z/OS
Cryptographic Services
Integrated Cryptographic Service Facility

Writing PKCS #11 Applications

SA23-2231-04

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 69.

This edition applies to Version 1 Release 13 of z/OS (5694–A01) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition applies to ICSF FMID HCR7790.

This edition replaces SA23-2231-03.

Contents

Figures

Tables

About this document

This document supports z/OS® (5694-A01). This document describes the support for PKCS #11 provided by the z/OS Integrated Cryptographic Service Facility (ICSF). ICSF is a component of z/OS Cryptographic Services, which includes the following components:

- z/OS Integrated Cryptographic Service Facility (ICSF)
- z/OS Open Cryptographic Services Facility (OCSF)
- z/OS System Secure Socket Level Programming (SSL)
- z/OS Public Key Infrastructure Services (PKI Services)

ICSF is a software element of z/OS that works with the hardware cryptographic feature and the Security Server (RACF®) to provide secure, high-speed cryptographic services. ICSF provides the application programming interfaces by which applications request the cryptographic services.

PKCS #11 is an industry-accepted standard provided by RSA Laboratories of RSA Security Inc. It specifies an application programming interface (API) to devices, referred to as *tokens*, that hold cryptographic information and perform cryptographic functions. PKCS #11 provides an alternative to IBM®'s Common Cryptographic Architecture (CCA).

Who should read this document

This document is primarily intended for application programmers who want to write PKCS #11 applications for z/OS. It also contains information for security administrators, system programmers, and auditors in installations that use PKCS #11 applications.

How this document is organized

- Chapter 1, "Overview of z/OS support for PKCS #11," on page 1 provides an overview of ICSF support for PKCS #11. It discusses tokens, the token data set (TKDS), auditing and tracing PKCS #11 functions, session objects, and tasks that must be performed before using PKCS #11 applications.
- Chapter 2, "The C API," on page 19 discusses the PKCS #11 C API provided by ICSF, highlighting differences between the z/OS implementation and the PKCS #11 standard.
- Chapter 3, "The testpkcs11 program," on page 51 discusses how to build and run the testpkcs11 sample.
- Chapter 4, "ICSF PKCS #11 callable services," on page 55 provides a brief introduction to the PKCS #11 callable services, which are documented in *z/OS Cryptographic Services ICSF Application Programmer's Guide*.

How to use this document

Application programmers should read the entire book.

Security administrators should read the section "Tasks for the security administrator" on page 17 and the information that it references.

System programmers should read the section "Tasks for the system programmer" on page 17 and the information that it references.

Auditors should read the section “Tasks for the auditor” on page 17 and the information that is references.

Where to find more information

Before using this document, application programmers must be familiar with the PKCS #11 specification. The PKCS #11 standard is defined on the RSA Laboratories Web site at http://www.rsa.com/rsalabs/. Application programmers should also be familiar with the ICSF library and C programming.

Security administrators should be familiar with *z/OS Security Server RACF Security Administrator's Guide*.

Auditors should be familiar with *z/OS Security Server RACF Auditor's Guide*.

The callable services for PKCS #11 functions are documented in *z/OS Cryptographic Services ICSF Application Programmer's Guide*.

The format of the token data set is documented in *z/OS Cryptographic Services ICSF System Programmer's Guide*.

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

Use one of the following methods to send us your comments:

1. Send an e-mail to mhvrcfs@us.ibm.com
2. Visit the Contact z/OS Web page at http://www.ibm.com/servers/eserver/zseries/zos/webqs.html
3. Mail the comments to the following address:

   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Mail Station P181
   2455 South Road
   Poughkeepsie, NY 12601-5400
   U.S.A.
4. Fax the comments to us as follows:

   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address
- Your e-mail address
- Your telephone or fax number
- The publication title and order number:

  z/OS Cryptographic Services Writing PKCS #11 Applications
  SA23-2231-04
- The topic and page number related to your comment
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you submit.

If you have a technical problem

Do not use the feedback methods listed above. Instead, do one of the following:

- Contact your IBM service representative
- Call IBM technical support
- Visit the IBM zSeries support Web page at http://www.ibm.com/servers/eserver/support/zseries/.

Chapter 1. Overview of z/OS support for PKCS #11

RSA Laboratories of RSA Security Inc. offers its Public Key Cryptography Standards (PKCS) to developers of computers that use public key and related technology. PKCS #11, also known as Cryptoki, is the cryptographic token interface standard. It specifies an application programming interface (API) to devices, referred to as *tokens*, that hold cryptographic information and perform cryptographic functions. The PKCS #11 API is an industry-accepted standard commonly used by cryptographic applications. ICSF supports PKCS #11, providing an alternative to IBM's Common Cryptographic Architecture (CCA) and broadening the scope of cryptographic applications that can make use of zSeries® cryptography. PKCS #11 applications developed for other platforms can be recompiled and run on z/OS.

The PKCS #11 standard is defined on the RSA Laboratories Web site at http://www.rsa.com/rsalabs/. This document describes how ICSF supports that standard. The support includes the following:

- A token data set (TKDS) that serves as a repository for persistent cryptographic keys and certificates used by PKCS #11 applications.
- Instore memory that serves as a repository for temporary (session-only) cryptographic keys and certificates used by PKCS #11 applications.
- A C application programming interface (API) that supports a subset of the V2.20 level of the PKCS #11 specification
- PKCS #11 specific ICSF callable services. The C API uses these callable services.

Tokens

On most single-user systems, a token is a smart card or other plug-installed cryptographic device, accessed through a card reader or *slot*. The PKCS #11 specification assigns numbers to slots, known as *slot IDs*. An application identifies the token that it wants to access by specifying the appropriate slot ID. On systems that have multiple slots, it is the application's responsibility to determine which slot to access.

z/OS must support multiple users, each potentially needing a unique key store. In this multiuser environment, the system does not give users direct access to the cryptographic cards installed as if they were personal smart cards. Instead, z/OS PKCS #11 tokens are virtual, conceptually similar to RACF (SAF) key rings. An application can have one or more z/OS PKCS #11 tokens, depending on its needs.

Typically, PKCS #11 tokens are created in a factory and initialized either before they are installed or upon their first use. In contrast, z/OS PKCS #11 tokens can be created using system software such as RACF, the gskkyman utility, or by applications using the C API. Each token has a unique token name, or label, that is specified by the end user or application at the time that the token is created.

Rules: A token name must follow these rules:

- Up to 32 characters in length
- Permitted characters are:
  - Alphanumeric
  - National: @ (X'5B'), # (X'7B'), or $ (X'7C')
  - Period: . (X'4B')
- The first character must be alphabetic or national

- Lowercase letters can be used, but are folded to uppercase
- The IBM1047 code page is assumed

In addition to any tokens your installation may create, ICSF will also create a token that will be available to all applications. This "omnipresent" token is created by ICSF in order to enable PKCS #11 services when no other token has been created. This token supports session objects only. Session objects are objects that do not persist beyond the life of a PKCS #11 session. The omnipresent token is always mapped to slot ID #0, and its token label is SYSTOK-SESSION-ONLY.

Tip: To reference the omnipresent token by label, use the constant SESS_ONLY_TOK, which is defined in csnpdefs.h.

Because PKCS #11 tokens are typically physical hardware devices, the PKCS #11 specification provides no mechanism to delete tokens. However, because z/OS PKCS #11 tokens are virtual, z/OS must provide a way to delete them. For information on how to delete tokens using the C API, see "Deleting z/OS PKCS #11 tokens" on page 19.

The token data set (TKDS)

The token data set (TKDS) is a VSAM data set that serves as the repository for persistent cryptographic keys and certificates used by PKCS #11 applications. The system programmer creates the TKDS and updates the ICSF installation options data set to identify the data set name of the TKDS.

As of ICSF FMID HCR7770, a TKDS is no longer required in order to run PKCS #11 applications. If ICSF is started without a TKDS, however, only the omnipresent token will be available.

Rules: The token data set must follow these rules:

- It must be a key-sequenced VSAM data set with spanned variable length records.
- It must be allocated on a permanently resident volume.

Keys in the token data set are not encrypted. Therefore, it is important that the security administrator create a RACF profile to protect the token data set from unauthorized access.

For a sample job to define the TKDS data set, see "Sample job to define the TKDS" on page 3. For the format of the TKDS, see *z/OS Cryptographic Services ICSF System Programmer's Guide*.

To optimize performance, ICSF utilizes in-storage copy of the TKDS.

Options for the TKDS in the ICSF installation options data set

The ICSF installation options data set contains two options related to the token data set:

- TKDSN
- SYSPLEXTKDS

The TKDSN option: The TKDSN option identifies the VSAM data set that contains the token data set. The format of this option is:

```
TKDSN(data_set_name)
```

data_set_name is the name of an existing token data set or an empty VSAM data set to be used as the token data set.

The SYSPLEXTKDS option: The SYSPLEXTKDS option specifies whether the token data set should have sysplex-wide data consistency. The SYSPLEXTKDS option is in effect only if the TKDSN option has also been specified. The format of this option is:

```
SYSPLEXTKDS(YES|NO,FAIL(YES|NO))
```

The SYSPLEXTKDS option can be YES or NO. The default value is NO.

- If `SYSPLEXTKDS(NO,FAIL(`*`fail-option`*`))` is specified, no XCF signalling is performed when an update to a TKDS record occurs.
- If `SYSPLEXTKDS(YES,FAIL(`*`fail-option`*`))` is specified, the system is notified of updates made to the TKDS by other members of the sysplex who have also specified `SYSPLEXTKDS(YES,FAIL(`*`fail-option`*`))`.

The value of FAIL can be YES or NO.

- If `FAIL(YES)` is specified, ICSF initialization terminates abnormally if there is a failure creating the TKDS latch set.
- If `FAIL(NO)` is specified, ICSF initialization processing continues even if the request to create a TKDS latch set fails with an environment failure. The system is not notified of updates to the TKDS by other members of the ICSF sysplex group.

The default is `SYSPLEXTKDS(NO,FAIL(NO))`.

Sample job to define the TKDS

A sample job illustrating the definition of the TKDS data set is shipped in SYS1.SAMPLIB, member CSFTKDS.

```
//CSFTKDS   JOB <JOB CARD PARAMETERS>
///********************************************************************
//*  Licensed Materials - Property of IBM                             *
//*  5694-A01                                                         *
//*  (C) Copyright IBM Corp. 2007                                     *
//*                                                                   *
//* THIS JCL DEFINES A VSAM TKDS TO USE FOR PKCS #11 TOKENS           *
//*    AND OBJECTS                                                    *
//*                                                                   *
//*  CAUTION: This is neither a JCL procedure nor a complete JOB.     *
//*  Before using this JOB step, you will have to make the following  *
//*  modifications:                                                   *
//*                                                                   *
//*  1) Add the job parameters to meet your system requirements.      *
//*  2) Be sure to change CSF to the appropriate HLQ if you choose    *
//*     not to use the default.                                       *
//*  3) The TKDS needs to be on a permanently resident volume.        *
//*                                                                   *
//*********************************************************************
//DEFINE  EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT DD  SYSOUT=*
//SYSIN DD *
        DEFINE CLUSTER(NAME(CSF.CSFTKDS)       -
                  VOLUMES(XXXXXXX)      -
                  RECORDS(100 50)      -
                  RECORDSIZE(2200 32756)   -
                  KEYS(72 0)      -
                  FREESPACE(0 0)      -
                  SPANNED               -
                  SHAREOPTIONS(2 3) )      -
               DATA (NAME(CSF.CSFTKDS.DATA)  -
                    BUFFERSPACE(100000)  -
                    ERASE         -
                    WRITECHECK)        -
               INDEX (NAME(CSF.CSFTKDS.INDEX))
/*
```

Figure 1. Sample job to define the TKDS

Controlling access to tokens

The PKCS #11 standard was designed for systems that grant access to token information based on a PIN. The standard defines two types of users, the standard user (*User*) and the security officer (*SO*), each having its own personal identification number (PIN). The SO can initialize a token (zero the contents) and set the User's PIN. The SO can also access the public objects on the token, but not the private ones. The User has access to the private objects on a token and has the power to change his or her own PIN. The User cannot reinitialize a token. The PIN that a user enters determines which role that user takes. A user can fill both roles by having knowledge of both PINs.

z/OS does not use PINs. Instead, profiles in the SAF CRYPTOZ class control access to tokens. For each token, there are two resources in the CRYPTOZ class for controlling access to tokens:

- The resource `USER.`*`token-name`* controls the access of the User role to the token.
- The resource `SO.`*`token-name`* controls the access of the SO role to the token.

In addition to these two resources for controlling access to tokens, each token also has a FIPSEXEMPT.*token-name* resource for identifying applications that are subject to FIPS 140 restrictions when ICSF is running in FIPS compatibility mode. Refer to “Operating in compliance with FIPS 140-2” on page 11 for more information.

A user's access level to each of these resources (read, update, or control) determines the user's access level to the token.

There are six possible token access levels. Three are defined by the PKCS #11 standard, and three are unique to z/OS. The PKCS #11 token access levels are:
- User R/O: Allows the user to read the token including its private objects, but the user cannot create new token or session objects or alter existing ones.
- User R/W: Allows the user read/write access to the token object including its private objects.
- SO R/W: Allows the user to act as the security officer for the token and to read, create, and alter public objects on the token.

The token access levels unique to z/OS are:
- Weak SO: A security officer that can modify the CA certificates contained in a token but not initialize the token. (For example, a system administrator who determines the trust policy for all applications on the system.)
- Strong SO: A security officer that can add, generate or remove private objects in a token. (For example, a server administrator.)
- Weak User: A User that cannot change the trusted CAs contained in a token. (For example, to prevent an end-user from changing the trust policy of his or her token.)

Table 1 shows how a user's access level to a token is derived from the user's access level to a resource in the SAF CRYPTOZ class.

Table 1. Token access levels

	SAF access level		
CRYPTOZ resource	**READ**	**UPDATE**	**CONTROL**
SO.*token-label*	Weak SO Can read, create, delete, modify, and use public objects	SO R/W Same ability as Weak SO plus can create and delete tokens	Strong SO Same ability as SO R/W plus can read but not use (see Note2 on page 6) private objects; create, delete, and modify private objects
USER.*token-label*	User R/O Can read and use (see Note 2 on page 6) public and private objects	Weak User Same ability as User R/O plus can create, delete, and modify private and public objects. Cannot add, delete, or modify certificate authority objects	User R/W Same ability as Weak User plus can add, delete, and modify certificate authority objects

Notes:

1. The USER.*token-name* and SO.*token-name* profiles will **not** be checked to determine access to the omnipresent token SYSTOK-SESSION-ONLY. ICSF

creates this token to provide PKCS #11 support even if no other token is available to an application. All users will always by considered to have R/W access to this token.

2. “Use” is defined as any of the following:
   - Performing any cryptographic operation involving the key object; for example C_Encrypt
   - Searching for key objects using sensitive search attributes
   - Retrieving sensitive key object attributes.

   The sensitive attribute for a secret key is CKA_VALUE. The sensitive attribute for Diffie Hellman, DSA, and Elliptic Curve private key objects is CKA_VALUE. The sensitive attributes for RSA private key objects are CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2, CKA_EXPONENT_1, CKA_EXPONENT_2, and CKA_COEFFICIENT.
3. The CRYPTOZ resources can be defined as “RACF-DELEGATED” if required. For information about delegated resources, see the section on delegated resources in *z/OS Security Server RACF Security Administrator's Guide*.
4. Although the use of generic profiles in the CRYPTOZ class is permitted, we recommend that you do not use a single generic profile to cover both the SO.*token-label* and USER.*token-label* resources. You should not do this, because another resource (FIPSEXEMPT.*token-label*, which is described in more detail in “Operating in compliance with FIPS 140-2” on page 11) can be used to specify compliance with the FIPS 140-2 standard. Creating a generic profile that uses generic characters to match both the SO and USER portion of the resource name (for example *.*token-label*) will also inadvertently match the FIPSEXEMPT.*token-label* resource and can have unintended consequences.
5. If the CSFSERV class is active, ICSF performs access control checks on the underlying callable services. The user must have READ access to the appropriate CSFSERV class resource. Table 2 lists the resources in the CSFSERV class for token services.

Table 2. Resources in the CSFSERV class for token services

Name of resource	Service	Called by
CSF1TRC	Token or object creation	C_InitToken, C_CreateObject, C_CopyObject
CSF1TRD	Token or object deletion	C_InitToken, C_DestroyObject
CSF1TRL	Token or object find	C_Initialize, C_FindObjects, CSN_FindALLObjects
CSF1SAV	Set object attributes	C_SetAttributeValue
CSF1GAV	Get object attributes	C_GetAttributeValue
CSF1GSK	Generate secret key	C_GenerateKey
CSF1GKP	Generate key pair	C_GenerateKeyPair
CSF1PKS	Private key sign	C_Decrypt, C_DecryptUpdate, C_DecryptFinal, C_Sign, C_SignFinal
CSF1PKV	Public key verify	C_Encrypt, C_EncryptUpdate, C_EncryptFinal, C_Verify, C_VerifyFinal

Table 2. Resources in the CSFSERV class for token services (continued)

Name of resource	Service	Called by
CSF1SKD	Secret key decrypt	C_Decrypt, C_DecryptUpdate, C_DecryptFinal
CSF1SKE	Secret key encrypt	C_Encrypt, C_EncryptUpdate, C_EncryptFinal
CSFOWH	One-way hash	C_Digest, C_DigestUpdate, C_DigestFinal, C_Sign, C_SignUpDate, C_SignFinal, C_Verify, C_VerifyUpdate, C_VerifyFinal
CSF1WPK	Wrap key	C_Wrapkey
CSF1UWK	Unwrap key	C_Unwrapkey
CSF1HMG	Generate HMAC	C_Sign
CSF1HMV	Verify HMAC	C_Verify
CSF1DVK	Derive key	C_DeriveKey
CSF1DMK	Derive multiple keys	C_DeriveKey
CSFIQA	PKCS #11 initialization	C_Initialize
CSFRNG	Random number generate	C_GenerateRandom

Guidelines:

1. If your organization controls access to ICSF callable services using the CSFSERV class, define the resources listed in Table 2 on page 6 and grant access accordingly.

