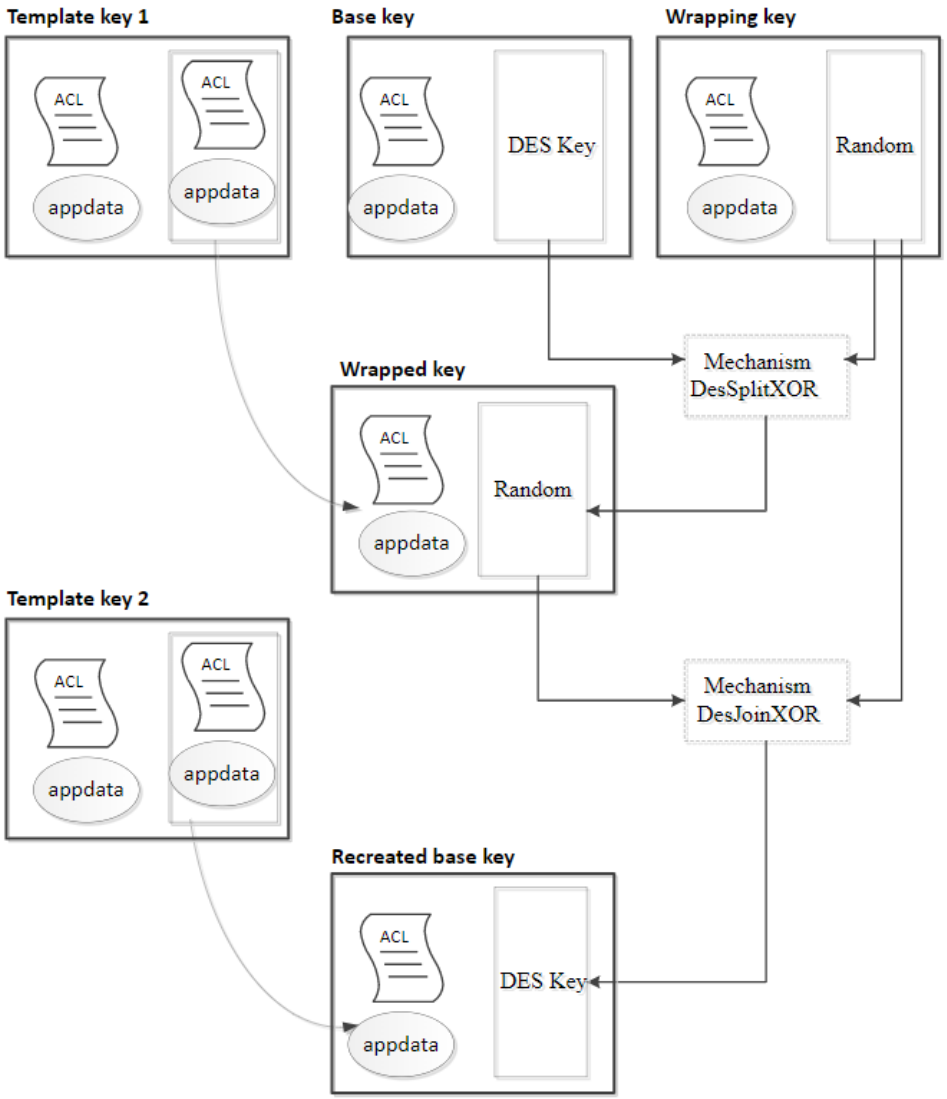
  The derive key mechanism parameters— see Derive Key Mechanisms.

  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
  - 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.

9.8.11. 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. The following diagram illustrates this 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 there fore 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

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
```

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 para meters:

```
key         IDKA 0015
```

This command returns:

```
type        DES
hash        HKA 0013
```

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

9.9. Certificates

The nShield module uses certificates to enable a given user to authorize another user to per form 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 remembers a maximum of 126 challenges. These challenges automatically expire after 30 seconds or on redemption. Expired challenges are removed from the mod- ules memory.

If the module memory contains more than 40 challenges, it delays the issuance of new chal lenges. This means that with 40 or fewer challenges outstanding, it issues new challenges instantly. With more than 40 challenges outstanding, you get a successful response after a two-second delay. If the module memory is full, it fails after a delay of four seconds unless an existing outstanding challenge expires or is redeemed during the delay period. In this situ ation, if an outstanding challenge expires or is redeemed during the delay, you get a suc- cessful response. These delays apply to each `Cmd_GetChallenge` independently.

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 cer tificate 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.

9.9.1. 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`.

9.9.2. 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.

9.9.2.1. 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_KeyID 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`.

10. NFKM 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.

10.1. 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.

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.

10.2. 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.

10.2.1. NFKM_changepp

Change the passphrase 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 passphrase hash
- `const M_Hash *newpp` is a pointer to the new passphrase hash
- `NFKM_ShareFlag remove` is a list of shares whose passphrases you want to remove, regardless of newpp
- `NFKM_ShareFlag` set is a list of shares whose passphrases you want to set or change.



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.

10.2.2. 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).

10.2.3. NFKM_checkpp

Verifies that a passphrase is correct for a given card. Each share on the card which has a passphrase 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
);
```

10.2.4. 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.

10.2.5. 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.

10.2.6. 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.

10.2.7. 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.

10.2.8. 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
);
```

10.2.9. 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.

10.2.10. NFKM_hashpp

This function hashes a passphrase for use as an Operator Card Set passphrase:

```
M_Status NFKM_hashpp(
                                NFast_AppHandle app,
                                NFastApp_Connection conn,
                                const char *string,
                                M_Hash *hash_r,
                                struct NFast_Call_Context *cctx
);
```

10.2.11. NFKM_initworld_*

10.2.11.1. NFKM_initworld_abort

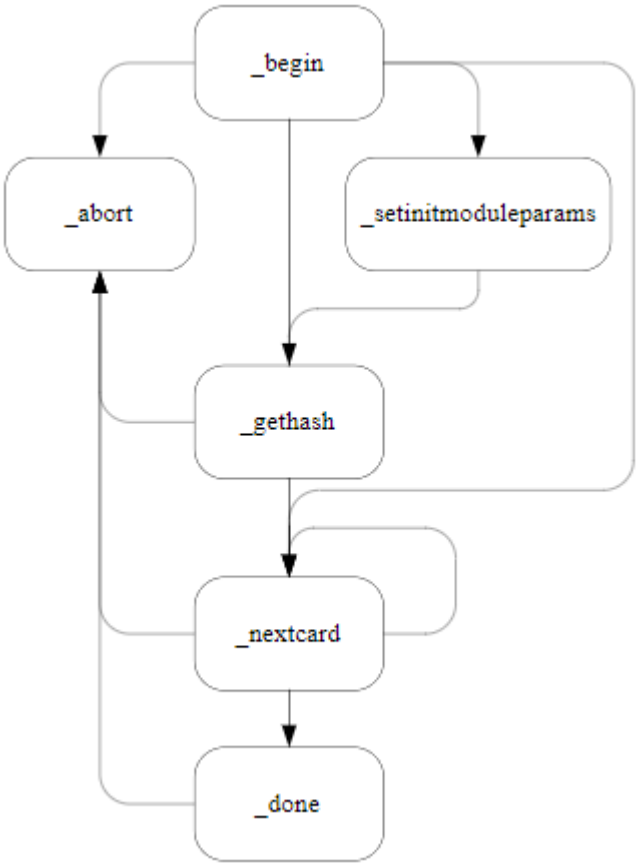
Destroys a Security World initialization context.

