PKCS #11 openCryptoki for Linux HOWTO

Kristin Thomas

kristint@us.ibm.com

This HOWTO describes the implementation of the RSA Security Inc. Public Key Cryptographic Standard #11 (PKCS #11) cryptoki application program interface (API) on Linux (openCryptoki). The HOWTO explains what services openCryptoki provides and how to build and install it. Additional resources and a simple sample program are also provided.

Table of Contents

1. Copyright Notice and Disclaimer

Special Notices

This publication/presentation was produced in the United States. IBM may not offer the products, programs, services or features discussed herein in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the products, programs, services, and features available in your area. Any reference to an IBM product, program, service, or feature is not intended to state or imply that only IBM's product, program, service, or feature may be used. Any functionally equivalent product, program, service, or feature that does not infringe on IBM's intellectual property rights may be used instead.

Questions on the capabilities of non-IBM products should be addressed to suppliers of those products. IBM may have patents or pending patent applications covering subject matter in this presentation. Furnishing this presentation does not give you any license to these patents. Send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA. All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of a specific Statement of General Direction.

The information contained in this presentation has not been submitted to any formal IBM test and is distributed "AS IS." While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. The use of this information or the implementation of any techniques described herein is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The information contained in this document represents the current views of IBM on the issues discussed as of the date of publication. IBM cannot guarantee the accuracy of any information presented after the date of publication.

Any performance data in this document was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements quoted in this book may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this book may have been estimated through extrapolation. Actual results may vary. Users of this book should verify the applicable data for their specific environment.

A full list of U.S. trademarks owned by IBM may be found at http://www.ibm.com/legal/copytrade.shtml. Linux is a trademark of Linus Torvalds. Other company, product, and service names may be trademarks or service marks of

others.

2. Introduction

Cryptography is rapidly becoming a critical part of our daily lives. However, the application of cryptographic technology adds a heavy computational burden to today's server platforms. More systems are beginning to use specialized hardware to offload the computations, as well as to help ensure the security of secret key material. In this HOWTO we will discuss openCryptoki, an API that is rapidly becoming the defacto, non-Windows-platform industry standard for interfacing between cryptographic hardware and user space applications. In particular we will introduce the specifics of the PKCS #11 implementation to IBM cryptographic hardware (openCryptoki) that is available in open source form at IBM's developerWorks site.

3. What is openCryptoki?

openCryptoki is an implementation of the PKCS #11 API that allows interfacing to devices (such as a smart card, smart disk, or PCMCIA card) that hold cryptographic information and perform cryptographic functions. openCryptoki provides application portability by isolating the application from the details of the cryptographic device. Isolating the application also provides an added level of security because all cryptographic information stays within the device. The openCryptoki API provides a standard programming interface between applications and all kinds of portable cryptographic devices.

4. Architectural Overview

openCryptoki consists of a slot manager and an API for slot token dynamic link libraries (STDLLs). The slot manager runs as a daemon to control the number of token slots provided to applications, and it interacts with applications using a shared memory region. Each device that has a token associated with it places that token into a slot in the slot manager database. The shared memory region allows for proper sharing of state information between applications to help ensure conformance with the PKCS #11 specification.

4.1. Slot Manager

The Slot Manager Daemon (`pkcsslotd`) manages slots (and therefore tokens) in the system. A fixed number of processes can be attached to `pkcsslotd`, so a static table in shared memory is used. The current limit of the table is 1000 processes using the subsystem. The daemon sets up this shared memory upon initialization and acts as a garbage collector thereafter, helping to ensure that only active processes remain registered. When a process attaches

to a slot and opens a session, `pkcsslotd` will make future processes aware that a process has a session open and will lock out certain function calls, if the they need exclusive access to the given token. The daemon will constantly search through its region of shared memory and make sure that when a process is attached to a token it is actually running. If an attached process terminates abnormally, `pkcsslotd` will "clean up" after the process and free the slot for use by other processes.

4.2. Main API

The main API for the STDLLs lies in `/usr/lib/pkcs11/PKCS11_API.so`. This API includes all the functions as outlined in the PKCS #11 API specification. The main API provides each application with the slot management facility. The API also loads token specific modules (STDLLs) that provide the token specific operations (cryptographic operations and session and object management). STDLLs are customized for each token type and have specific functions, such as an initialization routine, to allow the token to work with the slot manager. When an application initializes the subsystem with the `C_Initialize` call, the API will load the STDLL shared objects for all the tokens that exist in the configuration (residing in the shared memory) and invoke the token specific initialization routines.