   Tip: Define generic profiles. For example, a profile named CSF* covers all the ICSF services. A profile named CSF1* covers the PKCS #11 subset of the ICSF services, with the exception of those covered by the CSFOWH, CSFIQF, and CSFRNG resources.
2. The CRYPTOZ class supports generic profiles. Take advantage of this by creating a token naming convention for your organization and enforce it with generic profiles. For example, require users and applications to prefix their token names with their user IDs, as with data set names. (See “Sample scenario for setting up z/OS PKCS #11 tokens” on page 8.)
3. For server applications, grant security officers (server administrators) Strong SO access and their end-users (server daemon user IDs) Weak User or User R/W access.
4. For applications for which you do not wish to separate the security officer and end-user roles, grant the appropriate user IDs access to both the SO and USER profiles.

Managing tokens

z/OS provides several facilities to manage tokens:

- A C language application programming interface (API) that implements a subset of the PKCS #11 specification. For a description of this API, see Chapter 2, “The C API,” on page 19.
- PKCS #11 specific ICSF callable services. The C API uses these callable services. For information about these callable services, see Chapter 4, “ICSF PKCS #11 callable services,” on page 55.

- ISPF panels. The ICSF ISPF panels provide the capability to see a formatted view of TKDS objects, and make limited updates to them.
- The RACF RACDCERT command supports the certificate, public key, and private key objects, and provides the following subfunctions to manage these objects:
  - ADDTOKEN - creates a new empty token
  - DELTOKEN - deletes an existing token and everything in it
  - LISTTOKEN - displays information on the certificate objects in a token and whether associated public and private key objects exist
  - BIND - connects a RACF certificate, its public key, and potentially its private key to an existing token
  - UNBIND - removes a certificate and its keys from a token
  - IMPORT - defines a token certificate to RACF

  For information about the RACDCERT command, see *z/OS Security Server RACF Command Language Reference* and *z/OS Security Server RACF Security Administrator's Guide*.
- The SAF CRYPTOZ class controls access to tokens. For information about this class, see “Controlling access to tokens” on page 4.
- The RACF R_Datalib callable service (IRRSDL00) allows applications to read tokens by providing a user ID of `*TOKEN*` to indicate that the key ring name is really a token name. For information about R_Datalib, see *z/OS Security Server RACF Callable Services*.

  Note: IRRSDL00 was originally created to allow applications to read RACF (SAF) key rings, but has been enhanced to read PKCS #11 tokens as well. Thus applications written to read key rings can also read tokens without being modified.

Sample scenario for setting up z/OS PKCS #11 tokens

The following examples show how to control access to z/OS PKCS #11 tokens. In this scenario, a company wants to use z/OS PKCS #11 tokens as the key stores for its FTP and Web servers. The company has established a naming convention for their tokens requiring that all tokens have the owning user ID as the high-level qualifier. The owning user IDs for the FTP and Web server tokens are the daemons FTPSRV and WEBSRV, respectively. User ABIGAIL is the administrator for the servers.

The security administrator, who has the RACF SPECIAL attribute, creates the protection profiles for the tokens. The security administrator's goal is to give user ABIGAIL the Security Officer role for these profiles, and to give the daemon user IDs the User role. To do this, the security administrator issues RACF TSO commands. First, the security administrator activates the CRYPTOZ class with generics and RACLISTs it:

```
SETROPTS CLASSACT(CRYPTOZ) GENERIC(CRYPTOZ) RACLIST(CRYPTOZ)
```

Next, the security administrator creates profiles for the security officer's access to the FTP and Web Server tokens:

```
RDEFINE CRYPTOZ SO.FTPSRV.* UACC(NONE)
RDEFINE CRYPTOZ SO.WEBSRV.* UACC(NONE)
```

Then, the security administrator creates profiles for the standard user's access to the FTP and Web Server tokens:

```
RDEFINE CRYPTOZ USER.FTPSRV.* UACC(NONE)
RDEFINE CRYPTOZ USER.WEBSRV.* UACC(NONE)
```

The security administrator now gives user ABIGAIL Strong SO power for the tokens by giving her CONTROL access to the profiles that protect the tokens. The Strong SO power does not allow ABIGAIL to use the private objects in the tokens:

```
PERMIT SO.FTPSRV.* CLASS(CRYPTOZ) ID(ABIGAIL) ACC(CONTROL)
PERMIT SO.WEBSRV.* CLASS(CRYPTOZ) ID(ABIGAIL) ACC(CONTROL)
```

Next, the security administrator gives the users FTPSRV and WEBSRV Weak User power for their respective tokens. This power allows them to use the private objects within the tokens, but not change the set of trusted CA certificates.

```
PERMIT USER.FTPSRV.* CLASS(CRYPTOZ) ID(FTPSRV) ACC(UPDATE)
PERMIT USER.WEBSRV.* CLASS(CRYPTOZ) ID(WEBSRV) ACC(UPDATE)
```

Finally, the security administrator refreshes the in-storage profiles for the CRYPTOZ class, so that the changes he just made take effect:

```
SETROPTS RACLIST(CRYPTOZ) REFRESH
```

Now the set up is complete: ABIGAIL has Strong SO power over the tokens for the FTP server and the Web server, and can create the required tokens. FTPSRV and WEBSRV have User power over their respective tokens, and can use them as key stores after ABIGAIL has created them.

The task now is to create and populate the tokens for the servers with RACF certificates. The following certificates exist:

1. A root CA certificate installed under CERTAUTH with label 'Local Root CA for Servers'
2. An end-entity certificate and private key installed under user FTPSRV with label 'FTP Key'. This certificate was signed by the first certificate.
3. An end-entity certificate and private key installed under user WEBSRV with label 'Web Key'. This certificate was also signed by the first certificate.

User ABIGAIL issues the following TSO commands to create the tokens, using the company's naming conventions:

```
RACDCERT ADDTOKEN(ftpsrv.ftp.server.pkcs11.token)
RACDCERT ADDTOKEN(websrv.web.server.pkcs11.token)
```

Next, issue the commands that bind the root CA certificate to the two tokens:

```
RACDCERT BIND(CERTAUTH LABEL('Local Root CA for Servers')
TOKEN(ftpsrv.ftp.server.pkcs11.token)
RACDCERT BIND(CERTAUTH LABEL('Local Root CA for Servers')
TOKEN(websrv.web.server.pkcs11.token)
```

Now, bind the end-entity certificates to their respective tokens. Each should be the default in the token.

```
RACDCERT BIND(ID(FTPSRV) LABEL("FTP key")
TOKEN(ftpsrv.ftp.server.pkcs11.token) DEFAULT)
RACDCERT BIND(ID(WEBSRV) LABEL("Web key")
TOKEN(websrv.web.server.pkcs11.token) DEFAULT)
```

The final step is for the user (ABIGAIL) to configure both servers to use their respective tokens: add directives to the servers' configuration files.

For the web server (IBM HTTP Server), the keyfile directive in the httpd.conf file is set as follows:

```
keyfile *TOKEN*/WEBSRV.WEB.SERVER.PKCS11.TOKEN SAF
```

The SAF keyword indicates to SSL that this is a key ring and is controlled by SAF; it is not a KDB file. The TOKEN keyword indicates that the key ring is a token. The FTP server configuration file also requires a token-qualified key ring name:

```
keyfile *TOKEN*/FTPSRV.FTP.SERVER.PKCS11.TOKEN
```

Auditing PKCS #11 functions

PKCS #11 functions are audited in the following ways:

- The SMF type 82 subtype 1 record that is written during ICSF initialization contains the data set name of the token data set (TKDS).
- The SMF type 82 subtype 21 record that is written when a member joins or leaves a sysplex group contains the cryptographic keys data set (CKDS) data set name if the member joined or left the ICSF CKDS sysplex group, or the TKDS data set name if the member joined or left the ICSF TKDS sysplex group.
- ICSF writes SMF type 82 subtype 23 records whenever a TKDS record for a token or token object is created, modified, or deleted. ICSF does not write SMF records for changes to session objects.

For descriptions of the SMF records that ICSF writes, see *z/OS MVS System Management Facilities (SMF)*.

Component trace for PKCS #11 functions

The following ICSF component trace entries trace events related to the token data set (TKDS):

- Type 16 (XCFTMSGS) traces the broadcast of an XCF message related to TKDS I/O.
- Type 17 (XCFTMSGR) traces the receipt of an XCF message related to TKDS I/O.
- Type 18 (XCFTENQ) traces the return of control to the TKDS I/O subtask following the request for an exclusive ENQ on the SYSZTKDS.TKDS*dsn* resource.

These trace entry types are always traced.

When viewed via IPCS, these entries show the ASCB address, the TCB address, the ASID, the general purpose registers, the GPR length, and the CSS address. For more information about IPCS, see *z/OS MVS IPCS User's Guide*.

Object types

ICSF supports PKCS #11 session objects and token objects. The following classes of objects can be associated with these object types:

- Certificate
- Public key
- Private key
- Secret key
- Data objects
- Domain parameters

Session objects

A session object exists for the life of a PKCS #11 session. ICSF allocates session object memory areas to hold session objects; they are not maintained on DASD. ICSF associates a session object memory area with the application that requested the creation of a session object. There is only one session object memory area for an application, even if the application spawns multiple PKCS #11 sessions. The same session objects are available to all PKCS #11 sessions within an application.

ICSF creates a session object memory area the first time a session object is created, if there is currently no session object memory area associated with the application. The session object memory area exists as long as the PKCS #11 application's address space and job step TCB exist. ICSF deletes the memory area if either the address space or job step TCB terminates. If ICSF terminates, all session object memory areas are destroyed.

ICSF creates one session-object token, the omnipresent token, to provide PKCS #11 support even if no other token is available to an application. For example, no other token is available when a TKDS is not identified using the TKDSN option in the ICSF installation options data set, or when the SAF CRYPTOZ class has not been activated. This session object token (labeled SYSTOK-SESSION-ONLY) is write protected, cannot be used to store persistent attributes, and cannot be deleted.

On z/OS, an application can be running in either single address space mode, or in cross memory mode. The PKCS #11 standard has no concept of cross memory mode, so there is no predefined expected behavior for a PKCS #11 application running in cross memory mode. If running in cross memory mode, you should be aware of the guidelines pertaining to session objects described in “Cross memory considerations” on page 20.

Token objects

Token objects are stored in the token data set, with one record per object. They are visible to all applications that have sufficient permission to the token. They are persistent: they remain associated with the token even after a session is closed.

Operating in compliance with FIPS 140-2

The National Institute of Standards and Technology (NIST) is the US federal technology agency that works with industry to develop and apply technology, measurements, and standards. One of the standards published by NIST is the Federal Information Processing Standard Security Requirements for Cryptographic Modules, referred to as FIPS 140-2. FIPS 140-2 provides a standard that can be required by organizations who specify that cryptographic-based security systems are to be used to provide protection for sensitive or valuable data.

z/OS PKCS #11 cryptography is designed to meet FIPS 140-2 Level 1 criteria, and can be configured to operate in compliance with FIPS 140-2 specifications. Applications that need to comply with the FIPS 140-2 standard can therefore use the z/OS PKCS #11 services in a way that allows only the cryptographic algorithms (including key sizes) approved by the standard and restricts access to the algorithms that are not approved. There are two modes of FIPS operation:

- The services can be configured so that all z/OS PKCS #11 applications are forced to comply with the FIPS 140-2 standard. This is called *FIPS standard mode*.
- For installations where only certain z/OS PKCS #11 applications need to comply with the FIPS 140-2 standard, the services can be configured so that only the necessary applications are restricted from using the non-approved algorithms and key sizes, while other applications are not. This is called *FIPS compatibility mode*.

You can also use FIPS compatibility mode to test individual applications to ensure FIPS compliance before switching to FIPS standard mode.

ICSF installation options are described in the *z/OS Cryptographic Services ICSF System Programmer's Guide*. The installation option FIPSMODE indicates one of the following:

- the z/OS PKCS #11 services will operate in FIPS standard mode. The installation option to specify this is FIPSMODE(YES, FAIL(*fail-option*)) and is described in more detail in "Requiring FIPS 140-2 compliance from all z/OS PKCS #11 applications" on page 14.
- the z/OS PKCS #11 services will operate in FIPS compatibility mode. The installation option to specify this is FIPSMODE(COMPAT, FAIL(*fail-option*)). When operating in FIPS compatibility mode, it is expected that further specifications will be made to identify which applications must comply with the FIPS 140-2 standard, and which applications do not need to comply. These further specifications can be made:
  - at the PKCS #11 token and application level, using FIPSEXEMPT.*token-name* resource profiles in the CRYPTOZ class.
  - within applications themselves for individual keys. When an application creates a key, the application can specify that the key must be used in a FIPS 140-2 compliant fashion. The application can specify this by setting the Boolean key attribute CKA_IBM_FIPS140 to TRUE.

  The FIPSMODE(COMPAT, FAIL(*fail-option*)) installation option, FIPSEXEMPT.*token-name* resource profiles, and the CKA_IBM_FIPS140 key attribute, are described in more detail in "Requiring FIPS 140-2 compliance from select z/OS PKCS #11 applications" on page 14.
- no FIPS 140-2 compliance is required by any application. This is the default behavior if the FIPSMODE installation option is not used, but can be set explicitly using the FIPSMODE(NO, FAIL(fail-option)) installation option.

If any z/OS PKCS #11 application intends to use the services in compliance with the FIPS 140-2 standard, then, in accordance with that standard, the integrity of the load module containing the z/OS PKCS #11 services must be checked when ICSF is started. This load module is digitally signed, and, in order for applications using its services to be FIPS 140-2 compliant, the signature must be verified when ICSF is started. For more information, refer to "Requiring signature verification for ICSF module CSFINPV2."

If any application will use PKCS #11 objects for AES Galois/Counter Mode (GCM) encryption or GMAC generation, and will have ICSF generate the initialization vectors, then you need to set ECVTSPLX or CVTSNAME to a unique value. Refer to *z/OS Cryptographic Services ICSF System Programmer's Guide* for more information.

Requiring signature verification for ICSF module CSFINPV2

If your installation needs to operate z/OS PKCS #11 in compliance with the FIPS 140-2 standard, then the integrity of the cryptographic functions shipped by IBM must be verified at your installation during ICSF startup. The load module that contains the software cryptographic functions is SYS1.SIEALNKE(CSFINPV2), and this load module is digitally signed when it is shipped from IBM. Using RACF, you can verify that the module has remained unchanged from the time it was built and installed on your system. To do this, you create a profile in the PROGRAM class for the CSFINPV2 module, and use this profile to indicate that signature verification is required before the module can be loaded.

To require signature verification for ICSF module CSFINPV2:

1. Make sure that RACF has been prepared to verify signed programs. As described in *z/OS Security Server RACF Security Administrator's Guide*, a security administrator prepares RACF to verify signed programs by creating a key ring for signature verification, and adding the code-signing CA certificate that is supplied with RACF to the key ring. If RACF has been prepared to verify signed programs, there will be a key ring dedicated to signature verification, the code-signing CA certificate will be attached to the key ring, and the PROGRAM class will be active.

   a. If RACF has been prepared to verify signed programs, the discrete profile IRR.PROGRAM.SIGNATURE.VERIFICATION in the FACILITY class will specify the name of the signature-verification key ring. To determine if a signature key ring is already active, enter the command:

   ```
   RLIST FACILITY IRR.PROGRAM.SIGNATURE.VERIFICATION
   ```

   If there is no discrete profile with this name, have your security administrator prepare RACF to verify signed programs using the information in *z/OS Security Server RACF Security Administrator's Guide*.

   b. If the signature verification key ring exists, the RLIST command will display information for the discrete profile IRR.PROGRAM.SIGNATURE.VERIFICATION in the FACILITY class. The name of the signature verification key ring and the name of the key ring owner will be included in the APPLICATION DATA field of the RLIST command output. Using this information, enter the RACDCERT LISTRING command to make sure the code-signing CA certificate is attached to the key ring:

   ```
   RACDCERT ID(key-ring-owner) LISTRING(key-ring-name)
   ```

   The label of the code-signing CA certificate is 'STG Code Signing CA'. If this label is not shown in the RACDCERT LISTRING command output, have your security administrator prepare RACF to verify signed programs using the information in *z/OS Security Server RACF Security Administrator's Guide*.

   c. Program control must be active in order for RACF to perform signature verification processing. To make sure the PROGRAM class is active, enter the SETROPTS LIST command.

   ```
   SETROPTS LIST
   ```

   The ACTIVE CLASSES field of the command output should include the PROGRAM class. If it does not, have your security administrator prepare RACF to verify signed programs using the information in *z/OS Security Server RACF Security Administrator's Guide*.

2. Create a profile for the CSFINPV2 program module in the PROGRAM class, indicating that the program must be signed. The following command specifies that the program should fail to load if the signature cannot be verified for any reason. This command also specifies that all signature verification failures should be logged.

   Note: Due to space constraints, this command example appears on two lines. However, the RDEFINE command should be entered completely on one line.

   ```
   RDEFINE PROGRAM CSFINPV2 ADDMEM('SYS1.SIEALNKE'//NOPADCHK) UACC(READ)
    SIGVER(SIGREQUIRED(YES) FAILLOAD(ANYBAD) SIGAUDIT(ANYBAD))
   ```

You will need to activate your profile changes in the PROGRAM class.

```
SETROPTS WHEN(PROGRAM) REFRESH
```

Requiring FIPS 140-2 compliance from all z/OS PKCS #11 applications

If all z/OS PKCS #11 applications running on your system must comply with the FIPS 140-2 standard, your installation's system programmer should configure ICSF so that z/OS PKCS #11 operates in FIPS standard mode. To do this:

1. Make sure the integrity of the cryptographic functions shipped by IBM in the ICSF module CSFINPV2 will be verified by RACF before the module is loaded. This is done by following the instructions in “Requiring signature verification for ICSF module CSFINPV2” on page 12. If the these steps are not followed to verify the digital signature of the module, no application calling the z/OS PKCS #11 services can be considered FIPS 140-2 compliant.
2. To specify FIPS standard mode, have you installation's system programmer include the installation option FIPSMODE(YES, FAIL(*fail-option*)) in the ICSF installation options data set.