```
void NFKM_initworld_abort(
                                      NFKM_InitWorldHandle iwh
);
```

- `NFKM_InitWorldHandle iwh` is the handle for Security World initialization returned by `NFKM_initworld_begin`.

10.2.11.2. NFKM_initworld_begin

Does the initial part of work for a Security World initialization. The following diagram illustrates the paths through the NFKM_initworld process:




```
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.

10.2.11.3. 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`.

10.2.11.4. 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

10.2.11.5. 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.

10.2.11.6. NFKM_initworld_setinitmoduleparams

Configures the parameters for module initialization at the end of the world initialization.

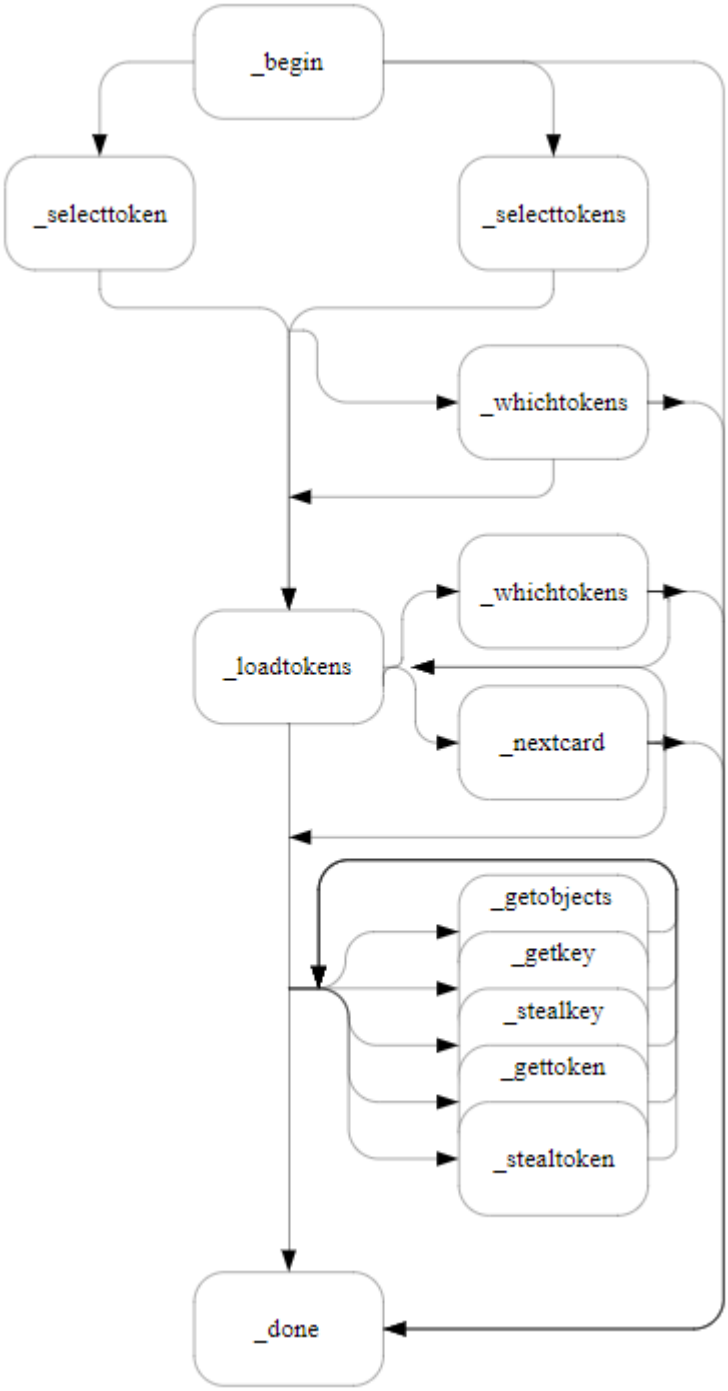
```
M_Status NFKM_initworld_setinitmoduleparams(
                                                        NFKM_InitWorldHandle iwh,
                                                        const NFKM_InitModuleParams *imp
);
```

- `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.

10.2.12. NFKM_loadadminkeys_*

10.2.12.1. NFKM_loadadminkeys_begin

Initializes an operation to load administrator keys. Initially, no tokens are selected for loading. The following diagram illustrates the paths through the `NFKM_loadadminkeys` process:




```
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.

10.2.12.2. 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`

10.2.12.3. 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



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.

10.2.12.4. 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.



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 LT_R has been loaded. Two calls are made to getobjects: one which fetches K_{RE}, and a second which steals the token LT_R. It is no longer possi-

ble to get K_{RA} (because LT_R is now unavailable), but further requests to get 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.

10.2.12.5. 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.

10.2.12.6. 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 passphrase hash, or `NULL` if the card has no passphrase
- `int *left` is the address at which to store the number of cards remaining.

10.2.12.7. 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.

10.2.12.8. 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.

10.2.12.9. 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.

10.2.13. NFKM_loadcardset_*

10.2.13.1. NFKM_loadcardset_abort

This function aborts the loading of a card set:

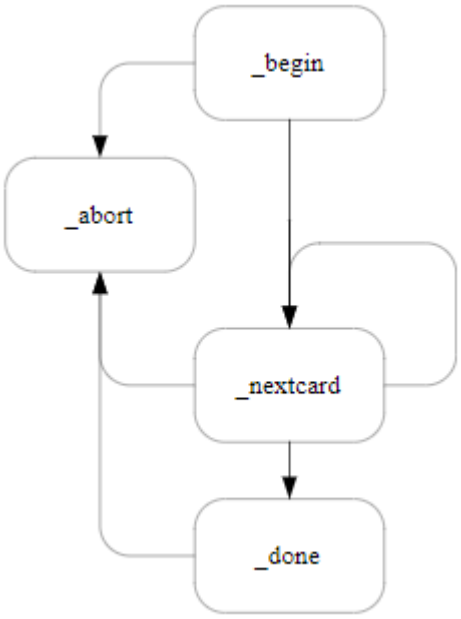
```
void NFKM_loadcardset_abort(
                                        NFKM_LoadCSHandle state
);
```

10.2.13.2. NFKM_loadcardset_begin



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. The following diagram illustrates the paths through the NFKM_loadcardset process:




```
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
);
```

10.2.13.3. NFKM_loadcardset_done

This function completes the loading of a card set:

```
M_Status NFKM_loadcardset_done(
                                        NFKM_LoadCSHandle state,
                                        M_KeyID *logtokid_r
);
```

10.2.13.4. NFKM_loadcardset_nextcard



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.

10.2.14. NFKM_loadworld_*

10.2.14.1. NFKM_loadworld_abort

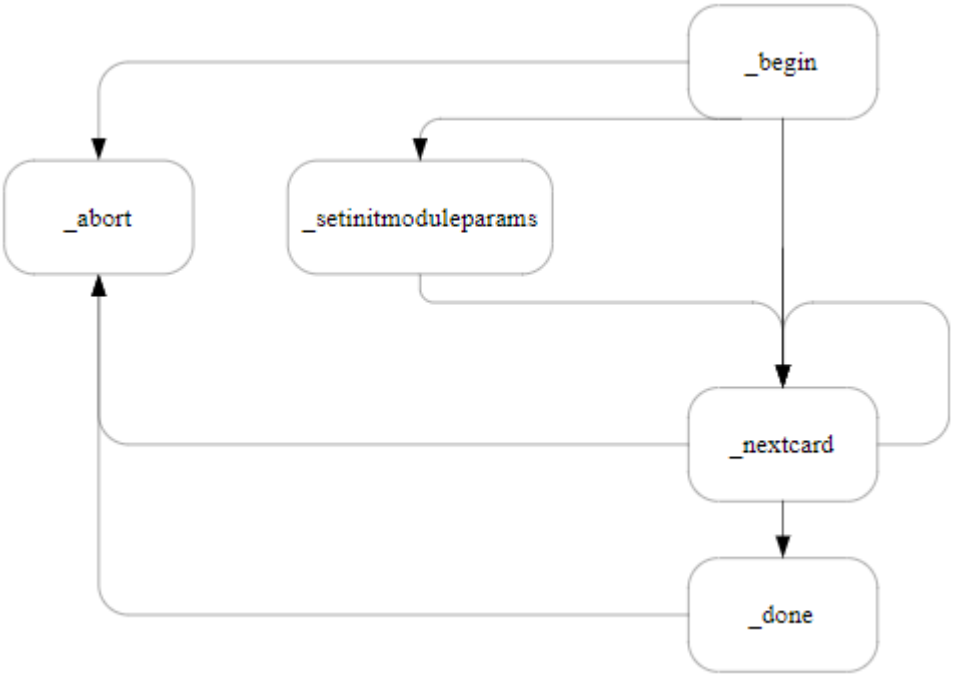
Destroys a Security World loading context.

```
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`.

10.2.14.2. NFKM_loadworld_begin

Initializes an operation to program a module with an existing Security World. The following diagram illustrates the paths through the `NFKM_loadworld` process:




```
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.

10.2.14.3. 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`.

10.2.14.4. 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

10.2.14.5. 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.

10.2.15. NFKM_makecardset_*

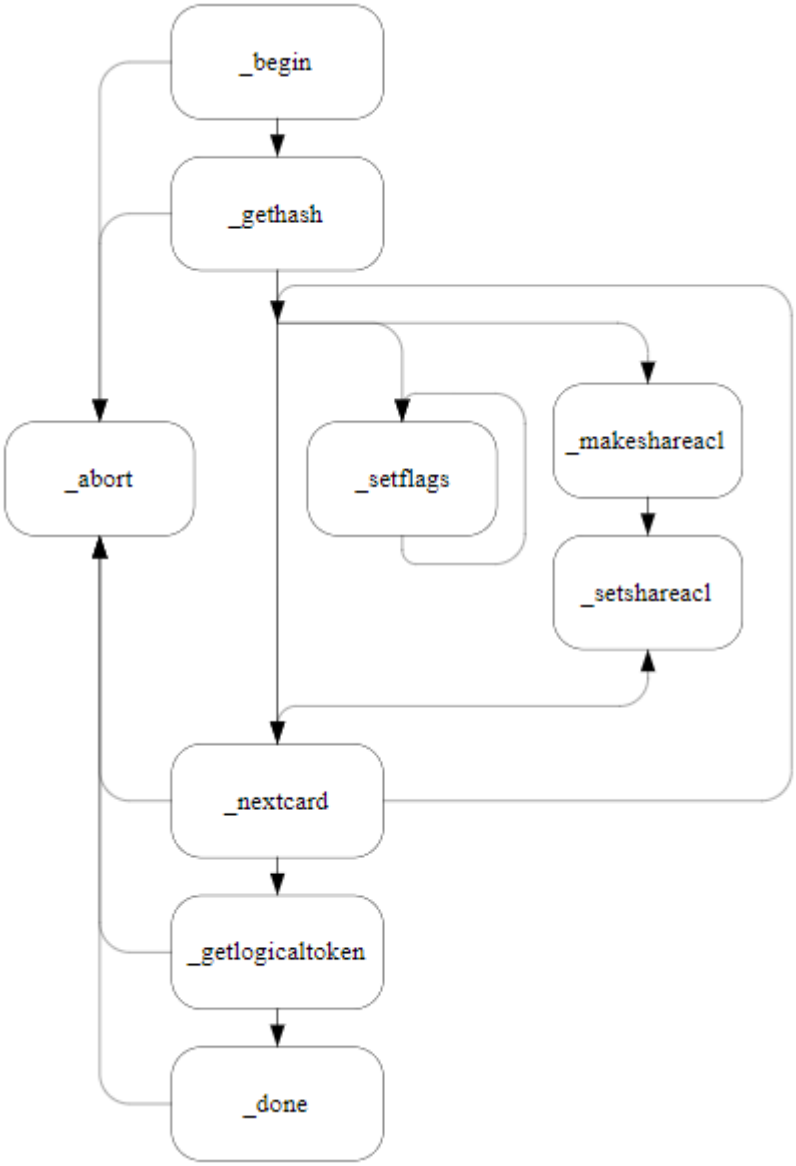
10.2.15.1. NFKM_makecardset_abort

This function aborts the creation of a card set:

```
void NFKM_makecardset_abort(
                                    NFKM_MakeCSHandle state
);
```

10.2.15.2. NFKM_makecardset_begin

This function prepares to make a new card set. The following diagram illustrates the paths through the `NFKM_makecardset`:





`NFKM_makecardset_setflags`, `NFKM_makecardset_makeshareacl` or `NFKM_-makecardset_setshareacl` are not recommended for normal use



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 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 loading of 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 Level 3 Security Worlds.

10.2.15.3. 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
);
```

10.2.15.4. 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
);
```

10.2.15.5. 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.

10.2.15.6. 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.

10.2.15.7. NFKM_makecardset_nextcard



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.

10.2.15.8. 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.



It is best to avoid using this function; instead, pass appropriate `Card-Set_flags_` to `NFKM_makecardset_begin` and it will automatically set appropriate share flags.

10.2.15.9. 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.



It is best to avoid using this function; instead, pass appropriate `Card-Set_flags_` to `NFKM_makecardset_begin` and it will construct and use an appropriate ACL.

10.2.16. NFKM_newkey_*

10.2.16.1. 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_make-blobs`.

    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_make-blobs`.

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, _UseAsCertifi-

cate, 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.

10.2.16.2. 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.



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.

10.2.16.3. 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.

10.2.16.4. 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;
```

10.2.16.5. 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`.

10.2.17. NFKM_operatorcard_changepp



This function has been superseded by the `NFKM_changepp` function, see NFKM_changepp.

This function changes the passphrase 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 passphrase.

10.2.18. NFKM_operatorcard_checkpp



This function has been superseded by the `NFKM_checkpp` function, see NFKM_checkpp.

This function checks the passphrase 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 passphrase.

10.2.19. 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`.

10.2.20. 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.