4.3. Slot Token Dynamic Link Libraries

STDLLs are plug-in modules to the main API. They provide token-specific functions beyond the main API functions. Specific devices can be supported by building an STDLL for the device. Each STDLL must provide at least a token specific initialization function. If the device is an intelligent device, such as a hardware adapter that supports multiple mechanisms, the STDLL can be thin because much of the session information can be stored on the device. If the device only performs a simple cryptographic function, all of the objects must be managed by the software. This flexibility allows the STDLLs to support any cryptographic device.

4.4. Shared Memory

The slot manager sets up its database in a region of shared memory. Since the maximum number of processes allowed to attach to `pkcsslotd` is finite, a fixed amount of memory can be set aside for token management. This fixed memory allotment for token management allows applications easier access to token state information and helps ensure conformance with the PKCS #11 specification.

5. Getting Started with openCryptoki

This section describes the system requirements for openCryptoki. It also explains where you can get openCryptoki and how to compile and install it.

5.1. System Requirements

A system running openCryptoki must provide support for at least one of the tokens that are provided with openCryptoki (4758 Cryptographic Coprocessor, ICA model 2058, and the System 390 Linux Cryptographic hardware support)

The following lists show the system requirements for running openCryptoki.

Hardware Requirements

- x86 PCI bus system or
- System 390 with Linux cryptographic hardware support

Software Requirements

- Linux operating system running at least a 2.2.16 kernel
- Device drivers and associated support libraries for the installed tokens (some of the header files from those distributions may also be required)

5.2. Obtaining openCryptoki

The openCryptoki project is hosted on the IBM developerWorks Web site (http://www-124.ibm.com/developerworks/projects/openCryptoki), and the source is available there. Updates providing bug fixes and functional enhancements area also available on the site.

5.3. Compiling and Installing openCryptoki

Assuming that the device support (and header files) for the required devices are on the system, then you can build openCryptoki by running **make** from the top level directory. Using the build target allows for the compilation of the

code only. The install target attempts to install the function onto the current system. Setting the `INSROOT` environment variable will allow you to target a different installation location. The clean target cleans the object files and other build output from the source tree.

If you are building openCryptoki for only one specific device, then the top-level Makefile for the source tree should be modified. Each component has its own Makefile, and the top level Makefile walks through each of its sub-directories building all Makefiles it finds. To build specific components, you do not have to use the top level Makefile. Running the **make -f** command in any given sub-directory will build that specific component. Components that have a Linux directory are built from that directory with the **make -f ../Makefile.linux** command.

6. Configuring openCryptoki

This section explains the various components and processes associated with configuring openCryptoki.

6.1. Configuration Utilities

There are three configuration utilities that are part of the openCryptoki kit. These are:

`/usr/lib/pkcs11/methods/pkcs11_startup`

> This utility is an executable shell script that builds the configuration information (`/etc/pkcs11/pkcs11_config_data`).

`/usr/lib/pkcs11/methods/4758_status`

> This utility is a binary executable that queries the 4758 Coprocessor to determine whether there is a shallow or deep firmware load on the card.

`/usr/lib/pkcs11/methods/pkcs_slot`

> This utility is a shell script that writes the configuration file to disk as `/etc/pkcs11/config_data`.

To simplify start-up, it is recommended that you create a shell script called `/etc/rc.pkcs11`. An example of this script is located in Section 6.3.

6.2. Configuration Files and Data Stores

Some of the STDLLs require local disk space to store persistent data, such as token information, personal identification numbers (PINs) and token objects. For the shallow 4758 token this information is stored in `/etc/pkcs11/4758shallow`, and for the ICA device it is stored in `/etc/pkcs11/lite`. Within each of these

directories is a sub-directory `TOK_OBJ` that contains the token object. Each of these STDLLs are limited to 2048 Public and 2048 Private token objects or the disk space allocated to `/etc`. The configuration scripts automatically create the required directories and assign the required permissions.