   When this option is used, ICSF will operate in FIPS standard mode. In this mode, ICSF initialization will test that it is running on an IBM System z model type, and a version and release of z/OS, that supports FIPS. If so, then ICSF will perform a series of cryptographic known answer tests as required by the FIPS 140-2 standard. If the tests succeed, then all applications calling z/OS PKCS services will be restricted from using the PKCS #11 algorithms and key sizes that are prohibited by the FIPS 140-2 standard (as outlined in Table 5 on page 26).

   If any of the installation tests should fail, the action ICSF initialization takes depends on the *fail-option* specified. The *fail-option* within the FIPSMODE(YES, FAIL(*fail-option*)) installation option can be either:

   - YES (which indicates that ICSF should terminate abnormally if there is a failure in any of the tests that are performed).
   - NO (which indicates that ICSF initialization processing should continue even if there is a failure in one or more of the tests that are performed). If an initialization test does fail, however, PKCS #11 support will be limited or nonexistent depending on the test that failed.
     - If ICSF is running on an IBM system z model type or with a version of z/OS that does not support FIPS, most FIPS processing is bypassed. PKCS #11 callable services will be available, but ICSF will not adhere to FIPS 140 restrictions. Requests to generate or use a key with CKA_IBM_FIPS140=TRUE will result in a failure return code.
     - If a known answer test failed, all ICSF PKCS #11 callable services will be unavailable.

   For more information on this on other ICSF installation options, refer to *z/OS Cryptographic Services ICSF System Programmer's Guide*.

Requiring FIPS 140-2 compliance from select z/OS PKCS #11 applications

If only certain z/OS PKCS #11 applications running on your system must comply with the FIPS 140-2 standard, while other z/OS PKCS #11 applications do not, your system programmer should configure ICSF so that z/OS PKCS #11 operates in FIPS compatibility mode. In FIPS compatibility mode, you can use resource profiles in the CRYPTOZ class to specify, at a token level, the applications that are exempt from FIPS 140-2 compliance and, for that reason, should not be subject to FIPS restrictions. To configure the z/OS PKCS #11 services to operate in FIPS compatibility mode:

1. Make sure the integrity of the cryptographic functions shipped by IBM in the module ICSF module CSFINPV2 will be verified by RACF before the module is loaded. This is done by following the instructions in “Requiring signature verification for ICSF module CSFINPV2” on page 12. If the these steps are not followed to verify the digital signature of the module, no application calling the z/OS PKCS #11 services can be considered FIPS 140-2 compliant.
2. To specify FIPS compatibility mode, have you installation's system programmer include the installation option FIPSMODE(COMPAT, FAIL(*fail-option*)) in the ICSF installation options data set.

   When this option is used, ICSF will operate in FIPS compatibility mode. In this mode, ICSF initialization will test that it is running on a IBM System z model type, and a version and release of z/OS, that supports FIPS. If so, then ICSF will perform a series of cryptographic known answer tests as required by the FIPS 140-2 standard. If the tests are successful, then, by default, all applications calling z/OS PKCS services will be restricted from using the PKCS #11 algorithms and key sizes that are prohibited by the FIPS 140-2 standard (as outlined in Table 5 on page 26). Using profiles in the CRYPTOZ class, however, you can identify applications that are exempt from FIPS 140-2 compliance (as described in the next step).

   If any of the installation tests should fail, the action ICSF initialization takes depends on the *fail-option* specified. The *fail-option* within the FIPSMODE(COMPAT, FAIL(*fail-option*)) installation option can be either:

   - YES (which indicates that ICSF should terminate abnormally if there is a failure in any of the tests that are performed).
   - NO (which indicates that ICSF initialization processing should continue even if there is a failure in one or more of the tests that are performed). If an initialization test does fail, however, PKCS #11 support will be limited or nonexistent depending on the test that failed.
     - If ICSF is running on an IBM system z model type or with a version of z/OS that does not support FIPS, most FIPS processing is bypassed. PKCS #11 callable services will be available, but ICSF will not adhere to FIPS 140 restrictions. Requests to generate or use a key with CKA_IBM_FIPS140=TRUE will result in a failure return code.
     - If a known answer test failed, all ICSF PKCS #11 callable services will be unavailable.

   For more information on this on other ICSF installation options, refer to *z/OS Cryptographic Services ICSF System Programmer's Guide*.
3. To specify which applications must comply with FIPS 140-2 restrictions and which applications do not need to comply, create FIPSEXEMPT.*token-label* resource profiles in the CRYPTOZ class. If no FIPSEXEMPT.*token-label* resource profiles are created, then all z/OS PKCS #11 applications will be subject to FIPS restrictions. By creating a FIPSEXEMPT.*token-label* resource profile for a particular token, however, you can specify whether or not a particular user ID should be considered exempt from FIPS restrictions when using that token.
   - If a user ID has access authority NONE to the FIPSEXEMPT.*token-label* resource, ICSF will enforce FIPS 140-2 compliance for that user ID.
   - If a user ID has access authority READ to the FIPSEXEMPT.*token-label* resource, that user ID is exempt from FIPS 140-2 restrictions.

To specify which applications must comply with the FIPS 140-2 restrictions, and which do not, the security administrator must:

a. If it is not already activated, activate the CRYPTOZ class with generics and RACLIST it:

   ```
   SETROPTS CLASSACT(CRYPTOZ) GENERIC(CRYPTOZ) RACLIST(CRYPTOZ)
   ```

b. Create the FIPSEXEMPT.*token-label* resource profile for each z/OS PKCS #11 token. The following command creates the profile for the omnipresent session-object token SYSTOK-SESSION-ONLY.

   ```
   RDEF CRYPTOZ FIPSEXEMPT.SYSTOK-SESSION-ONLY UACC(NONE)
   ```

   Although the use of generic profiles in the CRYPTOZ class is permitted, we recommend you begin the profile name with “FIPSEXEMPT”. Failure to do this could result in generic characters unintentionally matching the SO.*token-label* or USER.*token-label* resources for token access, and so could have unintended consequences.

c. Using the PERMIT command, specify READ access authority for user IDs that are exempt from FIPS 140-2 restrictions, and NONE access authority for user IDs that must comply with FIPS 140-2. The following command indicates that all user IDs are exempt, except for the daemon user ID BOGD.

   ```
   PERMIT FIPSEXEMPT.SYSTOK-SESSION-ONLY CLASS(CRYPTOZ) ID(*) ACC(READ)
   PERMIT FIPSEXEMPT.SYSTOK-SESSION-ONLY CLASS(CRYPTOZ) ID(BOGD) ACC(NONE)
   ```

d. Refresh the CRYPTOZ class in common storage:

   ```
   SETROPTS RACLIST(CRYPTOZ) REFRESH
   ```

Specifying FIPS 140-2 compliance from within a z/OS PKCS #11 application

When running in FIPS compatability mode, a PKCS #11 application can, when creating a key, specify that generation and subsequent use of the key must adhere to FIPS 140-2 restrictions. An application specifies this by setting the Boolean attribute CKA_IBM_FIPS140 to TRUE when creating the key. If an application does this, the FIPS 140-2 restrictions (as outlined in Table 5 on page 26) will be enforced for the key regardless of any specifications made at the token level using FIPSEXEMPT.*token-label* resource profiles.

An application controls FIPS 140-2 compliance for a key when in FIPS compatibility mode as specified by the FIPSMODE(COMPAT, FAIL(*fail-option*)) installation option. If the installation option FIPSMODE(NO, FAIL(*fail-option*)), which indicates no FIPS 140-2 compliance for any application, is specified (or defaulted to), an application that sets the Boolean attribute CKA_IBM_FIPS140 to TRUE will fail with return/reason code 8/3069. If the FIPSMODE(YES, FAIL(*fail-option*)) installation option is specified, indicating FIPS 140-2 compliance is required by all applications, setting the Boolean attribute CKA_IBM_FIPS140 to TRUE is merely redundant and does not result in an error.

Preparing to use PKCS #11 applications

Before an installation can use PKCS #11 applications, some preparation is required on the part of the system programmer, security administrator, auditor, and application programmers. This section describes the preparation required.

Tasks for the system programmer

If persistent PKCS #11 tokens and objects are needed, the system programmer allocates a token data set (TKDS) for use by PKCS #11 functions, and specifies the data set name of the TKDS in the TKDSN option of the ICSF installation options data set.

The system programmer must decide whether or not sysplex-wide consistency of the TKDS is required, and must specify the SYSPLEXTKDS option in the ICSF installation options data set to define the processing of TKDS updates in a sysplex environment.

For information on the TKDSN and SYSPLEXTKDS options of the ICSF installation options data set, see “Options for the TKDS in the ICSF installation options data set” on page 2. For a sample job to create the token data set, see “Sample job to define the TKDS” on page 3.

If any application must comply with the FIPS 140-2 standard, the system programmer must configure ICSF to run PKCS #11 services in compliance with FIPS 140-2. To do this, the system administrator uses the FIPSMODE option to specify either FIPS standard mode or FIPS compatibility mode as required. For more information on the FIPSMODE option, refer to “Operating in compliance with FIPS 140-2” on page 11, and the *z/OS Cryptographic Services ICSF System Programmer's Guide*.

The system programmer should run the testpkcs11 utility program to test the PKCS #11 configuration. For information about running the testpkcs11 program, see “Running the pre-compiled version of testpkcs11” on page 51.

Tasks for the security administrator

The security administrator should create a RACF profile to protect the data set that contains the token data set. It is important to protect this data set, because keys in the token data set are not encrypted.

The security administrator needs to grant the appropriate access authority to users for accessing tokens and objects, by defining profiles in the CRYPTOZ class. For more information, see “Controlling access to tokens” on page 4.

The security administrator controls access to the PKCS #11 callable services by defining profiles in the CSFSERV class. For information about defining profiles in the CSFSERV class, see *z/OS Cryptographic Services ICSF Administrator's Guide*. For a list of the resource names for token services, see Table 2 on page 6.

If PKCS #11 services must run in compliance with the FIPS 140-2 standard, then the security administrator must ensure that the digital signature of the load module containing the z/OS PKCS #11 services is verified when ICSF starts. This must be done to satisfy FIPS 140-2 requirements. Refer to “Requiring signature verification for ICSF module CSFINPV2” on page 12, and *z/OS Security Server RACF Security Administrator's Guide* for more information.

Tasks for the auditor

Auditors should become familiar with the data in SMF records that is related to PKCS #11 functions. For more information, see “Auditing PKCS #11 functions” on page 10.

Tasks for application programmers

Application programmers can write applications using the PKCS #11 API provided by ICSF. They should become familiar with the PKCS #11 specification, and with the information in this book. The PKCS #11 standard is defined on the RSA Laboratories Web site at http://www.rsa.com/rsalabs/.

Application programmers can use the sample program, testpkcs11, to learn about building and running PKCS #11 applications, and to troubleshoot problems. For information about the sample program, see Chapter 3, “The testpkcs11 program,” on page 51.

Chapter 2. The C API

ICSF provides a PKCS #11 C language application program interface (API). This chapter highlights the differences between the z/OS API and the PKCS #11 V2.20 specification. To use this API, you must be familiar with both the PKCS #11 specification and the information in this chapter.

All manifest constants specified in this chapter can be found in the csnpdefs.h include file and (with the exceptions noted) in the PKCS #11 specification.

Using the C API

To create or use a z/OS PKCS #11 token, an application needs to do the following:

1. Implicitly or explicitly load the PKCS #11 API DLL (CSNPCAPI for applications running in 31-bit addressing mode not using XPLINK, CSNPCA3X for applications running in 31-bit addressing mode using XPLINK, CSNPCA64 for 64-bit addressing mode).
2. Locate the functions within that DLL, using the C_GetFunctionList function.
3. Call C_Initialize, which enables the application to call other functions in the API.
4. Determine the slots present, using the C_GetSlotList function. This function returns a slot number for each existing token to which the application has access.
5. To use an existing token, the application iterates through the slots using C_GetTokenInfo to find the token wanted.

   To create a new token, the application uses the C_WaitForSlotEvent function to add a new slot containing an uninitialized token. The application then uses the C_InitToken function to initialize the new token and save it in the TKDS.

Deleting z/OS PKCS #11 tokens

Because PKCS #11 tokens are typically physical hardware devices, the PKCS #11 specification provides no mechanism to delete tokens. However, for z/OS PKCS #11 tokens, which are virtual, there must be a capability to delete tokens. An application does this by calling the C_InitToken function with a special label value `$$DELETE-TOKEN$$` (assuming code page IBM1047), left justified and padded on the right to 32 characters.

Tip: Use the constant DEL_TOK defined in csnpdefs.h.

You cannot delete the omnipresent token SYSTOK-SESSION-ONLY (created by ICSF to provide PKCS #11 support even if no other token is available to an application). If an application attempts to delete the omnipresent token, the C_InitToken function will fail with a return value of CKR_TOKEN_WRITE_PROTECTED.

Environment

Restriction: The calling program must be running as a Language-Environment-enabled (LE-enabled) application in TCB mode only. SRB mode is not supported.

Guideline: To use PKCS #11 in SRB mode, you must call the PKCS #11 ICSF callable services directly.

Cross memory considerations

On z/OS, an application can be running in either single address space mode, or in cross memory mode. The PKCS #11 standard has no concept of cross memory mode, so there is no predefined expected behavior for a PKCS #11 application running in cross memory mode.

When running in cross memory mode, the unit of work is running with PRIMARY set to an address space that differs from HOME. This PRIMARY space may be another address space that is logically part of the overall application (for example, if the application was designed to be cross memory aware) or it may be a daemon or subsystem address space dedicated to some system service that the calling application has invoked using a Program Call (PC). Either way, you should be aware of the following z/OS PKCS #11 application behaviors and associated guidelines.

- The C API invokes Language Environment (LE) services that are not supported in cross memory mode.

  Guideline: To use PKCS #11 in cross memory mode, you must call the PKCS #11 ICSF callable services directly.
- Tokens, token objects, and session objects belonging to installation-defined PKCS #11 tokens (but not to the omnipresent token) are protected by RACF access control. Additionally, the ICSF callable services themselves may also be protected by RACF access control. In both cases, the user ID that is used for the access check is always associated with the unit of work. This is either the user ID assigned to the HOME ASCB or the user ID assigned to the Task Control Block (TCB) or System Request Block (SRB).

  Guideline: If the PRIMARY address space function invoked by a PC uses PKCS #11 services, the user ID associated with the caller's unit of work must be appropriately permitted to the CRYPTOZ or CSFSERV resource being checked. If this is a system service, then all such callers must have access.
- FIPS140 compatibility mode behavior is also controlled by resources in the CRYPTOZ Class.

  Guideline: If the system is configured for FIPS140 compatibility mode and the PRIMARY address space function invoked by a PC is expected to adhere to FIPS 140-2 restrictions, the user ID associated with the caller's unit of work should not be permitted to the CRYPTOZ FIPSEXEMPT resource being checked. If this is a system service, then all such callers should not to be given access.
- By definition, session objects (including those belonging to the omnipresent token) are scoped to a single address space. For session objects belonging to installation defined PKCS #11 tokens, the scoping is to the HOME address space at the time of object creation, even if PRIMARY does not equal HOME. These objects are accessible to all units of work belonging to the HOME address space only, even if the PRIMARY address space function invoked by a PC is intended to be the logical owner of the PKCS #11 object.

  In contrast, session objects belonging to the omnipresent token are scoped to the PRIMARY address space at the time of object creation and are addressable by all units of work running with that address space set as PRIMARY. For session objects created by system services invoked by a PC, such session objects would not be addressable by the caller once returning from the service call.

  Guideline: System services invoked by a PC should use the omnipresent token instead of an installation defined PKCS #11 token when creating session objects that are to be owned by the system service.
- For certain multipart PKCS #11 cryptographic operations, ICSF will save session-state information across calls. This state information is scoped to the

PRIMARY address space, similar to the scoping for omnipresent token objects. Such state objects are only addressable to units of work running with that address space set as PRIMARY.

Guideline: If you begin a multipart PKCS #11 cryptographic operation, you must remain running in the same PRIMARY address space in order to continue the operation.

Key types and mechanisms supported

ICSF supports the following PKCS #11 key types (CK_KEY_TYPE). All of these key types are supported in software. Whether they are also supported in hardware will depend on the limitations of your cryptographic hardware configuration.

- CKK_AES - key lengths 128, 192, and 256 bits
- CKK_BLOWFISH - key lengths 8 up to 448 bits (in increments of 8 bits)
- CKK_DES
- CKK_DES2
- CKK_DES3
- CKK_DH - key lengths 512 up to 2048 bits (in increments of 64 bits)
- CKK_DSA - key lengths 512 up to 2048 bit prime lengths (in increments of 64 bits)
- CKK_EC (CKK_ECDSA) - key lengths 160 up to 521 bits
- CKK_GENERIC_SECRET - key lengths 8 up to 2048 bits, unless further restricted by the generation mechanism:
  - CKM_DH_PKCS_DERIVE - key lengths 512 up to 2048 bits
  - CKM_SSL3_MASTER_KEY_DERIVE - 384-bit key lengths
  - CKM_SSL3_MASTER_KEY_DERIVE_DH - 384-bit key lengths
  - CKM_SSL3_PRE_MASTER_KEY_GEN - 384-bit key lengths
  - CKM_TLS_MASTER_KEY_DERIVE - 384-bit key lengths
  - CKM_TLS_MASTER_KEY_DERIVE_DH - 384-bit key lengths
  - CKM_TLS_PRE_MASTER_KEY_GEN - 384-bit key lengths
- CKK_RC4 - key lengths 8 up to 2048 bits
- CKK_RSA - key lengths 512 up to 4096 bits

The following table shows the mechanisms supported by different hardware configurations. All the mechanisms are supported in software, and some may be available in hardware. If the mechanism is available in hardware, ICSF will use the hardware mechanism. If the mechanism is not available in hardware, ICSF will use the software mechanism. The following table also shows the flags returned by the C_GetMechanismInfo function in the CK_MECHANISM_INFO structure. Whether or not the CKF_HW flag is returned in the CK_MECHANISM_INFO structure indicates whether or not the mechanism is supported in the hardware.