10.2.21. NFKM_replaceacs_*

10.2.21.1. NFKM_replaceacs_abort

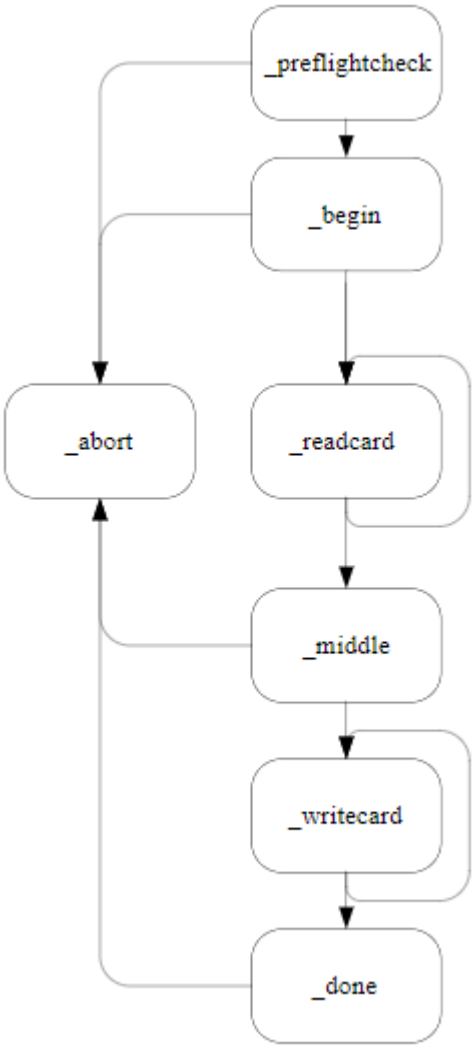
Destroys an admin card replacement context.

```
void NFKM_replaceacs_abort(
                                    NFKM_ReplaceACSHandle rah
);
```

`NFKM_ReplaceACSHandle rah` is the job handle returned by `NFKM_replaceacs_begin`

10.2.21.2. NFKM_replaceacs_begin

Starts a job to replace the Administrator Card Set. The following diagram illustrates the paths through the `NFKM_replaceacs` process:




```
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.

10.2.21.3. 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`

10.2.21.4. 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

10.2.21.5. 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`

10.2.21.6. 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).

10.2.21.7. 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

10.2.21.8. 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

11. OpenSSL with NFKM Engine

11.1. Quick usage

Assuming you have a key named `sslkey` protected by OCS `sslOCS` and the current working directory contains your `index.html` file. Ensure that the environment variable `OPENSSL_ENGINES` is defined as `$NFAST_HOME/openssl/lib/engines-3/0` on Linux or `%NFAST_HOME%\openssl\lib\engines-3\0` on Windows before running the following. The command assumes that you already have a certificate. If you don't have one, see the command in Testing with a self-signed certificate.

The command has been wrapped for readability but should be written on one line.

```
preload -c sslOCS openssl s_server -engine nfkm -keyform engine
    -key simple_sslkey -port 4433 -cert <path-to-certificate> -HTTP
```

You can verify that this works with cURL command in a different terminal window:

```
curl https://www.example.com:4433/index.html
```

The output should print the contents of your `index.html` file.

You can see the server using the HSM to make signatures by running `openssl` with `NFLOG_SEVERITY=debug1` set.

11.2. Testing with a self-signed certificate

The following assumes there is an existing OCS called `exampleocs` present in the Security World and that the environment variable `OPENSSL_ENGINES` is defined as `$NFAST_HOME/openssl/lib/engines-3/0` on Linux or `%NFAST_HOME%\openssl\lib\engines-3\0` on Windows. The commands have been wrapped for readability but should each be written on one line.

Verify that the NFKM engine works with `openssl`.

1. Create a key using the `generatekey` utility.

   ```
   generatekey simple protect=token recovery=yes ident=ssltest
       plainname=ssltest type=RSA size=2048 pubexp='' nvram=no
   ```

2. Create a self-signed certificate for the key using `openssl req`.

```
preload -c exampleocs openssl req -x509 -engine nfkm -keyform engine
    -subj /CN=www.example.com -addext subjectAltName=DNS:www.example.com
    -key simple_ssltest -new > ssltest.pem
```

3. `openssl s_server` includes an example web server, which can be told to use the NFKM engine with the newly created key and certificate.
   a. Make a new directory in your current directory with a new file called `index.html` containing the text `<h1>Sample page</h1>`.
   b. From the new directory, run the following command.

```
preload -c exampleocs openssl s_server -engine nfkm -keyform engine
    -key simple_ssltest -port 4433 -cert ../ssltest.pem -HTTP
```

You can now request the page using cURL in a different terminal window.

```
curl --insecure https://www.example.com:4433/index.html
```

This should print `<h1>Sample page</h1>`.

You can see the server using the HSM to make signatures by running `openssl` with `NFLOG_SEVERITY=debug1` set.

11.3. Private key operations

NFKM engine implements select private key operations outlined in Key type and mechanism support. The public and symmetric key operations are left to the OpenSSL software implementation.

The commands used for signing, verifying, encryption and decryption with NFKM engine are outlined below.

11.3.1. Signing/verifying (DSA/ECDSA/RSA)

```
# Sign
openssl dgst -engine nfkm -keyform engine -out <sigfile> <shaflag> \
  -sigopt rsa_padding_mode:<padding mode> -sign <keyname> <infile>

# Export public key
openssl pkey -engine nfkm -inform engine -pubout -in <keyname> -out <pubkeyfile>

# Verify
openssl dgst -verify <pubkeyfile> <shaflag> -sigopt -signature <sigfile> \
  rsa_padding_mode:<padding mode>
```

keyname	Key to use, in the format `appname_ident`.
infile	Input file to sign.
sigfile	Signed file.
shaflag	Hash mechanism flag. One of `-sha1`, `-sha224`, `-sha256`, `-sha384`, `-sha512`. Default is `-sha256`.
padding mode	Padding mode for RSA only (omit `rsa_padding_mode` otherwise). One of `x931`, `pkcs1`, `pss`.
pubkeyfile	Exported public key data.

11.3.2. Encryption/decryption (RSA)

```
# Encrypt
openssl pkeyutl -encrypt -engine nfkm -keyform engine -inkey <keyname> \
  -in <infile> -out <encfile> -pkeyopt rsa_padding_mode:<padding mode>