> **Note:** The `FindObject` function's performance is directly related to the number of objects that exist in the system. Applications should be implemented, so they don't perform such searches unnecessarily.

The `/etc/pkcs11/pk_config_data` file stores all of the configuration information for the current tokens within a system. When running, the subsystem's shared memory contains the actual usable configuration information.

6.3. Preparing the System for Configuration

In order to ensure proper access to openCryptoki tokens, only root and users who are members of the "pkcs11" group are allowed to access the subsystem. To properly configure the system, complete the following steps:

1. Create the "pkcs11" group by running the **groupadd pkcs11** command.

   > **Note:** This command is available on Red Hat Linux systems. Consult your Linux distribution's documentation for how to add the group if the **groupadd** command is not included in your implementation.

2. Add any user ids that will use the subsystem to the pkcs11 group by adding them to the `/etc/group` file or using the administrative GUI for your distribution.

   Alternatively, applications that are owned by the group and are setgid pkcs11 will also be able to run. The configuration detection scripts will create the data storage locations and assign the correct ownerships.

3. Create a shell script to start the subsystem at system boot.
4. Add this script to the local system start-up scripts. For example on a system running Red Hat Linux add the following as `/etc/rc.pkcs11` and add a call to it in `/etc/rc.local`.

```
#!/bin/bash
# Sample /etc/rc.pkcs11
# script for starting pkcs#11 at system IPL.
# this should be added to the end of the rc.local script, or
# what ever local startup script for your distribution applies
```

```
# Generate the configuration information

/usr/lib/pkcs11/methods/pkcs11_startup

# start the subsystem
/usr/sbin/pkcsslotd

#end of startup script
```

The configuration scripts of the subsystem interrogate the system, determining which devices are present, and putting them into the slot configuration automatically. Assuming that the devices are setup according to their specific instructions, the subsystem will reflect the configuration detected at system boot. If the devices are not found by the `pkcs11_startup` script, they will not be present. Usually the devices are not found because the modules for the device have not been loaded prior to the invocation of the `/etc/rc.pkcs11` script.

6.4. PKCS #11 Configuration Management Tools

openCryptoki provides a command line program (`/usr/lib/pkcs11/methods/pkcsconf`) to configure and administer tokens that are supported within the system. This program, provides the same capabilities as running the **pkcsconf -?** command. These capabilities include token initialization, security officer PIN initializing and changing, and user PIN initializing and changing. Operations that require a specific token must have the slot specified with the **-c** flag. You can view a list of tokens present within the system by specifying the **-s** option with no specific slot number. For example, the following code shows the options for the **pkcsconf** command and displays slot information for the system before token initialization:

```
[root@draeger /root]# /usr/lib/pkcs11/methods/pkcsconf -?
          /usr/lib/pkcs11/methods/pkcsconf: invalid option - ?
          usage:  /usr/lib/pkcs11/methods/pkcsconf [-itsmMIupP] [-c slotnumber -U user-
PIN -S SOPin -n newpin]
                  -i display PKCS11 info
                  -t display token info
                  -s display slot info
                  -m display mechanism list
                  -M display shared memory
                  -I initialize token
                  -u initialize user PIN
                  -p set the user PIN
                  -P set the SO PIN

          Output of the GetSlotInfo call
          [root@draeger /root]# /usr/lib/pkcs11/methods/pkcsconf -s
```

```
Slot #0 Info
        Description: Linux 2.4.2 Linux (ICA)
        Manufacturer: Linux 2.4.2
        Flags: 0x1
        Hardware Version: 0.0
        Firmware Version: 1.1



An un-initialized ICA token, the information is obtained via the C_GetTokenInfo call
[root@draeger /root]# /usr/lib/pkcs11/methods/pkcsconf -i
PKCS#11 Info
        Version 2.1
        Manufacuter: IBM AIX Software PKCS11
        Flags: 0x0
        Library Description: Meta PKCS11 LIBRARY
        Library VersionToken #0 Info:
        Label: IBM ICA  PKCS #11
        Manufacturer: IBM Corp.
        Model: IBM ICA
        Serial Number: 123
        Flags: 0x45
        Sessions: -1/-1
        R/W Sessions: -1/-1
        PIN Length: 4-8
        Public Memory: 0xFFFFFFFF/0xFFFFFFFF
        Private Memory: 0xFFFFFFFF/0xFFFFFFFF
        Hardware Version: 1.0
        Firmware Version: 1.0
        Time: 11:11:17 AM
```

This program also allows you to display the subsystem shared memory segment for debugging purposes. Additionally, the program provides some simple query functions on the tokens, such as token info and mechanism list.