Table 3. Mechanism information as returned by C_GetMechanismInfo (CK_MECHANISM_INFO)

Type (CK_MECHANISM_TYPE)	Size factor	Flags
CKM_RSA_PKCS_KEY_PAIR_GEN	Bits	[CKF_HW] CKF_GENERATE_KEY_PAIR
CKM_DES_KEY_GEN	not applicable	[CKF_HW] CKF_GENERATE
CKM_DES2_KEY_GEN	not applicable	[CKF_HW] CKF_GENERATE
CKM_DES3_KEY_GEN	not applicable	[CKF_HW] CKF_GENERATE

Table 3. Mechanism information as returned by C_GetMechanismInfo (CK_MECHANISM_INFO) (continued)

Type (CK_MECHANISM_TYPE)	Size factor	Flags
CKM_RSA_PKCS[6]	Bits	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT CKF_WRAP CKF_UNWRAP CKF_SIGN CKF_VERIFY CKF_SIGN_RECOVER CKF_VERIFY_RECOVER
CKM_RSA_X_509[6, 7]	Bits	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT CKF_SIGN CKF_VERIFY CKF_SIGN_RECOVER CKF_VERIFY_RECOVER
CKM_MD2_RSA_PKCS[6, 7]	Bits	CKF_SIGN CKF_VERIFY
CKM_MD5_RSA_PKCS[6, 7]	Bits	[CKF_HW] CKF_SIGN CKF_VERIFY
CKM_SHA1_RSA_PKCS[6, 7]	Bits	[CKF_HW] CKF_SIGN CKF_VERIFY
CKM_SHA224_RSA_PKCS[6, 7]	Bits	[CKF_HW] CKF_SIGN CKF_VERIFY
CKM_SHA256_RSA_PKCS[6, 7]	Bits	[CKF_HW] CKF_SIGN CKF_VERIFY
CKM_SHA384_RSA_PKCS[6, 7]	Bits	[CKF_HW] CKF_SIGN CKF_VERIFY
CKM_SHA512_RSA_PKCS[6, 7]	Bits	[CKF_HW] CKF_SIGN CKF_VERIFY
CKM_DES_ECB[3]	not applicable	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT
CKM_DES_CBC	not applicable	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT
CKM_DES_CBC_PAD	not applicable	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT CKF_WRAP CKF_UNWRAP
CKM_DES3_ECB[3, 4]	not applicable	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT
CKM_DES3_CBC[4]	not applicable	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT
CKM_DES3_CBC_PAD[4]	not applicable	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT CKF_WRAP CKF_UNWRAP
CKM_SHA_1	not applicable	[CKF_HW] CKF_DIGEST
CKM_SHA224	not applicable	[CKF_HW] CKF_DIGEST
CKM_SHA256	not applicable	[CKF_HW] CKF_DIGEST
CKM_SHA384	not applicable	[CKF_HW] CKF_DIGEST
CKM_SHA512	not applicable	[CKF_HW] CKF_DIGEST
CKM_RIPEMD160	not applicable	CKF_DIGEST
CKM_MD2	not applicable	CKF_DIGEST
CKM_MD5	not applicable	[CKF_HW] CKF_DIGEST
CKM_AES_KEY_GEN	Bytes	[CKF_HW] CKF_GENERATE
CKM_AES_ECB[4]	Bytes	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT
CKM_AES_CBC[4]	Bytes	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT
CKM_AES_CBC_PAD[4]	Bytes	[CKF_HW] CKF_ENCRYPT CKF_DECRYPT CKF_WRAP CKF_UNWRAP
CKM_AES_GCM[4]	Bytes	CKF_ENCRYPT CKF_DECRYPT
CKM_DSA_KEY_PAIR_GEN	Bits	CKF_GENERATE_KEY_PAIR
CKM_DH_PKCS_KEY_PAIR_GEN	Bits	CKF_GENERATE_KEY_PAIR
CKM_EC_KEY_PAIR_GEN	Bits	CKF_GENERATE_KEY_PAIR CKF_EC_F_P[1] CKF_EC_NAMEDCURVE[2] CKF_EC_UNCOMPRESS
CKM_DSA_PARAMETER_GEN	Bits	CKF_GENERATE

Table 3. Mechanism information as returned by C_GetMechanismInfo (CK_MECHANISM_INFO) (continued)

Type (CK_MECHANISM_TYPE)	**Size factor**	**Flags**
CKM_DH_PKCS_PARAMETER_GEN	Bits	CKF_GENERATE
CKM_BLOWFISH_KEY_GEN	Bytes	[CKF_HW] CKF_GENERATE
CKM_RC4_KEY_GEN	Bits	[CKF_HW] CKF_GENERATE
CKM_SSL3_PRE_MASTER_KEY_GEN	Bytes	[CKF_HW] CKF_GENERATE
CKM_TLS_PRE_MASTER_KEY_GEN	Bytes	[CKF_HW] CKF_GENERATE
CKM_GENERIC_SECRET_KEY_GEN	Bits	[CKF_HW] CKF_GENERATE
CKM_BLOWFISH_CBC[5]	Bytes	CKF_ENCRYPT CKF_DECRYPT
CKM_RC4[5]	Bits	CKF_ENCRYPT CKF_DECRYPT
CKM_DSA_SHA1	Bits	CKF_SIGN CKF_VERIFY
CKM_DSA	Bits	CKF_SIGN CKF_VERIFY
CKM_ECDSA_SHA1	Bits	CKF_SIGN CKF_VERIFY CKF_EC_F_P[1] CKF_EC_NAMEDCURVE[2] CKF_EC_UNCOMPRESS
CKM_ECDSA	Bits	CKF_SIGN CKF_VERIFY CKF_EC_F_P[1] CKF_EC_NAMEDCURVE[2] CKF_EC_UNCOMPRESS
CKM_MD5_HMAC	not applicable	CKF_SIGN CKF_VERIFY
CKM_SHA_1_HMAC	not applicable	CKF_SIGN CKF_VERIFY
CKM_SHA224_HMAC	not applicable	CKF_SIGN CKF_VERIFY
CKM_SHA256_HMAC	not applicable	CKF_SIGN CKF_VERIFY
CKM_SHA384_HMAC	not applicable	CKF_SIGN CKF_VERIFY
CKM_SHA512_HMAC	not applicable	CKF_SIGN CKF_VERIFY
CKM_SSL3_MD5_MAC	Bits	CKF_SIGN CKF_VERIFY
CKM_SSL3_SHA1_MAC	Bits	CKF_SIGN CKF_VERIFY
CKM_DH_PKCS_DERIVE	Bits	CKF_DERIVE
CKM_ECDH1_DERIVE	Bits	CKF_DERIVE CKF_EC_F_P[1] CKF_EC_NAMEDCURVE[2] CKF_EC_UNCOMPRESS
CKM_SSL3_MASTER_KEY_DERIVE	Bytes	CKF_DERIVE
CKM_SSL3_MASTER_KEY_DERIVE_DH	Bytes	CKF_DERIVE
CKM_SSL3_KEY_AND_MAC_DERIVE	not applicable	CKF_DERIVE
CKM_TLS_MASTER_KEY_DERIVE	Bytes	CKF_DERIVE
CKM_TLS_MASTER_KEY_DERIVE_DH	Bytes	CKF_DERIVE
CKM_TLS_KEY_AND_MAC_DERIVE	not applicable	CKF_DERIVE
CKM_TLS_PRF	not applicable	CKF_DERIVE

Footnotes for table Table 3 on page 21.

[1] The PKCS11 standard designates two ways of implementing Elliptic Curve Cryptography, nicknamed F_p and F_{2^m}. z/OS PKCS11 supports the F_p variety only.

[2] ANSI X9.62 has the following ASN.1 definition for Elliptic Curve domain parameters:

```
   Parameters ::= CHOICE {
     ecParameters  ECParameters,
     namedCurve    OBJECT IDENTIFIER,
     implicitlyCA  NULL }
```

z/OS PKCS11 supports the specification of CKA_EC_PARAMS attribute using the namedCurved CHOICE. The following NIST-recommended named curves are supported:

- secp192r1 – { 1 2 840 10045 3 1 1 }
- secp224r1 – { 1 3 132 0 33 }
- secp256r1 – { 1 2 840 10045 3 1 7 }
- secp384r1 – { 1 3 132 0 34 }
- secp521r1 – { 1 3 132 0 35 }

The following Brainpool-defined named curves are supported:

- brainpoolP160r1 – { 1 3 36 3 3 2 8 1 1 1 }
- brainpoolP192r1 – { 1 3 36 3 3 2 8 1 1 3 }
- brainpoolP224r1 – { 1 3 36 3 3 2 8 1 1 5 }
- brainpoolP256r1 – { 1 3 36 3 3 2 8 1 1 7 }
- brainpoolP320r1 – { 1 3 36 3 3 2 8 1 1 9 }
- brainpoolP384r1 – { 1 3 36 3 3 2 8 1 1 11 }
- brainpoolP512r1 – { 1 3 36 3 3 2 8 1 1 13 }

In addition, z/OS PKCS11 has limited support for the ecParameters CHOICE. When specified, the DER encoding must contain the optional cofactor field and must not contain the optional Curve.seed field. Also, calls to C_GetAttributeValue to retrieve the CKA_EC_PARAMS attribute will always return the value in the namedCurve form regardless of how the attribute was specified when the object was created. Due to these limitations, the CKF_EC_ECPARAMETERS flag is not turned on for the applicable mechanisms.

[3] Mechanism not present on a CCF system.

[4] Mechanism not present on a system that is export controlled.

[5] Mechanism limited to 56-bit on a system that is export controlled.

[6] In general, z/OS PKCS #11 expects RSA private keys to be in Chinese Remainder Theorem (CRT) format. However, for Decrypt, Sign, or UnwrapKey (z890, z990 or higher only) where one of the following is true, the shorter Modulus Exponent (ME) is permitted:

- There is an accelerator present and the key is less than or equal to 2048 bits in length.
- There is a coprocessor present and the key is less than or equal to 1024 bits in length and FIPS restrictions don't apply.

[7] RSA public or private keys that have a public exponent greater than 8 bytes in length, or a modulus that has an odd number of bits, can only be used when an accelerator is present or a coprocessor is present and FIPS restrictions don't apply. If only an accelerator is present, the key must be less than or equal to 2048 bits in length.

The following table lists the mechanisms supported by specific cryptographic hardware. When a particular mechanism is not available in hardware, ICSF will use the software implementation of the mechanism.

Table 4. Mechanisms supported by specific cryptographic hardware

Machine type and cryptographic hardware	Mechanisms supported	Notes
z900 - CCF	CKM_DES_KEY_GEN CKM_DES2_KEY_GEN CKM_DES3_KEY_GEN CKM_RSA_PKCS CKM_RSA_X_509 CKM_MD5_RSA_PKCS CKM_SHA1_RSA_PKCS CKM_DES_CBC CKM_DES_CBC_PAD CKM_DES3_CBC CKM_DES3_CBC_PAD CKM_SHA_1 CKM_BLOWFISH_KEY_GEN CKM_RC4_KEY_GEN CKM_AES_KEY_GEN CKM_SSL3_PRE_MASTER_KEY_GEN CKM_TLS_PRE_MASTER_KEY_GEN CKM_GENERIC_SECRET_KEY_GEN	This is the base set. RSA private key operations limited to 1024 bits in length (maximum) and no key pair generation capability.
z900 - PCICC	Base set plus: CKM_RSA_PKCS_KEY_PAIR_GEN	RSA private key operations limited to 2048 bits in length (maximum).
z890, z990 - PCIXCC	Base set plus: CKM_RSA_PKCS_KEY_PAIR_GEN CKM_DES_ECB CKM_DES3_ECB	RSA private key operations limited to 2048 bits in length (maximum).
z890, z990 - CEX2C	Base set plus: CKM_RSA_PKCS_KEY_PAIR_GEN CKM_DES_ECB CKM_DES3_ECB	RSA private key operations limited to 2048 bits in length (maximum).
z9® - CEX2C	Base set plus: CKM_RSA_PKCS_KEY_PAIR_GEN CKM_DES_ECB CKM_DES3_ECB CKM_SHA224_RSA_PKCS CKM_SHA256_RSA_PKCS CKM_SHA224 CKM_SHA256 CKM_AES_CBC CKM_AES_CBC_PAD CKM_AES_ECB	AES key operations limited to 128 bits in length (maximum). RSA private key operations limited to 4096 bits in length (maximum).
z10 - CEX2C or CEX3C	z9 CEX2C set plus: CKM_SHA384_RSA_PKCS CKM_SHA512_RSA_PKCS CKM_SHA384 CKM_SHA512	AES key operations limited to 256 bits in length (maximum). RSA private key operations limited to 4096 bits in length (maximum).
z196 - CEX3C	z9 CEX2C set plus: CKM_SHA384_RSA_PKCS CKM_SHA512_RSA_PKCS CKM_SHA384 CKM_SHA512	AES key operations limited to 256 bits in length (maximum). RSA private key operations limited to 4096 bits in length (maximum).

The following table lists the algorithms and uses (by mechanism) that are not allowed when operating in compliance with FIPS 140-2. For more information about how the z/OS PKCS #11 services can be configured to operate in compliance with the FIPS 140-2 standard, refer to “Operating in compliance with FIPS 140-2” on page 11.

Table 5. Restricted algorithms and uses when running in compliance with FIPS 140-2

Algorithm	Mechanisms	Usage disallowed
RIPEMD	CKM_RIPEMD160	All
MD2	CKM_MD2, CKM_MD2_RSA_PKCS	All
MD5	CKM_MD5, CKM_MD5_RSA_PKCS, CKM_MD5_HMAC	All
SSL3	CKM_SSL3_MD5_MAC, CKM_SSL3_SHA1_MAC, CKM_SSL3_MASTER_KEY_DERIVE, CKM_SSL3_MASTER_KEY_DERIVE_DH, CKM_SSL3_KEY_AND_MAC_DERIVE	All
TLS	CKM_TLS_MASTER_KEY_DERIVE, CKM_TLS_MASTER_KEY_DERIVE_DH, CKM_TLS_KEY_AND_MAC_DERIVE	Base key sizes less than 10 bytes
Diffie Hellman	CKM_DH_PKCS_DERIVE	Prime size less than 1024 bits
	CKM_DH_PKCS_PARAMETER_GEN	Prime sizes other than 1024 or 2048 bits
DSA	CKM_DSA_SHA1, CKM_DSA	Prime sizes less than 1024 bits
	CKM_DSA_PARAMETER_GEN, CKM_DSA_KEY_PAIR_GEN or Sign	Combinations other than the following: • Prime size = 1024 bits, subprime size = 160 bits • Prime size = 2048 bits, subprime size = 224 bits, or 256 bits
Single DES	CKM_DES_ECB, CKM_DES_CBC, CKM_DES_CBC_PAD	All
Triple DES	CKM_DES3_ECB, CKM_DES3_CBC, CKM_DES3_CBC_PAD	Two key Triple DES
Blowfish	CKM_BLOWFISH_KEY_GEN, CKM_BLOWFISH_CBC	All
RC4	CKM_RC4	All
RSA	CKM_RSA_X_509	All
	CKM_RSA_PKCS	Key sizes less than 1024 bits
	CKM_RSA_PKCS_KEY_PAIR_GEN or Sign without an active accelerator	Key sizes that are less than 1024 bits or not a multiple of 256 bits or public key exponents less than 0x010001
ECC	CKM_ECDSA, CKM_ECDSA_SHA1, CKM_ECDH1_DERIVE	Brainpool curves
HMAC	CKM_SHA_1, CKM_SHA224, CKM_SHA256, CKM_SHA384, CKM_SHA512	Base key sizes less than one half the output size

Table 5. Restricted algorithms and uses when running in compliance with FIPS 140-2 (continued)

Algorithm	Mechanisms	Usage disallowed
AES GCM	CKM_AES_GCM	GCM encryption or GMAC generation with externally generated initialization vectors. Initialization vector lengths other than 12 bytes. Tag byte sizes 4 and 8

Objects and attributes supported

ICSF supports the following PKCS #11 object types (CK_OBJECT_CLASS):

- CKO_DATA
- CKO_CERTIFICATE - CKC_X_509 only
- CKO_DOMAIN_PARAMETERS - CKK_DSA and CKK_DH only
- CKO_PUBLIC_KEY - CKK_RSA, CKK_EC (CKK_ECDSA), CKK_DSA, and CKK_DH only
- CKO_PRIVATE_KEY - CKK_RSA, CKK_EC (CKK_ECDSA), CKK_DSA, and CKK_DH only
- CKO_SECRET_KEY - CKK_DES, CKK_DES2, CKK_DES3, CKK_AES, CKK_BLOWFISH, and CKK_RC4, CKK_GENERIC_SECRET only

The footnotes described in Table 6 are taken from the PKCS #11 specification and apply to the attribute tables that follow.