# Decrypt
openssl pkeyutl -decrypt -engine nfkm -keyform engine -inkey <keyname> \
  -in <encfile> -pkeyopt rsa_padding_mode:<padding mode>
```

keyname	Key to use, in the format `appname_ident`.
infile	Plaintext file to encrypt.
encfile	Encrypted file.
padding mode	Padding mode. One of `oaep`, `pkcs1`.

11.3.3. Key type and mechanism support

The following table lays out the key types and mechanisms which are supported by the NFKM engine. Details about the key size and curve restrictions in FIPS 140 Level 3 mode can be found in the nShield Support for Cryptographic Algorithms application note.

Feature		Unrestricted	FIPS 140 Level 3	Common Criteria CMTS
DSA signature	with SHA-1	✓	□	✓
	with SHA-224	✓	Restricted	✓
	with SHA-256	✓	Restricted	✓
	with SHA-384	✓	Restricted	✓
	with SHA-512	✓	Restricted	✓
ECDSA signature[1]	with SHA-1	✓	□	✓
	with SHA-224	✓	Restricted	✓
	with SHA-256	✓	Restricted	✓
	with SHA-384	✓	Restricted	✓
	with SHA-512	✓	Restricted	✓
RSA signature[2]	with SHA-1	✓	□	✓
	with SHA-224	✓	Restricted	✓
	with SHA-256	✓	Restricted	✓
	with SHA-384	✓	Restricted	✓
	with SHA-512	✓	Restricted	✓
RSA signature padding mode[2]	PKCS#1v1.5 padding (RSA_PKCS1_-PADDING)	✓	Restricted	✓
	PSS padding (RSA_PKCS1_PSS_-PADDING)	✓	Restricted	✓
	ANSI X9.31 padding (RSA_X931_-PADDING)	✓	Restricted	✓
RSA decryption padding mode[2]	PKCS#1v1.5 padding (RSA_PKCS1_-PADDING)	✓	□	✓
	OAEP padding (RSA_PKCS1_OAEP_PADDING)	✓	Restricted	✓

[1] ECDSA with all nShield named curves (subject to FIPS 140 Level 3 restrictions)

[2] Not possible in FIPS 140 Level 3 mode with the ECp521mAES ciphersuite

11.4. Common problems

11.4.1. invalid engine "nfkm"

Ensure the environment variable `OPENSSL_ENGINES` is defined as `$NFAST_HOME/openssl/lib/engines-3/0` on Linux or `%NFAST_HOME%\openssl\lib\engines-3\0` on Windows.

11.4.2. unable to load server certificate private key file

Ensure that `preload` is used when using operations with an OCS-protected or softcard-protected key:

- `preload -c ocsname openssl [⋯]`
- `preload -s softcardname openssl [⋯]`

12. 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



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.



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.

12.1. 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.

12.1.1. 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, for example ID_{KA} or ID_{KT}, 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}.

12.1.1.1. Arguments

```
struct M_Cmd_ClearUnit_Args {
   M_ModuleID module;                                ModuleID
};
```

12.1.1.2. 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).

12.1.1.3. 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.

12.1.2. 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 for more about the `ClearUnit` command.

12.1.2.1. Arguments

```
bitmap:                         flags
       harmless: 16-
ModuleID                        module      [<module to be reset>]
ModuleMode                      mode        [<desired module mode>]
```

- Flags are not currently used.

12.1.2.2. Module mode settings

The following desired module mode settings are available:

```
ModuleMode Default =0
ModuleMode Maintenance =1
ModuleMode Operational =2
ModuleMode Initialisation =3
```

12.1.2.3. 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 soft ware mode changes, then `@ref Status ModuleMonitorNotSupported` is returned.



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.

12.1.3. ModExp

Operational state	initialization state

This command performs modular exponentiation on parameters passed by the client.

12.1.3.1. Arguments

```
struct M_Cmd_ModExp_Args {
  M_Bignum a;                          A base
  M_Bignum p;                          P power
  M_Bignum n;                          N modulus
};
```

12.1.3.2. Reply

```
struct M_Cmd_ModExp_Reply {
M_Bignum r;
};
```

where $r = A^P \bmod N$

12.1.4. ModExpCrt

Operational state	initialization state

This command performs modular exponentiation on parameters passed by the client. `ModEx`

`pCrt` uses the Chinese Remainder Theorem to reduce the time it takes to perform the operation.

12.1.4.1. 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
};
```

12.1.4.2. Reply

Uses `M_Cmd_ModExp_Reply`.

12.1.4.3. Notes

It is assumed that `P >= Q`.

12.2. 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`.

12.2.1. 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.

12.2.1.1. 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`

12.2.1.2. Reply

The reply structure for this command is empty.

12.2.2. 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.



Channel operations are only available for symmetric algorithms.

12.2.2.1. 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 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.

12.2.2.2. 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`.

  The server will destroy the channel automatically when the last con nection 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 per-

form data transfer.

12.2.3. 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.



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.

12.2.3.1. 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)

12.2.3.2. 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.

12.2.4. 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.

12.2.4.1. 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 ID_{KA}.
- `M_Mech mech`: See Mechanisms 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.

12.2.4.2. Reply

```
struct M_Cmd_Decrypt_Reply {
   M_PlainText plain;
}
```

12.2.5. 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.

12.2.5.1. 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;
```

- The following `flag` is defined: `Cmd_DeriveKey_Args_flags_WorldHashMech` Indicates that the hash mechanism for Security World keys will be used for identifying keys. By enabling the `Cmd_DeriveKey_Args_flags_WorldHashMech` flag, keys shall be identified by the selected world hash mechanism. See *DeriveKey and DeriveKeyEx*.
- `M_DeriveMech mech`

  See *Derive Key Mechanisms* for information on supported mechanisms.

- `int n_keys`

  This value is the number of keys that have been supplied in the key table.

- `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)

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.



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*.

```
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_AESKeyUnwrap_DKParams aeskeyunwrap;
M_DeriveMech_RawDecryptZeroPad_DKParams rawdecryptzeropad;
M_DeriveMech_ECCMQV_DKParams eccmqv;
M_DeriveMech_ECDHKA_DKParams ecdhka;
M_DeriveMech_ECIESKeyUnwrap_DKParams ecieskeyunwrap;
M_DeriveMech_ECIESKeyWrap_DKParams ecieskeywrap;
M_DeriveMech_ConcatenateBytes_DKParams concatenatebytes;
M_DeriveMech_RawEncrypt_DKParams rawencrypt;
M_DeriveMech_NISTKDFmCTRpRijndaelCMACr32_DKParams nistkdfmctrprijndaelcmacr32;
M_DeriveMech_RawEncryptZeroPad_DKParams rawencryptzeropad;
};
```

12.2.5.2. 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.

12.2.5.3. 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, Entrust 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.

12.2.6. 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 ID_{KA}
- a logical token by specifying an 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 ID_{KA} that has not been issued previously by the nShield server or that has already been destroyed.



An 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) cannot be destroyed. Attempts to do so will return an `ObjectInUse` error. Destroy the merged `KeyID` first.

12.2.6.1. Arguments

```
struct M_Cmd_Destroy_Args {
M_KeyID key;
};
```

`M_KeyID key` can be any object with an `M_KeyID`, such as an ID_{KA}, an ID_{KT}, or the SEE World's *KeyID*.

12.2.6.2. Reply

The reply structure for this command is empty.

12.2.7. 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.

12.2.7.1. Arguments

```
struct M_Cmd_Duplicate_Args {
   M_KeyID key;
};
```

`M_KeyID key` is ID_{KA}.

12.2.7.2. Reply

```
struct M_Cmd_Duplicate_Reply {
   M_KeyID newkey;
```

```
};
```

`M_KeyID newkey` is ID_{KA2}.

12.2.8. 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.

12.2.8.1. 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 ID_{KA}.
- `M_Mech mech`

  See Mechanisms 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.

12.2.8.2. Reply