Prior to applications using the openCryptoki subsystem, you must perform the following initialization steps (each step is followed by example code):

1. Initialize the token: Before you use a token, it must be initialized. The key part of this process is signing a unique label. The security officer PIN is needed (the default for all IBM-provided tokens is 87654321).

   Note: All PINs show up as stars on-screen when entered.

```
[root@draeger /root]# /usr/lib/pkcs11/methods/pkcsconf -I -c 0
Enter the SO PIN: 87654321
Enter a unique token label: DraegerICA

[root@draeger /root]# /usr/lib/pkcs11/methods/pkcsconf -t -c 0
Token #0 Info:
Label: DraegerICA

Model: IBM ICA
Serial Number: 123
Flags: 0x45
Sessions: -1/-1
R/W Sessions: -1/-1
PIN Length: 4-8
Public Memory: 0xFFFFFFFF/0xFFFFFFFF
Private Memory: 0xFFFFFFFF/0xFFFFFFFF
Hardware Version: 1.0
Firmware Version: 1.0
Time: 11:14:17 AM
```

2. Set the security officer PIN: Prudent management practice will have the security officer change the security officer PIN immediatly after initializing the token. This procedure prevents unknown individuals from re-initializing the device and removing all of the stored objects (for example keys and certificates).

```
[root@draeger /root]# /usr/lib/pkcs11/methods/pkcsconf -P -c 0
Enter the SO PIN: 87654321
Enter the new SO PIN: fredrules
Re-enter the new SO PIN: fredrules
```

3. Set the user PIN (by the security officer).

```
[root@draeger /root]# /usr/lib/pkcs11/methods/pkcsconf -u -c 0
Enter the SO PIN: fredrules
Enter the new user PIN: billybob
Re-enter the new user PIN: billybob
```

4. Change the user PIN (by the user).

```
[root@draeger /root]# /usr/lib/pkcs11/methods/pkcsconf -p -c 0
Enter user PIN: billybob
Enter the new user PIN: 12345678
Re-enter the new user PIN: 12345678
```

7. Components of openCryptoki

This section describes the different components of the openCryptoki subsystem.

7.1. Slot Manager Daemon

The slot manager daemon is an executable (`/usr/sbin/pkcsslotd`) that reads in `/etc/pkcs11/pk_config_data`, populating shared memory according to what devices have been found within the system. `pkcsslotd` then continues to run as a daemon. Any other applications attempting to use the subsystem must first attach to the shared memory region and register as part of the API initialization process, so `pkcsslotd` is aware of the application. If `/etc/pkcs11/pk_config_data` is changed, `pkcsslotd` must be stopped and restarted to read in the new configuration file. The daemon can be stopped by issuing the **kill <pkcsslotd config>** command. The daemon will not terminate if there are any applications using the subsystem.

7.2. `PKCS11_API.so`

This library contains the main API (`/usr/lib/pkcs11/PKCS11_API.so`) and is loaded by any application that uses any PKCS #11 token managed by the subsystem. Before an application uses a token, it must load the API and call `C_Initialize`, as per the PKCS #11 specification. The loading operation is performed by the application using the `dlopen` facilities.

7.3. Slot Token DLLs

Three STDLLs ship in the initial offering. These support the IBM 4758 PCI Cryptographic Coprocessor using two different firmware loads, and the IBM Cryptographic Accelerator (ICA), model 2058 or Leeds Lite card.

> **Note:** The compilation process attempts to build all of the tokens that are supported on the target platform, as well as all of the required support programs. If some of the headers and libraries are not present, those components will not be built.

7.3.1. IBM 4758 PCI Cryptographic Coprocessor Support

The IBM 4758 PCI Cryptographic Coprocessor (4758 Coprocessor) is a programmable cryptographic coprocessor in a Federal Information Processing Standards 4 (FIPS4) tamper responding package. As a programmable device, it can be given multiple "personalities." openCryptoki provides STDLLs that interface to two different firmware loads: deep and shallow.