Table 6. Common footnotes for object attribute tables

Footnote number	Footnote meaning
1	Must be specified when object is created with **C_CreateObject**.
2	Must *not* be specified when object is created with **C_CreateObject**.
3	Must be specified when object is generated with **C_GenerateKey** or **C_GenerateKeyPair**.
4	Must *not* be specified when object is generated with **C_GenerateKey** or **C_GenerateKeyPair**.
5	Must be specified when object is unwrapped with **C_UnwrapKey**.
6	Must *not* be specified when object is unwrapped with **C_UnwrapKey**.
7	Cannot be revealed if object has its **CKA_SENSITIVE** attribute set to TRUE or its **CKA_EXTRACTABLE** attribute set to FALSE.
8	May be modified after object is created with a **C_SetAttributeValue** call, or in the process of copying object with a **C_CopyObject** call. However, it is possible that a particular token may not permit modification of the attribute, or may not permit modification of the attribute during the course of a **C_CopyObject** call.
9	Default value is token-specific, and may depend on the values of other attributes.
10	Can only be set to TRUE by the SO user.
11	May be changed during a **C_CopyObject** call but not on a **C_SetAttributeValue** call

Table 7. Data object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_CLASS[1]	CKO_OBJECT_CLASS	Object class (type). An application can specify the value when the object is created (or generated) only.
CKA_TOKEN[11]	CK_BBOOL	Default value on create is FALSE. Object hardened to the TKDS if TRUE. An application can specify the value when the object is created (or generated) only.
CKA_PRIVATE[11]	CK_BBOOL	Default value on create is TRUE. An application can specify the value when the object is created (or generated) only.
CKA_MODIFIABLE[11]	CK_BBOOL	Default value is TRUE. An application can specify the value when the object is created (or generated) only.
CKA_LABEL	Printable EBCDIC string	Application-specific nickname. Limited to 32 characters. Default is empty. The string is assumed to come from the IBM1047 code page. An application can set or change the value at any time.
CKA_ID	Byte array	Key or other identifier. Default is empty. An application can set or change the value at any time.
CKA_VALUE	Byte array	Any value. Default is empty. An application can set or change the value at any time.
CKA_APPLICATION	Printable EBCDIC string	Description of the application that created the object. Default is empty. The string is assumed to come from the IBM1047 code page. An application can set or change the value at any time.
CKA_OBJECT_ID	Byte array	DER-encoded OID. Default is empty. An application can set or change the value at any time.

Table 8. X.509 certificate object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_CLASS[1]	CKO_OBJECT_CLASS	Object class (type). An application can specify the value when the object is created (or generated) only.
CKA_TOKEN[11]	CK_BBOOL	Default value on create is FALSE. Object hardened to the TKDS if TRUE. An application can specify the value when the object is created (or generated) only.

Table 8. X.509 certificate object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PRIVATE[11]	CK_BBOOL	Default value on create is FALSE. An application can specify the value when the object is created (or generated) only.
CKA_MODIFIABLE[11]	CK_BBOOL	Default value is TRUE. An application can specify the value when the object is created (or generated) only.
CKA_LABEL	Printable EBCDIC string	Application-specific nickname. Limited to 32 characters. Default is empty. The string is assumed to come from the IBM1047 code page. An application can set or change the value at any time.
CKA_CERTIFICATE_TYPE	CK_CERTIFICATE_TYPE	Always CKC_X_509. An application can specify the value when the object is created (or generated) only.
CKA_TRUSTED	CK_BBOOL	Always set to TRUE. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_SUBJECT	Byte array	DER-encoding as found in certificate. If not specified, ICSF sets it from the certificate. If specified, ICSF enforces that it matches the subject in the certificate. An application can specify the value when the object is created (or generated) only.
CKA_ID	Byte array	Key identifier. Default is empty. An application can set or change the value at any time.
CKA_ISSUER	Byte array	DER-encoding as found in certificate. If not specified, ICSF sets from the certificate. If specified, ICSF enforces that it matches the issuer in the certificate An application can specify the value when the object is created (or generated) only.
CKA_SERIAL_NUMBER	Byte array	DER-encoding as found in certificate. If not specified, ICSF sets from the certificate. If specified, ICSF enforces that it matches the serial number in the certificate. An application can specify the value when the object is created (or generated) only.
CKA_VALUE	Byte array	This is the DER-encoding of the certificate. (Required.) An application can specify the value when the object is created (or generated) only.

Table 8. X.509 certificate object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_CERTIFICATE_CATEGORY	CK_ULONG	Categorization of the certificate: **1** Token user **2** Certificate authority **3** Other entity If not specified, ICSF sets it to 2 if the certificate has the BasicConstraints CA flag on. Otherwise it is not set. **Note:** If specified (or defaulted) to 2, the certificate is considered a CA certificate. The user must have appropriate authority. An application can set or change the value at any time.
CKA_APPLICATION	Printable EBCDIC string	Description of the application that created the object. Default is empty. The string is assumed to come from the IBM1047 code page. An application can specify the value when the object is created (or generated) only.
CKA_IBM_DEFAULT (vendor specific attribute - 0x80000002)	CK_BBOOL	Default flag. Default is FALSE. An application can set or change the value at any time.

Table 9. Secret key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_CLASS[1]	CKO_OBJECT_CLASS	Object class (type). An application can specify the value when the object is created (or generated) only.
CKA_TOKEN[11]	CK_BBOOL	Default value on create is FALSE. Object hardened to the TKDS if TRUE. An application can specify the value when the object is created (or generated) only.
CKA_PRIVATE[11]	CK_BBOOL	Default value on create is TRUE. An application can specify the value when the object is created (or generated) only.
CKA_MODIFIABLE[11]	CK_BBOOL	Default value is TRUE. An application can specify the value when the object is created (or generated) only.
CKA_LABEL	Printable EBCDIC string	Application-specific nickname. Limited to 32 characters. Default is empty. The string is assumed to come from the IBM1047 code page. An application can set or change the value at any time.
CKA_ID	Byte array	Default is empty. An application can set or change the value at any time.

Table 9. Secret key object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_KEY_TYPE[1, 5]	CK_KEY_TYPE	Type of key: CKK_DES, CKK_DES2, CKK_DES3, CKK_BLOWFISH, CKK_RC4, CKK_GENERIC_SECRET, or CKK_AES. An application can specify the value when the object is created (or generated) only.
CKA_START_DATE[8]	CK_DATE	Start date for the key. Default is empty. An application can set or change the value at any time.
CKA_END_DATE[8]	CK_DATE	End date for the key. Default is empty. An application can set or change the value at any time.
CKA_DERIVE[8]	CK_BBOOL	TRUE if key supports key derivation (other keys can be derived from this one). Default is TRUE. An application can set or change the value at any time.
CKA_LOCAL[2, 4, 6]	CK_BBOOL	TRUE only if key was generated locally. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_GEN_MECHANISM[2, 4, 6]	CK_MECHANISM_TYPE	Identifier of the mechanism used to generate the key. Always CK_UNAVAILABLE_INFORMATION. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_ENCRYPT[8]	CK_BBOOL	TRUE if key supports encryption[9]. Default is TRUE. An application can set or change the value at any time.
CKA_VERIFY[8]	CK_BBOOL	TRUE if key supports verification where the signature is an appendix to the data. Default is TRUE. An application can set or change the value at any time.
CKA_WRAP[8]	CK_BBOOL	TRUE if key supports wrapping (can be used to wrap other keys).[9] Default is TRUE. An application can set or change the value at any time.
CKA_DECRYPT[8]	CK_BBOOL	TRUE if key supports decryption.[9] Default is TRUE. An application can set or change the value at any time.
CKA_SIGN[8]	CK_BBOOL	TRUE if key supports signatures where the signature is an appendix to the data.[9] Default is TRUE. An application can set or change the value at any time.

Table 9. Secret key object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_UNWRAP[8]	CK_BBOOL	TRUE if key supports unwrapping (can be used to unwrap other keys)[9]. Default is TRUE. An application can set or change the value at any time.
CKA_EXTRACTABLE[8]	CK_BBOOL	TRUE if key is extractable. Caller can change from TRUE to FALSE only. Default is TRUE. An application can set or change the value, as per PKCS #11 restrictions.
CKA_SENSITIVE[8]	CK_BBOOL	TRUE if key is sensitive. Caller can change from FALSE to TRUE only. Default is FALSE. An application can set or change the value, as per PKCS #11 restrictions.
CKA_ALWAYS_SENSITIVE[2, 4, 6]	CK_BBOOL	TRUE if key has always had the CKA_SENSITIVE attribute set to TRUE. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_NEVER_EXTRACTABLE[2, 4, 6]	CK_BBOOL	TRUE if key has never had the CKA_EXTRACTABLE attribute set to TRUE. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_VALUE[1, 4, 6, 7]	Byte array	The key. Sensitive key part. An application can specify the value when the object is created (or generated) only.
CKA_VALUE_LEN[2, 3]	CK_ULONG	Length of the key in bytes (AES, Blowfish, RC4, and Generic secret keys only). An application can specify the value when the object is generated only.
CKA_APPLICATION	Printable EBCDIC string	Description of the application that created the object. Default is empty. The string is assumed to come from the IBM1047 code page. An application can specify the value when the object is created (or generated) only.
CKA_IBM_FIPS140 (vendor specific attribute 0x80000005)	CK_BBOOL	TRUE if the key must only be used in a FIPS 140-2 compliant fashion. The default value is FALSE. An application can specify the value when the object is created (or generated) only.

Table 10. Public key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_CLASS[1]	CKO_OBJECT_CLASS	Object class (type). An application can specify the value when the object is created (or generated) only.
CKA_TOKEN[11]	CK_BBOOL	Default value on create is FALSE. Object hardened to the TKDS if TRUE. An application can specify the value when the object is created (or generated) only.
CKA_PRIVATE[11]	CK_BBOOL	Default value on create is FALSE. An application can specify the value when the object is created (or generated) only.
CKA_MODIFIABLE[11]	CK_BBOOL	Default value is TRUE. An application can specify the value when the object is created (or generated) only.
CKA_LABEL	Printable EBCDIC string	Application-specific nickname. Limited to 32 characters. Default is empty. The string is assumed to come from the IBM1047 code page. An application can set or change the value at any time.
CKA_TRUSTED	CK_BBOOL	Always set to TRUE. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_SUBJECT	Byte array	DER-encoding. Default empty. An application can set or change the value at any time.
CKA_ID	Byte array	Key identifier. Default empty. An application can set or change the value at any time.
CKA_KEY_TYPE[1, 5]	CK_KEY_TYPE	Type of key. CKK_RSA, CKK_EC, CKK_DSA, and CKK_DH only. An application can specify the value when the object is created (or generated) only.
CKA_START_DATE[8]	CK_DATE	Start date for the key. Default empty. An application can set or change the value at any time.
CKA_END_DATE[8]	CK_DATE	End date for the key. Default empty. An application can set or change the value at any time.
CKA_DERIVE[8]	CK_BBOOL	TRUE if key supports key derivation (if other keys can be derived from this one). Default is TRUE. An application can set or change the value at any time.

Table 10. Public key object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_LOCAL[2, 4, 6]	CK_BBOOL	TRUE only if key was generated locally. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_KEY_GEN_MECHANISM[2, 4, 6]	CK_MECHANISM_TYPE	Identifier of the mechanism used to generate the key. Always CK_UNAVAILABLE_INFORMATION. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_ENCRYPT[8]	CK_BBOOL	TRUE if key supports encryption.[9] Default is TRUE. An application can set or change the value at any time.
CKA_VERIFY[8]	CK_BBOOL	TRUE if key supports verification where the signature is an appendix to the data. Default is TRUE. An application can set or change the value at any time.
CKA_VERIFY_RECOVER[8]	CK_BBOOL	TRUE if key supports verification where the data is recovered from the signature.[9] Default is TRUE. An application can set or change the value at any time.
CKA_WRAP[8]	CK_BBOOL	TRUE if key supports wrapping (can be used to wrap other keys).[9] Default is TRUE. An application can set or change the value at any time.
CKA_APPLICATION	Printable EBCDIC string	Description of the application that created the object. Default is empty. The string is assumed to come from the IBM1047 code page. An application can specify the value when the object is created (or generated) only.
CKA_IBM_FIPS140 (vendor specific attribute 0x80000005)	CK_BBOOL	TRUE if the key must only be used in a FIPS 140-2 compliant fashion. The default value is FALSE. An application can specify the value when the object is created (or generated) only.

Table 11. RSA public key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_MODULUS[1, 4]	Big integer	Modulus *n* An application can specify the value when the object is created (or generated) only.
CKA_MODULUS_BITS[2, 3]	CK_ULONG	Length in bits of modulus *n* An application can specify the value when the object is created (or generated) only.

Table 11. RSA public key object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PUBLIC_EXPONENT[1]	Big integer	Public exponent e An application can specify the value when the object is created (or generated) only.

Table 12. DSA public key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PRIME[1,3]	Big integer	Prime p (512 to 2048 bits in steps of 64 bits)
CKA_SUBPRIME[1,3]	Big integer	Subprime q (160 bits for p <= 1024 bits, 224 bits or 256 bits for p > 1024 bits)
CKA_BASE[1,3]	Big integer	Base g
CKA_VALUE[1,4]	Big integer	Public value y

Table 13. Diffie-Hellman public key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PRIME[1,3]	Big integer	Prime p (512 to 2048 bits in steps of 64 bits)
CKA_BASE[1,3]	Big integer	Base g
CKA_VALUE[1,4]	Big integer	Public value y

Table 14. Elliptic Curve public key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_EC_PARAMS[1,3] (CKA_ECDSA_PARAMS)	Byte Array	DER-encoding of an ANSI X9.62 Parameters value
CKA_EC_POINT[1,4]	Byte Array	DER-encoding of an ANSI X9.62 ECPoint value Q

Table 15. Private key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_CLASS[1]	CKO_OBJECT_CLASS	Object class (type). An application can specify the value when the object is created (or generated) only.
CKA_TOKEN[11]	CK_BBOOL	Default value on create is FALSE. Object hardened to the TKDS if TRUE. An application can specify the value when the object is created (or generated) only.
CKA_PRIVATE[11]	CK_BBOOL	Default value on create is TRUE. An application can specify the value when the object is created (or generated) only.

Table 15. Private key object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_MODIFIABLE[11]	CK_BBOOL	Default value is TRUE. An application can specify the value when the object is created (or generated) only.
CKA_LABEL	Printable EBCDIC string	Application-specific nickname. Limited to 32 characters. Default is empty. The string is assumed to come from the IBM1047 code page. An application can set or change the value at any time.
CKA_SUBJECT	Byte array	DER-encoding. An application can set or change the value at any time.
CKA_ID	Byte array	Default is empty. An application can set or change the value at any time.
CKA_KEY_TYPE[1, 5]	CK_KEY_TYPE	Type of key. CKK_EC, CKK_RSA, CKK_DSA, and CKK_DH only. An application can specify the value when the object is created (or generated) only.
CKA_START_DATE[8]	CK_DATE	Start date for the key. Default empty. An application can set or change the value at any time.
CKA_END_DATE[8]	CK_DATE	End date for the key. Default empty. An application can set or change the value at any time.
CKA_DERIVE[8]	CK_BBOOL	TRUE if key supports key derivation (if other keys can be derived from this one). Default is TRUE. An application can set or change the value at any time.
CKA_LOCAL[2, 4 ,6]	CK_BBOOL	TRUE only if key was generated locally. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_KEY_GEN_ MECHANISM[2, 4, 6]	CK_MECHANISM_TYPE	Identifier of the mechanism used to generate the key material. Always CK_UNAVAILABLE_INFORMATION. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_DECRYPT[8]	CK_BBOOL	TRUE if key supports decryption.[9] Default is TRUE. An application can set or change the value at any time.

Table 15. Private key object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_SIGN[8]	CK_BBOOL	TRUE if key supports signatures where the signature is an appendix to the data.[9] Default is TRUE. An application can set or change the value at any time.
CKA_SIGN_RECOVER[8]	CK_BBOOL	TRUE if key supports signatures where the data can be recovered from the signature.[9] Default is TRUE. An application can set or change the value at any time.
CKA_UNWRAP[8]	CK_BBOOL	TRUE if key supports unwrapping (can be used to unwrap other keys).[9] Default is TRUE. An application can set or change the value at any time.
CKA_EXTRACTABLE[8]	CK_BBOOL	TRUE if key is extractable. Default is TRUE. An application can set or change the value, as per PKCS #11 restrictions. Caller can change from TRUE to FALSE only.
CKA_SENSITIVE[8]	CK_BBOOL	TRUE if key is sensitive. Default is FALSE. An application can set or change the value, as per PKCS #11 restrictions. Caller can change from FALSE to TRUE only.
CKA_ALWAYS_SENSITIVE[2,4, 6]	CK_BBOOL	TRUE if key has always had the CKA_SENSITIVE attribute set to TRUE. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_NEVER_EXTRACTABLE[2,4, 6]	CK_BBOOL	TRUE if key has never had the CKA_EXTRACTABLE attribute set to TRUE. Implicitly set by ICSF. An application cannot directly manipulate this value, but can view it.
CKA_APPLICATION	Printable EBCDIC string	Description of the application that created the object. Default is empty. The string is assumed to come from the IBM1047 code page. An application can specify the value when the object is created (or generated) only.
CKA_IBM_FIPS140 (vendor specific attribute 0x80000005)	CK_BBOOL	TRUE if the key must only be used in a FIPS 140-2 compliant fashion. The default value is FALSE. An application can specify the value when the object is created (or generated) only.

Table 16. RSA private key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_MODULUS[1, 4, 6]	Big integer	Modulus *n* An application can specify the value when the object is created (or generated) only.
CKA_PUBLIC_EXPONENT[4, 6]	Big integer	Public exponent *e* An application can specify the value when the object is created (or generated) only.
CKA_PRIVATE_EXPONENT[1, 4,6 ,7]	Big integer	Private exponent *d* Sensitive key part. An application can specify the value when the object is created (or generated) only.
CKA_PRIME_1[4, 6, 7]	Big integer	Prime *p* Sensitive key part. An application can specify the value when the object is created (or generated) only.
CKA_PRIME_2[4, 6, 7]	Big integer	Prime *q* Sensitive key part. An application can specify the value when the object is created (or generated) only.
CKA_EXPONENT_1[4, 6, 7]	Big integer	Private exponent *d* modulo *p*-1 Sensitive key part. An application can specify the value when the object is created (or generated) only.
CKA_EXPONENT_2[4, 6, 7]	Big integer	Private exponent *d* modulo *q*-1 Sensitive key part. An application can specify the value when the object is created (or generated) only.
CKA_COEFFICIENT[4, 6, 7]	Big integer	CRT coefficient q^{-1} mod *p* Sensitive key part. An application can specify the value when the object is created (or generated) only.