```
struct M_Cmd_Encrypt_Reply {
   M_CipherText cipher;
};
```

12.2.9. Export

Operational state, initialization state	Requires a ClientID

This command is used to extract key material in plain text.



Most private key objects should have an ACL (or ACLs) that forbid the reading of this data in plain text.

12.2.9.1. Arguments

```
struct M_Cmd_Export_Args {
   M_KeyID key;
};
```

12.2.9.2. Reply

```
struct M_Cmd_Export_Reply {
   M_KeyData data;
};
```

12.2.10. 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 authen ticate the monitor.

Use the `fwcheck` command-line utility to perform this operation.

12.2.11. FormatToken

Operational state, initialization state	May require a KNSO certificate

This command initializes a smart card.

12.2.11.1. 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.

12.2.11.2. Reply

The reply structure for this command is empty.

12.2.12. 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`.

12.2.12.1. 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^{-1}_{ML}
  - `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_HMACSHA3b224
- KeyType_HMACSHA3b256
- KeyType_HMACSHA3b384
- KeyType_HMACSHA3b512
- 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_Twofish
- KeyType_Void KeyType_Wrapped Created by DeriveKey
- KeyType_Any Do not use for key generation
- KeyType_None Do not use for key generation



When generating a key pair, you must specify the key type for the private half of the key pair.



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.
- `M_KeyType_DSAPrivate_GenParams dsaprivate`. See DSA.
- `M_KeyType_Random_GenParams random`. See Random.
- `M_KeyType_DSAComm_GenParams dsacomm`. See DSA.
- `M_KeyType_DHPrivate_GenParams dhprivate`. See Diffie-Hellman and ElGamal.
- `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.

- `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.

12.2.12.2. 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 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.

12.2.12.3. 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.

12.2.13. 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.

12.2.13.1. 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.

12.2.13.2. Reply

```
struct M_Cmd_GenerateLogicalToken_Reply {
   M_KeyID idkt;
   M_TokenHash hkt;
};
```

- `M_KeyID idkt` is ID_{KT}
- `M_TokenHash hkt` is H_{KT}

12.2.14. GetChallenge

Operational state, initialization state	Requires a ClientID

The `GetChallenge` command returns a nonce that is used to build a fresh certificate. See Certificates. `GetChallenge` is also used during impath setup.

12.2.14.1. Arguments

```
struct M_Cmd_GetChallenge_Args {
   M_ModuleID module;
};
```

12.2.14.2. Reply

```
struct M_Cmd_GetChallenge_Reply {
   M_KMHash nonce;
};
```

12.2.15. 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. 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.

12.2.15.1. Arguments

```
struct M_Cmd_GetKML_Args {
   M_ModuleID module;
};
```

12.2.15.2. Reply

```
struct M_Cmd_GetKML_Reply {
   M_KeyID idka;
};
```

`M_KeyID idka` is ID_{KA} for K_{ML}

12.2.16. 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.



The program should treat tickets as opaque objects. nShield reserves the right to change the structure of tickets at any time.

12.2.16.1. 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_AnyKer`

nelClient 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;
};
```

12.2.16.2. 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.

12.2.17. 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.

12.2.17.1. 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.
- `M_PlainText plain` This must be in the format `M_PlainTextType_Bytes_Data`.

12.2.17.2. Reply

```
struct M_Cmd_Hash_Reply {
   M_CipherText sig; Hash
};
```

12.2.18. 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.



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.

12.2.18.1. 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.

12.2.18.2. 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.)

12.2.19. 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.

12.2.19.1. 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.

12.2.19.2. Reply

The reply structure for this command is empty.

12.2.20. ImpathReceive

Operational state, initialization state	Requires a ClientID

This command decrypts a user-data message that was encrypted using an impath.

12.2.20.1. 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.

12.2.20.2. Reply

```
struct M_Cmd_ImpathReceive_Reply {
   M_ByteBlock data;
};
```

`M_ByteBlock data` is a recovered plain text message.

12.2.21. 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.

12.2.21.1. 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.

12.2.21.2. 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.

12.2.22. 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 K_M, including K_{M0}. It also erases the module's signing key, K_{ML}, and the hash of the Security Officer's keys, H_{KNSO}. It does not erase the long-term K_{LF} key. It then generates a new K_{ML} and K_{M0}.

In order to use the module after it has been initialized, you must set a new Security Officer's key.



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 `New-Client` command after the module enters the Initialization state.

12.2.22.1. Arguments

```
struct M_Cmd_InitialiseUnit_Args {
   M_ModuleID module;
};
```

12.2.22.2. Reply

The reply structure for this command is empty.

12.2.23. 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 ID_{KA} handle will be generated and returned.

For K_M blobs, the required K_M value must be present in the module's internal storage.

For K_T blobs, the logical token containing K_T must be "present". This is not possible if the K_M associated with that K_T is not present in the module. See GenerateLogicalToken and LoadLogicalToken.

For the archival key blobs K_i or K_{AR}, the appropriate key object must be loaded.

12.2.23.1. 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`.

12.2.23.2. Reply

```
struct M_Cmd_LoadBlob_Reply {
    M_KeyID idka;
};
```

`M_KeyID idka` is ID_{KA}.

12.2.24. 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).

12.2.24.1. 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 H_{KT}
- `M_KMHash hkm` is the 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.

12.2.24.2. Reply

```
struct M_Cmd_LoadLogicalToken_Reply {
   M_KeyID idkt;
};
```

`M_KeyID idkt` is the ID_{KT}.

12.2.25. 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 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 K_M key, the relevant key must be stored internally within the module.

For a 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.

12.2.25.1. 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.

  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.

- `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.

12.2.25.2. Reply

```
struct M_Cmd_MakeBlob_Reply {
   M_ByteBlock blob;
};
```

`M_ByteBlock blob` is a `KeyBlob`.

12.2.26. 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.

12.2.26.1. 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 ID_{KA}.

12.2.26.2. Reply

```
struct M_Cmd_MergeKeyIDs_Reply {
   M_KeyID newkey;
};
```

`M_KeyID newkey` is ID_{KA}

12.2.26.3. 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.

12.2.27. ReadShare

Operational state, initialization state	Requires a ClientID

This command is used to assemble a logical token from shares.



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.

12.2.27.1. 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.

    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 possi ble 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 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.

12.2.27.2. 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.

12.2.27.3. 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_TokenReassembly-Failed` 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 ID_{KT}. If the client closes before the token has been assembled, the server automatically issues the `Destroy` command.

12.2.28. RedeemTicket

Operational state, initialization state	Requires a ClientID

This command gets a `KeyID` in return for a key ticket.

12.2.28.1. 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`.

12.2.28.2. Reply

```
struct M_Cmd_RedeemTicket_Reply {
   M_KeyID obj;
};
```

`M_KeyID obj` is the new KeyID for this object.

12.2.29. RemoveKM

Operational state, initialization state	Requires a ClientID 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.



K_{M0} cannot be deleted.

12.2.29.1. 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}.

12.2.29.2. Reply

The reply structure for this command is empty.

12.2.30. RSAImmedSignDecrypt

Operational state, initialization state

This command performs RSA decryption by using an RSA private key that is provided in plain text.