The deep firmware load provides a complete PKCS #11 implementation running on the card. The STDLL in this case converts the host PKCS #11 operations into command blocks that are passed to the card. In this mode, the full security features of the 4758 Coprocessor are available. The shared object loaded by the API module is `/usr/lib/pkcs11/stdll/PKCS11_4758.so`; however, no applications need to be aware of the name since the configuration process detects and configures the subsystem for the proper tokens. This load is available on the 4758 Coprocessor Web site (http://www.ibm.com/security/cryptocards).

The shallow load does not exploit any of the security features of the card. In this case, the STDLL is more complex, providing for PKCS #11 object and session management and allowing the interoperation of multiple processes or threads as the PKCS #11 specification requires. The module loaded by the API is `/usr/lib/pkcs11/stdll/PKCS11_LW.so`. This firmware is available on a limited basis from IBM, and it essentially turns the 4758 Coprocessor into a cryptographic processor.

7.3.2. IBM Cryptographic Accelerator Model 2058 Support

The ICA device is modeled after the shallow firmware mode of the 4758 Coprocessor. It provides all of the object and session management facilities, and it exposes the PKCS #11 mechanisms to the application. The API loads the module `/usr/lib/pkcs11/stdll/PKCS11_ICA.so` on behalf of the application. One unique feature of this module is that it takes advantage of the transparent load balancing provided by the device driver. The subsystem only does one open on the device, but the driver distributes the work load across as many ICA devices as are in the system. With multiple devices in the system, there is a level of fault tolerance. As long as one deivce in the system is still functional, applications will continue to function but at a degraded level of performance.

8. Applications and openCryptoki

This section describes how to make openCryptoki available to applications and provides an example of how to write such an application.

8.1. Making openCryptoki Available to Applications

Many applications use PKCS #11 tokens. Most of these applications must be configured to load the specific shared object (DLL) for the token. In the case of openCryptoki, only one module (`/usr/lib/pkcs11/PKCS11_API.so`) must be loaded for access to all the tokens currently running in the subsystem. Multiple token types are supported, with each type taking up a slot in the subsystem according to the implementation specifics of the plug-in module.

If devices are added or removed, the PKCS #11 slot where the token resides may change. For this reason, applications should locate the specific token by the token label provided when the token is initialized and not assume

that a specific slot always contains the desired token.

For application-specific configuration information relating to exploitations of PKCS #11, refer to the application's documentation.

8.2. Writing an Application

To develop an application that uses openCryptoki, you must first load the shared object using the dynamic library calls. Then call `C_GetFunctionList`. For example, the following routine loads the shared library and gets the function list for subsequent calls.

```
CK_FUNCTION_LIST  *funcs;
int do_GetFunctionList( void )
{
   CK_RV            rc;
   CK_RV  (*pfoo)();
   void    *d;
   char    *e;
   char    f[]="/usr/lib/pkcs11/PKCS11_API.so";


   printf("do_GetFunctionList...\n");

   d = dlopen(f,RTLD_NOW);
   if ( d == NULL ) {
      return FALSE;
   }

   pfoo = (CK_RV (*)())dlsym(d,"C_GetFunctionList");
   if (pfoo == NULL ) {
      return FALSE;
   }
   rc = pfoo(&funcs);

   if (rc != CKR_OK) {
      show_error("   C_GetFunctionList", rc );
      return FALSE;
   }

   printf("Looks okay...\n");
   return TRUE;

}
```

Once loaded, the application must call the `C_Initialize` function. In the previous example, the function would be invoked with the following lines:

```
CK_C_INITIALIZE_ARGS  cinit_args;
memset(&cinit_args,0x0,sizeof(cinit_args));
funcs->C_Initialize(&cinit_args);
```

Refer to the PKCS #11 specification available from the RSA labs Web site (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/) for more options.

> **Note:** openCryptoki requires that operating system threads be allowed. If other thread routines are passed in, they are ignored. If the `no-os` threads argument is set in the initialize arguments structure, the call to `C_Initialize` will fail.