Table 17. DSA private key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PRIME[1,4,6]	Big integer	Prime *p* (512 to 2048 bits in steps of 64 bits)
CKA_SUBPRIME[1,4,6]	Big integer	Subprime *q* (160 bits for p <= 1024 bits, 224 bits or 256 bits for p > 1024 bits)
CKA_BASE[1,4,6]	Big integer	Base *g*
CKA_VALUE[1,4,6,7]	Big integer	Private value *x*

Table 18. Diffie-Hellman private key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PRIME[1,4,6]	Big integer	Prime p (512 to 2048 bits in steps of 64 bits)
CKA_BASE[1,4,6]	Big integer	Base g
CKA_VALUE[1,4,6,7]	Big integer	Private value x
CKA_VALUE_BITS[2,6]	CK_ULONG	Length in bits of private value x. For non-FIPS or when prime bit size = 1024, the default is 160. For FIPS prime bit size = 2048, the default is 256

Table 19. Elliptic Curve private key object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_EC_PARAMS[1,4,6] (CKA_ECDSA_PARAMS)	Byte Array	DER-encoding of an ANSI X9.62 Parameters value
CKA_VALUE[1,4,6,7]	Big integer	ANSI X9.62 private value d

Table 20. Domain parameter object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_CLASS[1]	CKO_OBJECT_CLASS	Object class (type). An application can specify the value when the object is created (or generated) only.
CKA_TOKEN[11]	CK_BBOOL	Default value on create is FALSE. Object hardened to the TKDS if TRUE.
CKA_PRIVATE[11]	CK_BBOOL	Default value on create is FALSE
CKA_MODIFIABLE[11]	CK_BBOOL	Default value is TRUE
CKA_LABEL	Printable EBCDIC string	Application specific nickname. Limit to 32 chars. Default is empty. The string is assumed to come from the IBM1047 code page.
CKA_KEY_TYPE[1]	CK_KEY_TYPE	Type of key the domain parameters can be used to generate. CKK_DSA and CKK_DH only in this release
CKA_LOCAL[2,4]	CK_BBOOL	TRUE only if the parameters were generated locally
CKA_APPLICATION	Printable EBCDIC string	Description of the application that created the object. Default is empty. The string is assumed to come from the IBM1047 code page.

Table 21. DSA domain parameter object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PRIME[1,4]	Big integer	Prime p (512 to 2048 bits in steps of 64 bits)
CKA_SUBPRIME[1,4]	Big integer	Subprime q (160 bits for p <= 1024 bits, 224 bits or 256 bits for p > 1024 bits)
CKA_BASE[1,4]	Big integer	Base g

Table 21. DSA domain parameter object attributes that ICSF supports (continued). For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PRIME_BITS[2,3]	CK_ULONG	Length of the prime value

Table 22. Diffie-Hellman domain parameter object attributes that ICSF supports. For the meanings of the footnotes, see Table 6 on page 27.

Attribute	Data type	Notes
CKA_PRIME[1,4]	Big integer	Prime *p* (512 to 2048 bits in steps of 64 bits)
CKA_BASE[1,4]	Big integer	Base *g*
CKA_PRIME_BITS[2,3]	CK_ULONG	Length of the prime value

Library, slot, and token information

PKCS #11 maintains information about the library code, slots, and tokens, which can be set and queried by calling the library functions. For z/OS this information is as follows:

CK_INFO - Returned by C_GetInfo. not modifiable by applications.

- cryptokiVersion - 2.20
- manufacturerID - “IBM Corp.   ”
- libraryDescription - “z/OS PKCS11 library   ”
- libraryVersion - 7.70

CK_SLOT_INFO - Returned by C_GetSlotInfo. not modifiable by applications

- slotDescription - “z/OS PKCS11 - virtual smart card   ”
- manufacturerID - “IBM Corp.   ”
- flags - for any slot returned by C_GetSlotList the following flags are set:
  - CKF_TOKEN_PRESENT=ON
  - CKF_REMOVABLE_DEVICE=ON
  - CKF_HW_SLOT=OFF
- hardwareVersion - 7.70
- firmwareVersion - 7.70

CK_TOKEN_INFO - Set by C_InitToken . Returned by C_GetTokenInfo

- label - As specified
- manufacturerID - “z/OS PKCS11 API   ” (Might be set to other values if the token was initialized outside of the C API.)
- model - “HCR7770   ” (coincides with the release that the token was created) (Might be set to other values if the token was initialized outside of the C API.)
- serialNumber - “0   ” (Might be set to other values if the token was initialized outside of the C API.)
- flags - the following flags are set ON for any initialized token. All others are OFF:
  - CKF_RNG
  - CKF_PROTECTED_AUTHENTICATION_PATH
  - CKF_TOKEN_INITIALIZED
  - CKF_USER_PIN_INITIALIZED

- ulMaxSessionCount - CK_UNAVAILABLE_INFORMATION
- ulSessionCount - CK_UNAVAILABLE_INFORMATION
- ulMaxRwSessionCount - CK_UNAVAILABLE_INFORMATION
- ulRwSessionCount - CK_UNAVAILABLE_INFORMATION
- ulMaxPinLen - CK_UNAVAILABLE_INFORMATION
- ulMinPinLen - 0
- ulTotalPublicMemory - CK_UNAVAILABLE_INFORMATION
- ulFreePublicMemory - CK_UNAVAILABLE_INFORMATION
- ulTotalPrivateMemory - CK_UNAVAILABLE_INFORMATION
- ulFreePrivateMemory - CK_UNAVAILABLE_INFORMATION
- hardwareVersion - 7.70
- firmwareVersion - 7.70
- utcTime - GMT date and time that the token was last updated

CK_SESSION_INFO - Returned by C_GetSessionInfo

- slotId - The slot in question
- state - CK_UNAVAILABLE_INFORMATION
- flags - As defined by the PKCS #11 specification
- ulDeviceError - A mapping of the last failing ICSF return and reason code values related to this session. For more information see “Function return codes” on page 47.

Functions supported

ICSF supports a subset of the standard PKCS #11 functions, and several non-standard functions.

Standard functions supported

Table 23 lists the standard PKCS #11 functions that ICSF supports. Any function not listed is not supported and returns the CKR_FUNCTION_NOT_SUPPORTED return code.

Table 23. Standard PKCS #11 functions that ICSF supports

Function	Usage notes
General purpose functions	
C_Initialize()	• The library always uses OS locking for thread serialization. Therefore, if C_Initialize is called with the CreateMutex, DestroyMutex, LockMutex, and UnlockMutex function pointer arguments set and the CKF_OS_LOCKING_OK flag is not set, C_Initialize fails and returns the value CKR_CANT_LOCK. • When C_Initialize is called, the application-specific set of (virtual) slot IDs is allocated, one for each preexisting token that the application is authorized to use. (See the descriptions of C_GetSlotList and C_WaitForSlotEvent for information on how this set can increase in size.) The one exception to this occurs when C_Initialize is called by a child process after fork. If the PKCS #11 environment is inherited by the child process, the slot list and token state is not refreshed.

Table 23. Standard PKCS #11 functions that ICSF supports (continued)

Function	Usage notes
C_Finalize()	dlclose() cannot be used as an implicit C_Finalize(). If an application uses dlclose() without calling C_Finalize(), and reinitializes PKCS #11, a subsequent call to C_Initialize() will result in error CKR_FUNCTION_FAILED being returned.
C_GetInfo()	
C_GetFunctionList()	
Slot and token management functions	
C_GetSlotList()	• If the pSlotList argument is NULL, this function returns only the number of allocated slots. In the process of returning this number C_GetSlotList searches for new tokens to which the application has access. If new tokens are found, slot IDs are allocated for them. This search is only performed if at least 5 seconds has passed since the last search was made. • If the pSlotList argument is non-NULL, this function returns the current list of virtual slot IDs. No attempt is made to discover new tokens created by other applications. • The tokenPresent argument flag is meaningless as all allocated slots have a token present.
C_GetSlotInfo()	
C_GetTokenInfo()	
C_WaitForSlotEvent()	• This function is used to dynamically allocate an additional slot in order to create a new token. There are no other slot events. The newly allocated slot ID is returned as the pSlot argument. • The CKF_DONT_BLOCK argument flag is meaningless because this function never blocks. The dynamic slot allocation occurs synchronously.
C_GetMechanismList()	The list of functions returned reflects the capabilities of the current cryptographic hardware configuration. **Note:** The loss or addition of hardware on the fly is not detected or reflected. (For example, on a z9-109, if the only CEX2C present is deactivated, this function still returns the mechanisms that require an active CEX2C to function.)
C_GetMechanismInfo()	The output of this function reflects the capabilities of the current cryptographic hardware configuration.
C_InitToken()	Tokens are protected by the security manager through profiles in the CRYPTOZ class. PINs are not used. The pPin and ulPinLen arguments are ignored.
C_InitPIN()	Tokens are protected by the security manager through profiles in the CRYPTOZ class. PINs are not used. This function performs no operation and always returns CKR_OK.
C_SetPIN()	Tokens are protected by the security manager through profiles in the CRYPTOZ class. PINs are not used. This function performs no operation and always returns CKR_OK.
Session management functions	
C_OpenSession()	The Notify and pApplication arguments are ignored.
C_CloseSession()	

Table 23. Standard PKCS #11 functions that ICSF supports (continued)

Function	Usage notes
C_CloseAllSessions()	
C_GetSessionInfo()	The state field returned is meaningless. It is always set to CK_UNAVAILABLE_INFORMATION .
C_GetOperationState()	Returns CKR_STATE_UNSAVEABLE if a find is active or more than one cryptographic operation is active.
C_SetOperationState()	
C_Login()	Tokens are protected by the security manager through profiles in the CRYPTOZ class. Applications are always logged in to the security manager. PINs are not used. This function has no effect on the session state and always returns CKR_OK.
C_Logout()	Tokens are protected by the security manager through profiles in the CRYPTOZ class. Applications are always logged in to the security manager. PINs are not used. This function has no effect on the session state and always returns CKR_OK.
Object management functions	
C_CreateObject()	
C_CopyObject()	
C_DestroyObject()	
C_GetObjectSize()	
C_GetAttributeValue()	
C_SetAttributeValue()	
C_FindObjectsInit()	
C_FindObjects()	Sensitive attributes cannot be used as search criteria when the object is marked sensitive or not exportable. Doing so results in no match found.
C_FindObjectsFinal()	
Encryption functions	
C_EncryptInit()	The following mechanisms are supported: • CKM_DES_ECB • CKM_DES_CBC • CKM_DES_CBC_PAD • CKM_DES3_ECB • CKM_DES3_CBC • CKM_DES3_CBC_PAD • CKM_RSA_PKCS • CKM_RSA_X_509 • CKM_AES_CBC • CKM_AES_ECB • CKM_AES_CBC_PAD • CKM_AES_GCM (Limited to single part encryption only and for no more than 1048576 bytes of clear text.) • CKM_BLOWFISH_CBC • CKM_RC4
C_Encrypt()	
C_EncryptUpdate()	Multiple-part encryption is not supported for the CKM_RSA_PKCS and CKM_RSA_X_509 mechanisms

Table 23. Standard PKCS #11 functions that ICSF supports (continued)

Function	Usage notes
C_EncryptFinal()	Multiple-part encryption is not supported for the CKM_RSA_PKCS and CKM_RSA_X_509 mechanisms.
Decryption functions	
C_DecryptInit()	The following mechanisms are supported: • CKM_DES_ECB • CKM_DES_CBC • CKM_DES_CBC_PAD • CKM_DES3_ECB • CKM_DES3_CBC • CKM_DES3_CBC_PAD • CKM_RSA_PKCS • CKM_RSA_X_509 • CKM_AES_CBC • CKM_AES_ECB • CKM_AES_CBC_PAD • CKM_AES_GCM (Limited to single part decryption only and for no more than 1048576 bytes of clear text.) • CKM_BLOWFISH_CBC • CKM_RC4
C_Decrypt()	
C_DecryptUpdate()	Multiple-part decryption is not supported for the CKM_RSA_PKCS and CKM_RSA_X_509 mechanisms.
C_DecryptFinal()	Multiple-part decryption is not supported for the CKM_RSA_PKCS and CKM_RSA_X_509 mechanisms.
Message digesting functions	
C_DigestInit()	The following mechanisms are supported: • CKM_MD2 • CKM_MD5 • CKM_SHA_1 • CKM_SHA224 • CKM_SHA256 • CKM_SHA384 • CKM_SHA512 • CKM_RIPEMD160
C_Digest()	
C_DigestUpdate()	
C_DigestFinal()	
Signing and message authentication coding (MACing) functions	

Table 23. Standard PKCS #11 functions that ICSF supports (continued)

Function	Usage notes
C_SignInit()	The following mechanisms are supported: • CKM_RSA_X_509 • CKM_RSA_PKCS • CKM_MD5_RSA_PKCS • CKM_SHA1_RSA_PKCS • CKM_SHA224_RSA_PKCS • CKM_SHA256_RSA_PKCS • CKM_SHA384_RSA_PKCS • CKM_SHA512_RSA_PKCS • CKM_DSA • CKM_DSA_SHA1 • CKM_MD5_HMAC • CKM_SHA_1_HMAC • CKM_SHA224_HMAC • CKM_SHA256_HMAC • CKM_SHA384_HMAC • CKM_SHA512_HMAC • CKM_SSL3_MD5_MAC • CKM_SSL3_SHA1_MAC • CKM_MD2_RSA_PKCS • CKM_ECDSA • CKM_ECDSA_SHA1
C_Sign()	
C_SignUpdate()	Multiple-part signature is not supported for the CKM_RSA_PKCS and CKM_RSA_X_509 mechanisms.
C_SignFinal()	Multiple-part signature is not supported for the CKM_RSA_PKCS and CKM_RSA_X_509 mechanisms.
Functions for verifying signatures and message authentication codes (MACs)	
C_VerifyInit()	The following mechanisms are supported: • CKM_RSA_X_509 • CKM_RSA_PKCS • CKM_MD5_RSA_PKCS • CKM_SHA1_RSA_PKCS • CKM_SHA224_RSA_PKCS • CKM_SHA256_RSA_PKCS • CKM_SHA384_RSA_PKCS • CKM_SHA512_RSA_PKCS • CKM_DSA • CKM_DSA_SHA1 • CKM_MD5_HMAC • CKM_SHA_1_HMAC • CKM_SHA224_HMAC • CKM_SHA256_HMAC • CKM_SHA384_HMAC • CKM_SHA512_HMAC • CKM_SSL3_MD5_MAC • CKM_SSL3_SHA1_MAC • CKM_MD2_RSA_PKCS • CKM_ECDSA • CKM_ECDSA_SHA1
C_Verify()	
C_VerifyUpdate()	Multiple-part verify is not supported for the CKM_RSA_PKCS and CKM_RSA_X_509 mechanisms.

Table 23. Standard PKCS #11 functions that ICSF supports (continued)

Function	Usage notes
C_VerifyFinal()	Multiple-part verify is not supported for the CKM_RSA_PKCS and CKM_RSA_X_509 mechanisms.
Key management functions	
C_DeriveKey()	The following mechanisms are supported: • CKM_DH_PKCS_DERIVE • CKM_SSL3_MASTER_KEY_DERIVE • CKM_SSL3_MASTER_KEY_DERIVE_DH • CKM_SSL3_KEY_AND_MAC_DERIVE • CKM_TLS_MASTER_KEY_DERIVE • CKM_TLS_MASTER_KEY_DERIVE_DH • CKM_TLS_KEY_AND_MAC_DERIVE • CKM_TLS_PRF (It is the caller's responsibility to supply an ASCII value for the seed) • CKM_ECDH1_DERIVE
C_GenerateKey()	The following mechanisms are supported: • CKM_DES_KEY_GEN • CKM_DES2_KEY_GEN • CKM_DES3_KEY_GEN • CKM_AES_KEY_GEN • CKM_DSA_PARAMETER_GEN • CKM_DH_PKCS_PARAMETER_GEN • CKM_BLOWFISH_KEY_GEN • CKM_RC4_KEY_GEN • CKM_GENERIC_SECRET_KEY_GEN • CKM_SSL3_PRE_MASTER_KEY_GEN • CKM_TLS_PRE_MASTER_KEY_GEN
C_GenerateKeyPair()	The following mechanisms are supported: • CKM_RSA_PKCS_KEY_PAIR_GEN • CKM_DSA_KEY_PAIR_GEN • CKM_DH_PKCS_KEY_PAIR_GEN • CKM_EC_KEY_PAIR_GEN
C_WrapKey()	The following mechanism is supported for wrapping secret keys: • CKM_RSA_PKCS The following mechanisms are supported for wrapping private keys: • CKM_DES_CBC_PAD • CKM_DES3_CBC_PAD • CKM_AES_CBC_PAD
C_UnwrapKey()	The following mechanism is supported for unwrapping secret keys: • CKM_RSA_PKCS The following mechanisms are supported for unwrapping private keys: • CKM_DES_CBC_PAD • CKM_DES3_CBC_PAD • CKM_AES_CBC_PAD
Random number generation functions	
C_SeedRandom()	This function always returns the value CKR_RANDOM_SEED_NOT_SUPPORTED because the z/OS hardware random number generator is self-seeding.

Table 23. Standard PKCS #11 functions that ICSF supports (continued)

Function	Usage notes
C_GenerateRandom()	

Non-standard functions supported

The following non-standard function is also supported. Because it is non-standard, it does not appear in the PKCS #11 CK_FUNCTION_LIST structure returned by C_GetFunctionList(). To invoke this function, the caller must either locate the function in the main DLL using dlsym(), or link the application program with the main DLL's sidedeck.

Table 24. Syntax of the CK_RV CSN_FindALLObjects() function

```
CSN_FindALLObjects (
      CK_SESSION_HANDLE         hSession,
      CK_OBJECT_HANDLE_PTR      phObject,
      CK_ULONG                  ulMaxObjectCount
      CK_ULONG_PTR              pulObjectCount
);
```

CSN_FindALLObjects() is identical to C_FindObjects(), except that it uses the ALL rule array keyword when invoking the ICSF CSFPTRL callable service. This can result in CSN_FindALLObjects() returning handles to private objects even if the caller has insufficient SAF authority to view such objects. CSN_FindALLObjects() returns a private key handle (and C_FindObjects does not) when the following conditions are all met:

1. The private object matches the search criteria.
2. No sensitive attributes were specified in the search criteria. The sensitive values for this service are:
   - For a secret key object: CKA_VALUE
   - For Diffie Hellman, DSA, and Elliptic Curve private key objects: CKA_VALUE
   - For an RSA private key object: CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2, CKA_EXPONENT_1, CKA_EXPONENT_2, CKA_COEFFICIENT
3. The caller has only Weak SO or SO R/W permission to the token.