12.2.30.1. 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}

12.2.30.2. Reply

```
struct M_Cmd_RSAImmedSignDecrypt_Reply {
   M_Bignum r;
};
```

`M_Bignum r` is plain text .

12.2.30.3. Notes

The plain text and cipher text are in the nShield bignum format.

No padding is done.

12.2.31. RSAImmedVerifyEncrypt

Operational state, initialization state

This command performs RSA encryption with an RSA public key provided in plain text.

12.2.31.1. 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

12.2.31.2. Reply

Uses `M_Cmd_RSAImmedSignDecrypt_Reply`.

12.2.31.3. Notes

The plain text and cipher text are in nShield bignum format.

No padding or unpadding is performed.

12.2.32. 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.

12.2.32.1. 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 ID_{KA}.
- `M_ACL *acl` is the new ACL for the key.

12.2.32.2. Reply

The reply structure for this command is empty.

12.2.32.3. 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`.



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.

12.2.33. SetKM

Operational state, initialization state	Requires a ClientID May require a KNSO certificate "Privileged" users only

This command allows a key object to be stored internally as a module key (K_M) value. The K_M value is derived from the key material given by ID_{KA}. The ACL and other information associated with ID_{KA} are not stored.

12.2.33.1. 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 ID_{KA}.
- K_A must be a DES3 key with `UseAsKM` permission.

12.2.33.2. Reply

The reply structure for this command is empty.

12.2.33.3. Notes

If you attempt to set as a K_M a key that has the same hash as an existing K_M, then `SetKM` will overwrite the existing module key with the new key. If you are attempting to overwrite K_{M0}, the command will return `Status_AccessDenied`.

12.2.34. SetNSOPerms

Initialization state only	Requires a ClientID "Privileged" users only

The SetNSOPerms command stores the key hash H_{KA}, which is returned by GetKeyInfo as the new Security Officer's key.

It also determines which operations require a KNSO certificate.



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). After it is set, the Security Officer's key can only be changed by completely reinitializing the module.

12.2.34.1. 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 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.

      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, 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



In order to fully comply with FIPS 140 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`

12.2.34.2. Reply

The reply structure for this command is empty.

12.2.34.3. Notes



Modules that are supplied by nShield are initialized with no operations that require K_{NSO} certificates. This means that the key whose hash is installed as H_{KNSO} is irrelevant.

12.2.35. SetRTC

Operational state, initialization state	Requires an SEE-Ready module May require a KNSO certificate "Privileged" users only

12.2.35.1. 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.

12.2.35.2. Reply

The reply structure for this command is empty.

12.2.36. Sign

Operational state, initialization state	Requires a ClientID

This command signs a message with a stored key.

For information on formats, see Encrypt.

`Sign` pads the message as specified by the relevant algorithm, unless you use plaintext of the type `Bignum`.



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.

12.2.36.1. 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 ID_{KA}.

12.2.36.2. Reply

```
struct M_Cmd_Sign_Reply {
   M_CipherText sig;
};
```

12.2.37. 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.

12.2.37.1. 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`

12.2.37.2. 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.

12.2.38. 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.

12.2.38.1. 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

12.2.38.2. Reply

The reply structure for this command is empty.

12.2.39. UpdateMergedKey

All non-error states	Processed by the nShield Server.

This command allows a merged key set to be manipulated, listed, or both.

12.2.39.1. 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.

  Including a key in this list deletes all copies of the specified key.

12.2.39.2. 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.

12.2.39.3. 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.

12.2.40. 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.

12.2.40.1. 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.

12.2.40.2. Reply

The reply structure for this command is empty.

12.2.41. 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.

12.2.41.1. 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 passphrase.

- `Cmd_WriteShare_Args_flags_UseProtectedPINPath`

  Set this flag if the token reads a passphrase 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.



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.



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.

12.2.41.2. Reply

The reply structure for this command is empty.

12.3. 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.

12.3.1. 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.

12.3.1.1. Arguments

```
struct M_Cmd_ExistingClient_Args {
    M_Cmd_ExistingClient_Args_flags flags;
    M_ClientID client;
};
```

- No `flags` are defined.
- `M_ClientID client`: R_{SC}

12.3.1.2. Reply

```
struct M_Cmd_ExistingClient_Reply {
    M_Cmd_ExistingClient_Reply_flags flags;
};
```

No `flags` are defined.

12.3.2. 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`.

12.3.2.1. Arguments

```
typedef struct M_Cmd_NewClient_Args {
   M_Cmd_NewClient_Args_flags flags;
};
```

No `flags` are defined.

12.3.2.2. Reply

```
struct M_Cmd_NewClient_Reply {
    M_Cmd_NewClient_Reply_flags flags;
    M_ClientID client;
};
```

- No `flags` are defined.
- `M_ClientID client`: R_{SC}

13. Transaction IDs

13.1. Introduction

Transaction IDs, also known as correlation IDs, are identifiers that allow a request to be traced through different layers, components, and log files of a system.

nCore supports Transaction IDs in conjunction with the Audit Logging scheme, provided in firmware versions v13.5 and later, for nShield Solo XC, Connect XC, 5s, and 5c HSMs.

Transaction IDs are represented in nCore as UTF-8 strings that are permitted to be up to 88 bytes in length, not including the terminator. This is sufficient to represent the base64 of a SHA512 or SHA3-512 hash.

By using human-readable strings, you can choose how the IDs appear when printed in nCore client logs and audit logs. You can also use this string to encode data such as sequence numbers, UUIDs, cryptographic hashes, or hex or base64 encodings of user-supplied data.

13.2. Limitations

Some limitations are present in this release.

- Sending commands with Transaction IDs set to HSMs running firmware versions older than v13.5 is not supported.
  - If the attempt is made, commands will fail with error `Status_UnknownFlag`.
- Transaction IDs are preserved where the command is directly forwarded from the client to the HSM. Where commands are implemented internally by the Security World software, the linkage might not be preserved.
  - In particular, this can apply to `Cmd_Destroy` when sent from client-side applications, as objects are reference counted in the hardserver, so the actual HSM `Cmd_Destroy` command is issued by the hardserver when the reference count reaches 0. This limitation does not apply to CodeSafe, which does not use hardserver.
  - Client-side code sending large commands, such as `Cmd_Decrypt`, with a symmetric key on a large ciphertext might be automatically split by library code into multiple commands of smaller buffers. Linkage is not preserved when the Transaction ID is set directly on the `M_Command` object, but if the Transaction ID is set on the connection, as a thread-local, or on any of the other supported interfaces besides setting on command directly, then it will be.

13.3. Unicode Notes

Because the Transaction ID strings can be UTF-8, the following filters and features are present to mitigate potential issues when displaying them in JSON or text output in the `nshieldaudit` tool. Where filtering occurs, one or more consecutive filtered characters are replaced with an underscore _ character as the replacement character.

- Control characters other than new-line, carriage-return, and tab are filtered.
- Whitespace, including non-ASCII Unicode whitespace, is filtered when outputting in the single-line text representation, but is not filtered in the full JSON output.
- Bidirectional text formatting characters are filtered.
- The option to restrict to only ASCII characters can be enabled by setting the environment variable `NSHIELDAUDIT_ASCII_ONLY=1` in the environment of the `nshieldaudit` tool when performing the `export`.