9. Resources

For additional information about PKCS #11 and openCryptoki, see the following resources:

- Open source location for the openCryptoki project (http://www-124.ibm.com/developerworks/projects/openCryptoki)
- RSA Security Inc. PKCS #11 Specification (http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/)
- IBM Cryptocards (4758) Web site (http://www.ibm.com/security/cryptocards)
- IBM 4758 open source device driver (http://alphaworks.ibm.com)
- openCryptoki mailing list (<`opencryptoki-discussion@www-124.ibm.com`>)

10. Appendix A: Sample Program

The following sample program prints out all of the current tokens and slots in use in the system. If you want to build the sample program, you will also need the Makefile included after the sample.

10.1. Sample Program

```
#if defined(LINUX)
```

```
#include <sys/stat.h>
#endif
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <dlfcn.h>
#include <pkcs11types.h>
#include "slotmgr.h"

#define CFG_SLOT           0x0004
#define CFG_PKCS_INFO      0x0008
#define CFG_TOKEN_INFO     0x0010

CK_RV init(void);
CK_RV cleanup(void);
CK_RV get_slot_list(int, CK_CHAR_PTR);
CK_RV display_slot_info(void);
CK_RV display_token_info(void);

void * dllPtr;
CK_FUNCTION_LIST_PTR  FunctionPtr = NULL;
CK_SLOT_ID_PTR        SlotList = NULL;
CK_ULONG              SlotCount = 0;
Slot_Mgr_Shr_t *      shmp = NULL;
int in_slot;

int
main(int argc, char *argv[]){
   CK_RV rc;                    // Return Code
   CK_FLAGS flags = 0;          // Bit mask for what options were passed in
   CK_CHAR_PTR slot = NULL;     // The PKCS slot number

   /* Load the PKCS11 library */
   init();

   /* Get the slot list and indicate if a slot number was passed in or not */      get_slot_list(fla

   /* Display the current token and slot info */
   display_token_info();
   display_slot_info();

   /* We are done, free the memory we may have allocated. */
   free (slot);
   return rc;
}
```

```
CK_RV
get_slot_list(int cond, CK_CHAR_PTR slot){
   CK_RV                rc;                  // Return Code

   /* Find out how many tokens are present in slots */
   rc = FunctionPtr->C_GetSlotList(TRUE, NULL_PTR, &SlotCount);
   if (rc != CKR_OK) {
      printf("Error getting number of slots: 0x%X\n", rc);
      return rc;
   }

   /* Allocate enough space for the slots information */
   SlotList = (CK_SLOT_ID_PTR) malloc(SlotCount * sizeof(CK_SLOT_ID));

   rc = FunctionPtr->C_GetSlotList(TRUE, SlotList, &SlotCount);
   if (rc != CKR_OK) {
      printf("Error getting slot list: 0x%X\n", rc);
      return rc;
   }

   return CKR_OK;
}

CK_RV
display_slot_info(void){
   CK_RV          rc;        // Return Code
   CK_SLOT_INFO   SlotInfo;  // Structure to hold slot information
   int            lcv;       // Loop control Variable

   for (lcv = 0; lcv < SlotCount; lcv++){
      /* Get the info for the slot we are examining and store in SlotInfo*/
      rc = FunctionPtr->C_GetSlotInfo(SlotList[lcv], &SlotInfo);
      if (rc != CKR_OK) {
         printf("Error getting slot info: 0x%X\n", rc);
         return rc;
      }

      /* Display the slot information */
      printf("Slot #%d Info\n", SlotList[lcv]);
      printf("\tDescription: %.64s\n", SlotInfo.slotDescription);
      printf("\tManufacturer: %.32s\n", SlotInfo.manufacturerID);
      printf("\tFlags: 0x%X\n", SlotInfo.flags);
      printf("\tHardware Version: %d.%d\n",
            SlotInfo.hardwareVersion.major,
```

```
            SlotInfo.hardwareVersion.minor);
      printf("\tFirmware Version: %d.%d\n",
            SlotInfo.firmwareVersion.major,