For more information about CSFPTRL processing with respect to the ALL rule array keyword, see *z/OS Cryptographic Services ICSF Application Programmer's Guide*.

Function return codes

In general, the PKCS #11 function return codes are defined in the PKCS #11 specification. However, the following function return codes have a meaning specific to z/OS:

CKR_TOKEN_NOT_PRESENT
ICSF is not running or the TKDS is not properly configured. Note that this return code has no relationship to the slot flag CKF_TOKEN_PRESENT.

CKR_TOKEN_NOT_RECOGNIZED
The caller is not authorized to perform the action requested.

CKR_MECHANISM_INVALID
The specified mechanism is either unknown or not supported by the current cryptographic hardware configuration.

CKR_DEVICE_REMOVED
The token no longer exists. When this error is detected, the token flags are cleared indicating that the token is no longer initialized. It can be re-initialized as a new token, if desired.

Other ICSF-related errors are returned as vendor-defined error codes (CKR_VENDOR_DEFINED). The ICSF return and reason codes are combined into the single return code as follows:

```
#define CKR_IBM_ICSF_ERROR 0xC0000000 /* High order byte mask indicating ICSF error */
#define CKR_IBM_ICSF_ERROR_RET 0x00FF0000 /* Second byte is the return code */
#define CKR_IBM_ICSF_ERROR_RSN 0x0000FFFF /* low order half word is reason code */
```

This mapping is also used to store the ICSF return and reason code values in the CK_SESSION_INFO ulDeviceError field.

The following constants are defined for select ICSF return reason codes:

```
/* ICSF not configured for FIPS mode OR system does not
support FIPS mode */
#define CKR_IBM_ICSF_NOT_FIPS_MODE 0xC0080BFD

/* Algorithm or key size is not valid in FIPS mode */
#define CKR_IBM_ICSF_NOT_VALID_FIPS 0xC0080BFE

/* FIPS known answer tests failed */
#define CKR_IBM_ICSF_FIPS_KAT_FAILED 0xC00C8D3C

/* Service or algorithm not available on current system */
#define CKR_IBM_ICSF_SERV_NOTAVAIL 0xC00C0008
```

Troubleshooting PKCS #11 applications

Note: The information and techniques described in this section are for use primarily by IBM service personnel in determining the cause of a problem with the ICSF PKCS #11 C API.

You can capture trace data using environment variables. To do this, the trace environment variables CSN_PKCS11_TRACE and CSN_PKCS11_TRACE_FILE must be exported prior to the application's first call to any of the PKCS #11 functions.

Table 25. Environment variables for capturing trace data

Environment variable	Usage	Valid values
CSN_PKCS11_TRACE	Specifies the level of tracing to be performed.	An integer value, 1-7. The higher the value, the greater the number of conditions traced. Each level includes the conditions of the levels below it: **7** Debug information **6** Informational conditions **5** Normal but significant conditions **4** Warning conditions **3** Error conditions **2** Critical conditions **1** Immediate action required Any other value causes tracing to be inactive (the default).

CSN_PKCS11_TRACE_FILE	Specifies the name of the trace file. Defaults to /tmp/csnpkcs11.%.trc.
	The current process identifier is included as part of the trace file name when the name contains a percent sign (%). For example, if CSN_PKCS11_TRACE_FILE is set to:
	`/tmp/csnpkcs11.%.trc`
	and the current process identifier is 247, the trace file name is
	`/tmp/csnpkcs11.247.trc`
		Must be set to the full path name of an HFS file in a directory for which the executing application has write permission. The maximum length for the path name is 255 bytes. Values longer than 255 bytes are truncated.

You can also use the utility program, testpkcs11, for troubleshooting. For information about running testpkcs11, see “Running the pre-compiled version of testpkcs11” on page 51.

Chapter 3. The testpkcs11 program

IBM provides a sample PKCS #11 program called testpkcs11 . The program is passed the name of a PKCS #11 token, and performs the following tasks:

1. Creates a token that has the name passed
2. Generates an RSA key-pair
3. Encrypts some test data using the public part of the key-pair
4. Decrypts the data using the private part of the key-pair
5. Deletes the key-pair and the token

You can use this program in several ways:

- As a utility program to test the system configuration for PKCS #11 and troubleshoot problems
- As a sample application to learn how to build and run a PKCS #11 application

IBM provides a pre-compiled version of this program installed in `/usr/lpp/pkcs11/bin`. IBM also provides the C source code for this program in the samples subdirectory, along with three Makefiles:

- Makefile - for 31-bit addressing mode
- Makefile3X - for 31-bit addressing mode with high performance (XPLINK) linkage
- Makefile64 - for 64-bit addressing mode with high performance (XPLINK) linkage

For the source code for this program, see "Source code for the testpkcs11 sample program" on page 59.

Running the pre-compiled version of testpkcs11

If you are testing the system configuration for PKCS #11, or troubleshooting problems with the configuration, you can run the pre-compiled version of testpkcs11.

Steps for running the pre-compiled version of testpkcs11

Before you begin: You need to know how to use z/OS UNIX shells.

Perform the following steps to run the pre-compiled version of testpkcs11.

1. Change to the PKCS #11 bin directory by entering the following command:

   ```
   cd /usr/lpp/pkcs11/bin
   ```

   ____

2. Choose a temporary token name to use. If you need to review the rules for token names, see "Tokens" on page 1.

   ____

3. Run testpkcs11, passing it your token name on the -t option. For example, to use a temporary token name of my.temp.token, enter the following command:

   ```
   ./testpkcs11 -t my.temp.token
   ```

If z/OS PKCS #11 has been set up properly and the you have sufficient authority to the token label specified, you should see the following output:

```
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
```

```
Opening a session...
Generating keys. This may take a while...
Enciphering data...
Deciphering data...
Destroying keys...
Closing the session...
Deleting the temporary token...
Test completed successfully!
```

If you see different messages, there is an error in either your PKCS #11 set up or in the token label that you specified.

The most common user error is specifying token label that is unacceptable to ICSF or already in use. In that case the following is displayed:

```
Getting the PKCS11 function list...
Initializing the PKCS11 environment...
Creating the temporary token...
   C_InitToken #1 returned 7 (0x07) CKR_ARGUMENTS_BAD
   Make sure your the token name you specified meets ICSF rules:
   Contains only alphanumeric characters, nationals (@#$), and periods.
   The first character cannot be a numeric or a period.
```

If you see other error messages, there is probably an error in the setup for the PKCS #11 environment. Determine the error represented by the PKCS #11 error code returned. For information about error codes, see “Function return codes” on page 47.

To display the help text for the testpkcs11 program, run the program with the -h option:

```
cd /usr/lpp/pkcs11/bin
./testpkcs11 -h
```

Building testpkcs11 from source code

If you are learning how to build and run a PKCS #11 application, you can use the source code for testpkcs11 to build and run a sample application.

Steps for building testpkcs11 from source code

Before you begin: You need to know in which directory the PKCS #11 header file is located. By default it is located in the standard include subdirectory under /usr. If your standard include subdirectory it in a different location, you will need to modify the Makefile in step 2. You also need to know how to use z/OS UNIX shells.

Perform the following steps to build the sample application, testpkcs11. Issue the commands from the z/OS UNIX command shell.

1. Copy the testpkcs11.c program and appropriate Makefile to the current directory. For example, for 64-bit addressing enter the following commands:

   ```
   cp /usr/lpp/pkcs11/samples/testpkcs11.c testpkcs11.c
   cp /usr/lpp/pkcs11/samples/Makefile64 Makefile
   ```

2. If the standard include subdirectory is not located under /usr, edit the Makefile copied in step 1 and change the PKCS11_INSTALL_DIR variable as required.

3. Enter the following command to compile and link and produce the executable, testpkcs11:

```
make
```

4. Update your C/C++ environment variable _CEE_RUNOPTS to include XPLINK(ON) if it does not already include it. For example, execute the following command from a UNIX shell:

   ```
   export _CEE_RUNOPTS=$_CEE_RUNOPTS' XPLINK(ON)'
   ```

When you are done, you have built the testpkcs1164 application and can run it in your directory. For example, to run testpkcs1164 in your directory and display its help text, enter the following command:

```
./testpkcs1164 -h
```

To run a test, choose a temporary token name and enter it with the -t option. If you need to review the rules for token names, see “Tokens” on page 1. For example, to use a temporary token name of `my.temp.token`, enter the following command:

```
./testpkcs1164 -t my.temp.token
```

For the output that should appear, see “Steps for running the pre-compiled version of testpkcs11” on page 51.

Chapter 4. ICSF PKCS #11 callable services

The PKCS #11 C language API (described in Chapter 2, “The C API,” on page 19) requires a Language Environment (LE) runtime to operate. Although an LE is normally provided with C application programs, if you are coding your application in some other language (for example, Assembler), acquiring an LE runtime may not be desirable. For these situations, ICSF provides a base set of PKCS #11 callable services that you can use. (In fact, the C API itself uses these services.) These callable services do not require an LE runtime. The ICSF PKCS #11 callable services include:

- Derive key (CSFPDVK)
- Derive multiple keys (CSFPDMK)
- Generate HMAC (CSFPHMG)
- Generate key pair (CSFPGKP)
- Generate secret key (CSFPGSK)
- Get attribute value (CSFPGAV)
- One-way hash generate (CSFPOWH)
- Private key sign (CSFPPKS)
- Pseudo-random function (CSFPPRF)
- Public key verify (CSFPPKV)
- Secret key decrypt (CSFPSKD)
- Secret key encrypt (CSFPSKE)
- Set attribute value (CSFPSAV)
- Token record create (CSFPTRC)
- Token record delete (CSFPTRD)
- Token record list (CSFPTRL)
- Unwrap key (CSFPUWK)
- Verify HMAC (CSFPHMV)
- Wrap key (CSFPWPK)

Calls to the system authorization facility (SAF) determine access authorization for the callable services. The CSFSERV class controls access to the PKCS #11 callable services.

For details about the PKCS #11 callable services, see *z/OS Cryptographic Services ICSF Application Programmer's Guide*.

SMP/E installation data sets, directories, and files

The following dynamic link libraries (DLLs) are linked into SYS1.SIEALNKE:

CSNPCAPI
The main DLL invoked by applications to use PKCS #11 functions. Also shipped as an HFS file at /usr/lpp/pkcs11/lib/csnpcapi.so.

CSNPCA64
64-bit addressing mode version of CSNPCAPI. Also shipped as an HFS file at /usr/lpp/pkcs11/lib/csnpca64.so.

CSNPCA3X
31-bit addressing mode version of CSNPCAPI with XPLINK. Also shipped as an HFS file at /usr/lpp/pkcs11/lib/csnpca3x.so

CSNPCINT
An internal DLL loaded by CSNPCAPI.

CSNPCI64
64-bit addressing mode version of CSNPCINT.

CSNPCI3X
31-bit addressing mode version of CSNPCINT with XPLINK.

CSNPCUTL
An internal DLL implicitly loaded for utilities.

CSNPCU64
64-bit addressing mode version of CSNPCUTL.

CSNPCU3X
31-bit addressing mode version of CSNPCUTL with XPLINK.

CSFDLL31
CSFDLL in 31-bit addressing mode

CSFDLL64
CSFDLL in 64-bit addressing mode.

CSFDLL3X
31-bit addressing mode version of CSFDLL31 with XPLINK.

SMP/E installs the product files into the HFS directory /usr/lpp/pkcs11. This directory contains the following subdirectories and files:

- /usr/lpp/pkcs11/include subdirectory (members are symbolically linked to /usr/include)

  csnpdefs.h
  A header file that applications must include to use PKCS #11 functions. Also copied to SYS1.SIEAHDR.H(CSNPDEFS).

  csfbext.h
  A header file that applications must include to use the CSFDLLs. Also copied to SYS1.SIEAHDR.H(CSFBEXT).

- /usr/lpp/pkcs11/lib subdirectory (members are symbolically linked to /usr/lib)

  CSNPCAPI.x
  Side deck for CSNPCAPI. Also copied to SYS1.SIEASID(CSNPCAPI).

  CSNPCA64.x
  Side deck for CSNPCA64. Also copied to SYS1.SIEASID(CSNPCA64).

CSNPCA3X.x
Side deck for CSNPCA3X. Also copied to SYS1.SIEASID(CSNPCA3X).

csnpcapi.so

csnpca64.so

csnpca3x.so

CSFDLL31.x
Side deck for CSFDLL31. Also copied to SYS1.SIEASID(CSFDLL31).

CSFDLL64.x
Side deck for CSFDLL64. Also copied to SYS1.SIEASID(CSFDLL64).

CSFDLL3X.x
Side deck for CSFDLL3X. Also copied to SYS1.SIEASID(CSFDLL3X).

- usr/lpp/pkcs11/bin subdirectory

  testpkcs11
  Program to test system configuration for PKCS #11.

- /usr/lpp/pkcs11/samples subdirectory

  testpkcs11.c
  Source code for the testpkcs11 program.

  Makefile
  Makefile for the testpkcs11 program.

  Makefile64
  Makefile for 64-bit addressing mode version of the testpkcs11 program.

  Makefile3X
  Makefile for 31-bit addressing mode version of the testpkcs11 program with XPLINK.