Additional options may be provided for controlling permitted characters in a future release.

13.4. Setting Transaction IDs

Transaction IDs are associated with an nCore command submitted by client code.

nShield client-side software and libraries support use of the `NFAST_TRANSACTION_ID` environment variable as a way to inject a Transaction ID into any operation.

It can also be set programmatically in nCore client libraries, as described in the following sections. In each case, library code will automatically copy and truncate the string to the 88 byte limit.

13.4.1. nCore C (Generic Stub)

When a command is submitted, for example by using `NFastApp_Submit` or `NFastApp_Transact`, library code searches for a Transaction ID that has been set by the following functions, in the following order of precedence:

1. Set directly on an `M_Command` object using the helper functions `NFastApp_SetTransactionID` for heap-allocation, where command will be freed, or `NFastApp_SetTransactionID_NoFree` where all memory is on stack.
2. A user-supplied callback function, "upcall", for retrieving the Transaction ID from user call context or transaction context that has been provided to the library when `NFastApp_InitEx` was called.

3. Thread-local value set via `NFastApp_SetThreadTransactionID`.
4. Set for a given `NFastApp_Connection` using `NFastApp_SetConnTransactionID`.
5. Set for a given `NFast_AppHandle` using `NFastApp_SetAppTransactionID`.
6. `NFAST_TRANSACTION_ID` environment variable.

Corresponding "Get" functions also exist for each of these "Set" functions.

The caller provides Transaction ID strings as `const char *` objects that can contain UTF-8 characters.

13.4.2. SEElib (CodeSafe CSEE)

Helper functions `SEElib_SetTransactionID` and `SEElib_SetTransactionID_NoFree` provide the corresponding functionality to their `NFastApp_*` equivalents for setting a UTF-8 Transac tion ID string on an `M_Command` directly.

Other mechanisms of setting the Transaction ID are not provided in this case. This includes the `NFAST_TRANSACTION_ID` environment variable.

13.4.3. nCore Python (nfpython)

When a command is submitted, for example by using the `submit()` or `transact()` methods of an `nfpython.connection` object, library code searches for a Transaction ID that has been set by the following functions, in the following order of precedence:

1. Set directly on an `nfpython.Command` object using the `nfpython.set_transaction_id()` function.
2. Thread-local value set using the `nfpython.set_thread_transaction_id()` function.
3. Set for a given `nfpython.connection` object using the `set_transaction_id()` method of the object.
4. `NFAST_TRANSACTION_ID` environment variable.

Corresponding "Get" functions also exist for each of these "Set" functions.

13.4.4. nCore Java (nfjava)

When a command is submitted, for example by using the `NFConnection.submit()` or `NFConnection.transact()` methods, library code searches for a Transaction ID that has been set by the following functions. It observes the following order of precedence:

1. Set directly on an `M_Command` object using the static helper function `NFUtils.setTransactionID()`.
2. Thread-local value set via static helper function `NFConnection.setThreadTransactionID`.
3. Set for a given `NFConnection` instance using `NFConnection.setTransactionID` method of the connection object.
4. `NFAST_TRANSACTION_ID` Java system property.
5. `NFAST_TRANSACTION_ID` environment variable.

Corresponding "Get" functions also exist for each of these "Set" functions.

13.4.5. Higher-level APIs

The thread-local Transaction ID setting function `NFastApp_SetThreadTransactionID`, and its Python and Java equivalents, does not take any library handles as arguments. This means that, in many cases, it can be called by higher-level code to provide the Transaction ID without having to directly modify or pass through to lower layers.

Because the storage for this function is thread-local, it can safely be used in a multi-threaded application. `NFastApp_SetThreadTransactionID` can also be be called with a NULL pointer to unset the thread-local, preventing subsequent code from unintentionally using the Transaction ID outside of the intended scope.

Setting the `NFAST_TRANSACTION_ID` environment variable also provides an implementation-agnostic and language-agnostic way to provide a Transaction ID. This can be done for a whole program, for example, where a whole request is self-contained, which might be the case with something like `signtool`. It can also be used dynamically in-process, because library code re-checks the environment variable each time an nCore command is submitted. Updating dynamically is only suitable when a single thread of the application is submitting nCore commands, because environment variables are process-wide.

Supporting additional setters for particular higher-level APIs like PKCS #11 or CNG, for example supporting setting the Transaction ID via an attribute or property on a relevant object or handle in the higher-level API, may be considered in future if there is enough demand and if existing interfaces are insufficient for user needs.

13.5. Transaction ID logging

13.5.1. Client debug logs

Transaction IDs appear in client-side debug logs as part of the traced nCore commands, as demonstrated in the following example produced from the nCore C Generic Stub library using the `NFLOG_SEVERITY=DEBUG1` environment variable.

The Transaction ID string in this example is `ExampleTransactionID1234`.

Generic Stub nCore Log

```
00:40:56 DEBUG1: NFastApp_Submit tag=17cf08be; conn=0x55851a740800; reply=0x55851a771e30; time=1732495256
  command.tag= 0x00000000 0
         .cmd= Sign
         .status zero
         .flags= sessioninfo_present 0x00000200
         .state absent
         .args.sign.flags= none 0x00000000
                   .key= 0x9e4fa111 2656018705 2656018705
                   .mech= RSApPKCS1
                   .plain.type= Bignum
                         .data.bignum.m= 128 bytes
                   .given_iv absent
         .certs absent
         .extractstate absent
         .sessionid absent
         .sessioninfo.v= 0x00000000 0
                     .flags= transaction_present 0x00000004
                     .transaction= "ExampleTransactionID1234"
```

13.5.2. nCore audit logs

Transaction IDs also appear in nCore audit logs emitted by firmware version v13.5 or later.

13.5.2.1. Text audit logs

nCore audit logs can be exported from the audit database, using the `nshieldaudit` command, in a condensed human-readable text format summarizing the most important information in one audit entry per line.

The Transaction ID is referenced with the `trID=` field in these logs, for example:

nCore text audit log

```
2024-11-25 00:40:56.254 idx=1940686 src=Host trID=ExampleTransactionID1234 cmd=Sign rc=OK obj=982
```

13.5.2.2. nCore JSON audit logs

nCore audit logs can be exported from the audit database in JSON format showing the full detail of the underlying nCore audit data. This is most useful if more context is needed or where a machine-readable format is needed for additional processing.

The following example displays the portion of a JSON audit segment that describes an audit event for a Sign command, similar to the one shown in the Client debug log:

nCore JSON audit segment


```
{
    "v": 0,
    "timestamp": 1732495256254,
    "source": "Host",
    "infos": [
    {
        "type": "Command",
        "body": {
        "flags": [
            "sessioninfo_present"
        ],
        "cmd": "Sign",
        "info": {},
        "sessioninfo": {
            "v": 0,
            "flags": [
            "transaction_present"
            ],
            "transaction": "ExampleTransactionID1234"
        },
        "status": "OK"
        }
    },
    {
        "type": "ObjectUse",
        "body": {
        "v": 0,
        "objid": 982,
        "action": {
            "type": "OpPermissions",
            "details": {
            "perms": [
                "Sign"
            ]
            }
        }
        }
    }
    ]
}
```