            SlotInfo.firmwareVersion.minor);
   }
   return CKR_OK;
}

CK_RV
display_token_info(void){
   CK_RV          rc;         // Return Code
   CK_TOKEN_INFO  TokenInfo;  // Variable to hold Token Information
   int            lcv;        // Loop control variable

   for (lcv = 0; lcv < SlotCount; lcv++){
      /* Get the Token info for each slot in the system */
      rc = FunctionPtr->C_GetTokenInfo(SlotList[lcv], &TokenInfo);
      if (rc != CKR_OK) {
         printf("Error getting token info: 0x%X\n", rc);
         return rc;
      }

      /* Display the token information */
      printf("Token #%d Info:\n", SlotList[lcv]);
      printf("\tLabel: %.32s\n", TokenInfo.label);
      printf("\tManufacturer: %.32s\n", TokenInfo.manufacturerID);
      printf("\tModel: %.16s\n", TokenInfo.model);
      printf("\tSerial Number: %.16s\n", TokenInfo.serialNumber);
      printf("\tFlags: 0x%X\n", TokenInfo.flags);
      printf("\tSessions: %d/%d\n", TokenInfo.ulSessionCount,
            TokenInfo.ulMaxSessionCount);
      printf("\tR/W Sessions: %d/%d\n",
            TokenInfo.ulRwSessionCount, TokenInfo.ulMaxRwSessionCount);
      printf("\tPIN Length: %d-%d\n", TokenInfo.ulMinPinLen,
            TokenInfo.ulMaxPinLen);
      printf("\tPublic Memory: 0x%X/0x%X\n",
            TokenInfo.ulFreePublicMemory, TokenInfo.ulTotalPublicMemory);
      printf("\tPrivate Memory: 0x%X/0x%X\n",
            TokenInfo.ulFreePrivateMemory, TokenInfo.ulTotalPrivateMemory);
      printf("\tHardware Version: %d.%d\n", TokenInfo.hardwareVersion.major,
            TokenInfo.hardwareVersion.minor);
      printf("\tFirmware Version: %d.%d\n",
            TokenInfo.firmwareVersion.major,
            TokenInfo.firmwareVersion.minor);
      printf("\tTime: %.16s\n", TokenInfo.utcTime);
```

```
   }
   return CKR_OK;
}

CK_RV
init(void){
   CK_RV rc;             // Return Code
   void (*symPtr)();     // Pointer for the Dll

   /* Open the PKCS11 API shared library, and inform the user if there is an
    * error */
   dllPtr = dlopen("/usr/lib/pkcs11/PKCS11_API.so", RTLD_NOW);
   if (!dllPtr) {
      rc = errno;
      printf("Error loading PKCS#11 library: 0x%X\n", rc);
      fflush(stdout);
      return rc;
   }

   /* Get the list of the PKCS11 functions this token supports */
   symPtr = (void (*)())dlsym(dllPtr, "C_GetFunctionList");
   if (!symPtr) {
      rc = errno;
      printf("Error getting function list: 0x%X\n", rc);
      fflush(stdout);
      return rc;
   }

   symPtr(&FunctionPtr);

   /* If we get here, we know the slot manager is running and we can use PKCS11
    * calls, so we will execute the PKCS11 Initialize command. */
   rc = FunctionPtr->C_Initialize(NULL);
   if (rc != CKR_OK) {
      printf("Error initializing the PKCS11 library: 0x%X\n", rc);
      fflush(stdout);
      cleanup();
   }

   return CKR_OK;
}

CK_RV
cleanup(void){
   CK_RV rc;  // Return Code
```

```
   /* To clean up we will free the slot list we create, call the Finalize
    * routine for PKCS11 and close the dynamically linked library */
   free (SlotList);
   rc = FunctionPtr->C_Finalize(NULL);
   if (dllPtr)
      dlclose(dllPtr);

   exit (rc);
}
```

10.2. Makefile

```
VPATH = ..

INCS = -I../. -I../../../../../include/pkcs11
CFLAGS =   $(OPTLVL)  $(INCS) -DAPI -DDEV -D_THREAD_SAFE -DLINUX -DDEBUG -DSPINXPL


CC = gcc
LD = gcc


LIBS = -ldl -lpthread

OBJS = sample.o

.c.o: ; $(CC) -c $(CFLAGS) -o $@ $<


all:  sample


sample: $(OBJS)
${CC}  ${OBJS} $(LIBS) -o $@

TARGET = sample

build: $(TARGET)

clean:
rm -f *.so *.o $(TARGET)
```