Source code for the testpkcs11 sample program

For information about building and using the testpkcs11 sample program, see Chapter 3, “The testpkcs11 program,” on page 51.

```
/*********************************************************************/
/*                                                                   */
/* COMPONENT_NAME: testpkcs11.c                                      */
/*                                                                   */
/* Licensed Materials - Property of IBM                              */
/* 5694-A01                                                          */
/* Copyright IBM Corp. 2007, 2009                                    */
/* Status = HCR7770                                                  */
/*                                                                   */
/*********************************************************************/
/*********************************************************************/
/*                                                                   */
/* This file contains sample code. IBM PROVIDES THIS CODE ON AN      */
/* 'AS IS' BASIS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR     */
/* IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES     */
/* OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.           */
/*                                                                   */
/*********************************************************************/
/*                                                                   */
/*  Change Activity:                                                 */
/*  $L0=P11C1   ,HCR7740, 060124,PDJS: PKCS11 initial release        */
/*  $D1=MG08269 ,HCR7740, 061114,PDJS: Misc fixes                    */
/*  $D2=MG08740 ,HCR7740, 070302,PDGL: XPLINK                        */
/*  $P1=MG13406 ,HCR7770, 090812,PDGL: fix XPLINK define             */
/*  $P2=MG13431 ,HCR7770, 090826,PDER: update prolog                 */
/*                                                                   */
/*********************************************************************/

#ifdef IBM
/* Customers may remove this copyright statement */
#pragma comment (copyright,"\
Licensed Materials - Property of IBM \
5694-A01 Copyright IBM Corp. 2007, 2009 \
All Rights Reserved \
US Government Users Restricted Rights - \
Use, duplication or disclosure restricted by \
GSA ADP Schedule Contract with IBM Corp.")
#endif

/* Install verification test for PKCS #11 */

#define  _UNIX03_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>
#include <dlfcn.h>
#include <sys/timeb.h>
#include <csnpdefs.h>

int skip_token_obj;

CK_FUNCTION_LIST  *funcs;
CK_SLOT_ID        slotID = CK_UNAVAILABLE_INFORMATION;
CK_BYTE           tokenName[32];



void ProcessRetCode( CK_RV rc )
{
   switch (rc) {
      case CKR_OK:                               printf(" CKR_OK");                               break;
      case CKR_CANCEL:                           printf(" CKR_CANCEL");                           break;
      case CKR_HOST_MEMORY:                      printf(" CKR_HOST_MEMORY");                      break;
      case CKR_SLOT_ID_INVALID:                  printf(" CKR_SLOT_ID_INVALID");                  break;
      case CKR_GENERAL_ERROR:                    printf(" CKR_GENERAL_ERROR");                    break;
      case CKR_FUNCTION_FAILED:                  printf(" CKR_FUNCTION_FAILED");                  break;
      case CKR_ARGUMENTS_BAD:                    printf(" CKR_ARGUMENTS_BAD");                    break;
case CKR_NO_EVENT:                         printf(" CKR_NO_EVENT");                         break;
case CKR_NEED_TO_CREATE_THREADS:           printf(" CKR_NEED_TO_CREATE_THREADS");           break;
case CKR_CANT_LOCK:                        printf(" CKR_CANT_LOCK");                        break;
case CKR_ATTRIBUTE_READ_ONLY:              printf(" CKR_ATTRIBUTE_READ_ONLY");              break;
case CKR_ATTRIBUTE_SENSITIVE:              printf(" CKR_ATTRIBUTE_SENSITIVE");              break;
case CKR_ATTRIBUTE_TYPE_INVALID:           printf(" CKR_ATTRIBUTE_TYPE_INVALID");           break;
case CKR_ATTRIBUTE_VALUE_INVALID:          printf(" CKR_ATTRIBUTE_VALUE_INVALID");          break;
case CKR_DATA_INVALID:                     printf(" CKR_DATA_INVALID");                     break;
case CKR_DATA_LEN_RANGE:                   printf(" CKR_DATA_LEN_RANGE");                   break;
case CKR_DEVICE_ERROR:                     printf(" CKR_DEVICE_ERROR");                     break;
case CKR_DEVICE_MEMORY:                    printf(" CKR_DEVICE_MEMORY");                    break;
case CKR_DEVICE_REMOVED:                   printf(" CKR_DEVICE_REMOVED");                   break;
case CKR_ENCRYPTED_DATA_INVALID:           printf(" CKR_ENCRYPTED_DATA_INVALID");           break;
case CKR_ENCRYPTED_DATA_LEN_RANGE:         printf(" CKR_ENCRYPTED_DATA_LEN_RANGE");         break;
case CKR_FUNCTION_CANCELED:                printf(" CKR_FUNCTION_CANCELED");                break;
case CKR_FUNCTION_NOT_PARALLEL:            printf(" CKR_FUNCTION_NOT_PARALLEL");            break;
case CKR_FUNCTION_NOT_SUPPORTED:           printf(" CKR_FUNCTION_NOT_SUPPORTED");           break;
case CKR_KEY_HANDLE_INVALID:               printf(" CKR_KEY_HANDLE_INVALID");               break;
case CKR_KEY_SIZE_RANGE:                   printf(" CKR_KEY_SIZE_RANGE");                   break;
case CKR_KEY_TYPE_INCONSISTENT:            printf(" CKR_KEY_TYPE_INCONSISTENT");            break;
case CKR_KEY_NOT_NEEDED:                   printf(" CKR_KEY_NOT_NEEDED");                   break;
case CKR_KEY_CHANGED:                      printf(" CKR_KEY_CHANGED");                      break;
case CKR_KEY_NEEDED:                       printf(" CKR_KEY_NEEDED");                       break;
case CKR_KEY_INDIGESTIBLE:                 printf(" CKR_KEY_INDIGESTIBLE");                 break;
case CKR_KEY_FUNCTION_NOT_PERMITTED:       printf(" CKR_KEY_FUNCTION_NOT_PERMITTED");       break;
case CKR_KEY_NOT_WRAPPABLE:                printf(" CKR_KEY_NOT_WRAPPABLE");                break;
case CKR_KEY_UNEXTRACTABLE:                printf(" CKR_KEY_UNEXTRACTABLE");                break;
case CKR_MECHANISM_INVALID:                printf(" CKR_MECHANISM_INVALID");                break;
case CKR_MECHANISM_PARAM_INVALID:          printf(" CKR_MECHANISM_PARAM_INVALID");          break;
case CKR_OBJECT_HANDLE_INVALID:            printf(" CKR_OBJECT_HANDLE_INVALID");            break;
case CKR_OPERATION_ACTIVE:                 printf(" CKR_OPERATION_ACTIVE");                 break;
case CKR_OPERATION_NOT_INITIALIZED:        printf(" CKR_OPERATION_NOT_INITIALIZED");        break;
case CKR_PIN_INCORRECT:                    printf(" CKR_PIN_INCORRECT");                    break;
case CKR_PIN_INVALID:                      printf(" CKR_PIN_INVALID");                      break;
case CKR_PIN_LEN_RANGE:                    printf(" CKR_PIN_LEN_RANGE");                    break;
case CKR_PIN_EXPIRED:                      printf(" CKR_PIN_EXPIRED");                      break;
case CKR_PIN_LOCKED:                       printf(" CKR_PIN_LOCKED");                       break;
case CKR_SESSION_CLOSED:                   printf(" CKR_SESSION_CLOSED");                   break;
case CKR_SESSION_COUNT:                    printf(" CKR_SESSION_COUNT");                    break;
case CKR_SESSION_HANDLE_INVALID:           printf(" CKR_SESSION_HANDLE_INVALID");           break;
case CKR_SESSION_PARALLEL_NOT_SUPPORTED:   printf(" CKR_SESSION_PARALLEL_NOT_SUPPORTED");   break;
case CKR_SESSION_READ_ONLY:                printf(" CKR_SESSION_READ_ONLY");                break;
case CKR_SESSION_EXISTS:                   printf(" CKR_SESSION_EXISTS");                   break;
case CKR_SESSION_READ_ONLY_EXISTS:         printf(" CKR_SESSION_READ_ONLY_EXISTS");         break;
case CKR_SESSION_READ_WRITE_SO_EXISTS:     printf(" CKR_SESSION_READ_WRITE_SO_EXISTS");     break;
case CKR_SIGNATURE_INVALID:                printf(" CKR_SIGNATURE_INVALID");                break;
case CKR_SIGNATURE_LEN_RANGE:              printf(" CKR_SIGNATURE_LEN_RANGE");              break;
case CKR_TEMPLATE_INCOMPLETE:              printf(" CKR_TEMPLATE_INCOMPLETE");              break;
case CKR_TEMPLATE_INCONSISTENT:            printf(" CKR_TEMPLATE_INCONSISTENT");            break;
case CKR_TOKEN_NOT_PRESENT:
     printf(" CKR_TOKEN_NOT_PRESENT - ICSF is not active or not configured for TKDS operations"); break;
case CKR_TOKEN_NOT_RECOGNIZED:
     printf(" CKR_TOKEN_NOT_RECOGNIZED - You are not authorized to perform the token operation"); break;
case CKR_TOKEN_WRITE_PROTECTED:            printf(" CKR_TOKEN_WRITE_PROTECTED");            break;
case CKR_UNWRAPPING_KEY_HANDLE_INVALID:    printf(" CKR_UNWRAPPING_KEY_HANDLE_INVALID");    break;
case CKR_UNWRAPPING_KEY_SIZE_RANGE:        printf(" CKR_UNWRAPPING_KEY_SIZE_RANGE");        break;
case CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT: printf(" CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT"); break;
case CKR_USER_ALREADY_LOGGED_IN:           printf(" CKR_USER_ALREADY_LOGGED_IN");           break;
case CKR_USER_NOT_LOGGED_IN:               printf(" CKR_USER_NOT_LOGGED_IN");               break;
case CKR_USER_PIN_NOT_INITIALIZED:         printf(" CKR_USER_PIN_NOT_INITIALIZED");         break;
case CKR_USER_TYPE_INVALID:                printf(" CKR_USER_TYPE_INVALID");                break;
case CKR_USER_ANOTHER_ALREADY_LOGGED_IN:   printf(" CKR_USER_ANOTHER_ALREADY_LOGGED_IN");   break;
case CKR_USER_TOO_MANY_TYPES:              printf(" CKR_USER_TOO_MANY_TYPES");              break;
case CKR_WRAPPED_KEY_INVALID:              printf(" CKR_WRAPPED_KEY_INVALID");              break;
case CKR_WRAPPED_KEY_LEN_RANGE:            printf(" CKR_WRAPPED_KEY_LEN_RANGE");            break;
case CKR_WRAPPING_KEY_HANDLE_INVALID:      printf(" CKR_WRAPPING_KEY_HANDLE_INVALID");      break;
case CKR_WRAPPING_KEY_SIZE_RANGE:          printf(" CKR_WRAPPING_KEY_SIZE_RANGE");          break;
case CKR_WRAPPING_KEY_TYPE_INCONSISTENT:   printf(" CKR_WRAPPING_KEY_TYPE_INCONSISTENT");   break;
case CKR_RANDOM_SEED_NOT_SUPPORTED:        printf(" CKR_RANDOM_SEED_NOT_SUPPORTED");        break;
case CKR_RANDOM_NO_RNG:                    printf(" CKR_RANDOM_NO_RNG");                    break;
case CKR_BUFFER_TOO_SMALL:                 printf(" CKR_BUFFER_TOO_SMALL");                 break;
case CKR_SAVED_STATE_INVALID:              printf(" CKR_SAVED_STATE_INVALID");              break;
case CKR_INFORMATION_SENSITIVE:            printf(" CKR_INFORMATION_SENSITIVE");            break;
case CKR_STATE_UNSAVEABLE:                 printf(" CKR_STATE_UNSAVEABLE");                 break;
case CKR_CRYPTOKI_NOT_INITIALIZED:         printf(" CKR_CRYPTOKI_NOT_INITIALIZED");         break;
case CKR_CRYPTOKI_ALREADY_INITIALIZED:     printf(" CKR_CRYPTOKI_ALREADY_INITIALIZED");     break;
```

```
      case CKR_MUTEX_BAD:                        printf(" CKR_MUTEX_BAD");                        break;
      case CKR_MUTEX_NOT_LOCKED:                 printf(" CKR_MUTEX_NOT_LOCKED");                 break;
      /* Otherwise - Value does not match a known PKCS11 return value */
   }
}



void showError( char *str, CK_RV rc )
{
   printf("%s returned:  %d (0x%0x)", str, rc, rc );
   ProcessRetCode( rc );
   printf("\n");
}

CK_RV createToken( void )
{
   CK_VOID_PTR             p = NULL;  // @D1C
   CK_RV                   rc;
   CK_FLAGS                flags = 0;

   printf("Creating the temporary token... \n");
   /* wait for slot event. On z/OS this creates a new slot synchronously */
   rc = funcs->C_WaitForSlotEvent(flags, &slotID, p);
   if (rc != CKR_OK) {
      showError("   C_WaitForSlotEvent #1", rc );
      return !CKR_OK;
   }
   /* The slot has been created. Now initialize the token in the slot */
   /* On z/OS no PIN is required, so we will pass NULL. */
   rc= funcs->C_InitToken(slotID, NULL, 0, tokenName);
   if (rc != CKR_OK) {
      showError("   C_InitToken #1", rc );
      if (rc == CKR_ARGUMENTS_BAD) {
         printf("   Make sure your the token name you specified meets ICSF rules:\n");
         printf("   Contains only alphanumeric characters, nationals (@#$), and periods.\n");
         printf("   The first character cannot be a numeric or a period.\n");
      }
      return !CKR_OK;
   }

   return CKR_OK;
}

CK_RV deleteToken( void )
{
   CK_VOID_PTR             p;
   CK_RV                   rc;
   CK_FLAGS                flags = 0;

   if (slotID != CK_UNAVAILABLE_INFORMATION) {
     printf("Deleting the temporary token... \n");
     /* C_InitToken with the reserved label $$DELETE-TOKEN$$ is the way to delete a token */
     /* on z/OS */
     memset(tokenName, ' ', sizeof(tokenName));
     memcpy(tokenName, DEL_TOK, sizeof(DEL_TOK));
     rc= funcs->C_InitToken(slotID, NULL, 0, tokenName);
     if (rc != CKR_OK) {
        showError("   C_InitToken #2 (for delete)", rc );
        return !CKR_OK;
     }
   }

   return CKR_OK;
}

CK_RV encryptRSA( void )
{
   CK_BYTE                 data1[100];
   CK_BYTE                 data2[256];
   CK_BYTE                 cipher[256];
   CK_SLOT_ID              slot_id;
   CK_SESSION_HANDLE       session;
   CK_MECHANISM            mech;
   CK_OBJECT_HANDLE        publ_key, priv_key;
   CK_FLAGS                flags;
```

```
CK_ULONG                i;
CK_ULONG                len1, len2, cipherlen;
CK_RV                   rc;
static CK_OBJECT_CLASS  class = CKO_PUBLIC_KEY;
static CK_KEY_TYPE      type= CKK_RSA;
static CK_OBJECT_CLASS  privclass = CKO_PRIVATE_KEY;
static CK_BBOOL         true = TRUE;
static CK_BBOOL         false = FALSE;


static CK_ULONG  bits = 1024;
static CK_BYTE   pub_exp[] = { 0x01, 0x00, 0x01 };

/* Attributes for the public key to be generated */
CK_ATTRIBUTE pub_tmpl[] = {
   {CKA_MODULUS_BITS,    &bits,    sizeof(bits)    },
   {CKA_ENCRYPT,         &true,    sizeof(true)    },
   {CKA_VERIFY,          &true,    sizeof(true)    },
   {CKA_PUBLIC_EXPONENT, &pub_exp, sizeof(pub_exp) }
};

/* Attributes for the private key to be generated */
CK_ATTRIBUTE priv_tmpl[] =
{
   {CKA_DECRYPT,  &true, sizeof(true) },
   {CKA_SIGN,     &true, sizeof(true) }
};

slot_id = slotID;
flags = CKF_SERIAL_SESSION | CKF_RW_SESSION;
printf("Opening a session... \n");
rc = funcs->C_OpenSession( slot_id, flags, (CK_VOID_PTR) NULL, NULL, &session );
if (rc != CKR_OK) {
   showError("   C_OpenSession #1", rc );
   return !CKR_OK;
}


printf("Generating keys. This may take a while... \n");
mech.mechanism      = CKM_RSA_PKCS_KEY_PAIR_GEN;
mech.ulParameterLen = 0;
mech.pParameter     = NULL;

rc = funcs->C_GenerateKeyPair( session,   &mech,
                               pub_tmpl,   4,
                               priv_tmpl,  2,
                               &publ_key, &priv_key );
if (rc != CKR_OK) {
   showError("   C_GenerateKeyPair #1", rc );
   return !CKR_OK;
}

/* now, encrypt some data */
len1      = sizeof(data1);
len2      = sizeof(data2);
cipherlen = sizeof(cipher);

for (i=0; i < len1; i++)
   data1[i] = (i) % 255;

mech.mechanism      = CKM_RSA_PKCS;
mech.ulParameterLen = 0;
mech.pParameter     = NULL;

printf("Enciphering data... \n");
rc = funcs->C_EncryptInit( session, &mech, publ_key );
if (rc != CKR_OK) {
   showError("   C_EncryptInit #1", rc );
   funcs->C_CloseSession( session );
   return !CKR_OK;
}

rc = funcs->C_Encrypt( session, data1, len1, cipher, &cipherlen );
if (rc != CKR_OK) {
   showError("   C_Encrypt #1", rc );
   funcs->C_CloseSession( session );
```

```
      return !CKR_OK;
   }

   /* now, decrypt the data */
   printf("Deciphering data... \n");
   rc = funcs->C_DecryptInit( session, &mech, priv_key );
   if (rc != CKR_OK) {
      showError("   C_DecryptInit #1", rc );
      funcs->C_CloseSession( session );
      return !CKR_OK;
   }

   rc = funcs->C_Decrypt( session, cipher, cipherlen, data2, &len2 );
   if (rc != CKR_OK) {
      showError("   C_Decrypt #1", rc );
      funcs->C_CloseSession( session );
      return !CKR_OK;
   }

   /* PKCS - returns clear data as is */
   if (len1 != len2) {
     printf("   ERROR:  lengths don't match\n");
     printf("   Length of original data = %d, after decryption = %d\n",len1, len2);
     funcs->C_CloseSession( session );
     return !CKR_OK;
   }

   for (i=0; i <len1; i++) {
      if (data1[i] != data2[i]) {
         printf("   ERROR:  mismatch at byte %d\n", i );
         funcs->C_CloseSession( session );
         return !CKR_OK;
      }
   }

   printf("Destroying keys... \n");
   rc = funcs->C_DestroyObject( session, priv_key );
   if (rc != CKR_OK) {
      showError("   C_DestroyObject #1", rc );
      funcs->C_CloseSession( session );
      return !CKR_OK;
   }

   rc = funcs->C_DestroyObject( session, publ_key );
   if (rc != CKR_OK) {
      showError("   C_DestroyObject #2", rc );
      funcs->C_CloseSession( session );
      return !CKR_OK;
   }

   printf("Closing the session... \n");
   rc = funcs->C_CloseSession( session );
   if (rc != CKR_OK) {
      showError("   C_CloseSession #1", rc );
      return !CKR_OK;
   }

   return CKR_OK;
}




CK_RV getFunctionList( void )
{
   CK_RV            rc;
   CK_RV  (*pFunc)();
   void    *d;
#ifdef _LP64
   char    e[]="CSNPCA64";
#elif __XPLINK__                        /* @P1C */
   char    e[]="CSNPCA3X";              /* @D2A */
#else
   char    e[]="CSNPCAPI";
#endif
```

```
   printf("Getting the PKCS11 function list...\n");

   d = dlopen(e,RTLD_NOW);
   if ( d == NULL ) {
      printf("%s not found in linklist or LIBPATH\n",e); // @D1A
      return !CKR_OK;
   }

   pFunc = (CK_RV (*)())dlsym(d,"C_GetFunctionList");
   if (pFunc == NULL ) {
      printf("C_GetFunctionList() not found in module %s\n",e); // @D1A
      return !CKR_OK;
   }
   rc = pFunc(&funcs);

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

   return CKR_OK;

}

void displaySyntax(char *pgm) {
   printf("usage:  %s { -t <token-name> | -h }\n\n", pgm );
   printf(" -t <token-name> = The name of a temporary token to create for the test. The\n");
   printf("  name must be less than 33 characters in length and contains only alphanumeric\n");
   printf("  characters, nationals (@#$), and periods. The first character cannot be a\n");
   printf("  numeric or a period. The token will be deleted when the test is complete.\n\n");
   printf(" -h = Displays this help.\n\n");
}



void main( int argc, char **argv )
{
   CK_C_INITIALIZE_ARGS  cinit_args;
   CK_RV        rc, i;

   memset(tokenName, ' ', sizeof(tokenName)); /* Token name is left justified, padded with blanks */

   if (argc == 3) {
     if (strcmp(argv[1], "-t") == 0)
       if (strlen(argv[2]) > 0 && strlen(argv[2]) < 33) {
         memcpy(tokenName, argv[2], strlen(argv[2]));
       }
       else {
         displaySyntax(argv[0]);
         return;
       }
     else {
       displaySyntax(argv[0]);
       return;
     }
   }
   else {
     displaySyntax(argv[0]);
     return;
   }

   rc = getFunctionList();
   if (rc != CKR_OK) {
      printf("getFunctionList failed!\n"); // @D1C
      return;
   }

   memset( &cinit_args, 0x0, sizeof(cinit_args) );
   cinit_args.flags = CKF_OS_LOCKING_OK;

   printf("Initializing the PKCS11 environment...\n");
   rc = funcs->C_Initialize( &cinit_args );
   if (rc != CKR_OK) {
      showError("   C_Initialize", rc );
      return;
   }

   rc = createToken();
   if (rc != CKR_OK) {
      funcs->C_Finalize( NULL );
      return;
   }

   rc = encryptRSA();
   if (rc != CKR_OK) {
      deleteToken();
      funcs->C_Finalize( NULL );
      return;
   }

   rc = deleteToken();
   if (rc != CKR_OK) {
      funcs->C_Finalize( NULL );
      return;
   }

   rc = funcs->C_Finalize( NULL );
   if (rc != CKR_OK) {
      showError("   C_Initialize", rc );
      return;
   }
   printf("Test completed successfully!\n");

}
```

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

`http://www.ibm.com/systems/z/os/zos/bkserv/`

Notices

This information was developed for products and services offered in the USA.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This publication documents intended programming interfaces that allow the customer to write programs to obtain the services of z/OS Integrated Cryptographic Services Facility (ICSF).

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Product Number: 5694-A01

Printed in USA

SA23-2231-04