Linux on System z

# libica Programmer's Reference

*Version 2.4*

SC34-2602-06

Linux on System z

# libica Programmer's Reference

*Version 2.4*

SC34-2602-06

**Note**

Before using this document, be sure to read the information in "Notices" on page 167.

**Edition notice**

This edition applies to the libica token version 2.4 for openCryptoki version 3.1 and to all subsequent releases and modifications until otherwise indicated in new editions.

# Contents

# Summary of changes

This revision reflects changes to the Development stream for libica version 2.4.

## Updates for libica version 2.4

### Edition SC34-2602-06

*New information*

- An enhanced version of the **icastats** utility collects statistical data per users, not per system. The data is persistently available beyond the context of a single process. See "icastats - Show use of libica functions" on page 78.
- An improved version of the **icainfo** function shows whether the supported cryptographic algorithms are implemented by hardware, software or both. See "icainfo - Show available libica functions" on page 77.

## Updates for libica version 2.3.0

There are two editions of this publication for libica version 2.3.0.

### Edition SC34-2602-05

*New information*

- An example of the openCryptoki configuration file has been added, see "Adjusting the openCryptoki configuration file" on page 65
- New cryptographic mechanisms are implemented for the ica token as of openCryptoki version 3.0, see "Supported mechanisms for the ica token" on page 70

### Edition SC34-2602-04

*New information*

- New API added. See "ica_get_functionlist" on page 59.
- New defines and structures have been added. See Chapter 5, "libica constants, type definitions, data structures, and return codes," on page 73

## Updates for libica version 2.2.0

*New information*

- Cryptographic hardware support with openCryptoki
- New APIs have been added:
  - ica_3des_cbc_cs
  - ica_3des_cmac
  - ica_3des_cmac_intermediate
  - ica_3des_cmac_last
  - ica_aes_cbc_cs
  - ica_aes_ccm
  - ica_aes_cmac_intermediate
  - ica_aes_cmac_last

  - ica_aes_gcm
  - ica_des_cbc_cs
  - ica_des_cmac
  - ica_des_cmac_intermediate
  - ica_des_cmac_last
- New commands have been added. See Chapter 6, “libica tools,” on page 77.

*Changed information*

- Minor changes and corrections have been made to some of the APIs.

*Deleted information*

- Some obsolete examples have been removed.

# About this document

This document describes how to install and use version 2.4 of the Library for IBM® Cryptographic Architecture (libica).

libica version 2.4 is a library of cryptographic functions used to write cryptographic applications on IBM System z®, both with and without cryptographic hardware.

You can find the latest version of this document on the developerWorks® website at:

www.ibm.com/developerworks/linux/linux390/documentation_dev.html

and on the IBM Knowledge Center at:

ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_lib.html

## How this document is organized

The information is divided into topics that describe installing, configuring and using libica together with descriptions of the functions and example programs.

Chapter 1, “General information about libica,” on page 1 has general information about the current libica version.

Chapter 2, “Installing and using libica version 2.4,” on page 5 contains installation and set up instructions, and coexistence information for the current libica version.

Chapter 3, “libica version 2.4 application programming interfaces,” on page 7 describes the libica APIs.

Chapter 4, “Accessing libica functions through the PKCS #11 (openCryptoki),” on page 61 describes how the cryptographic functions provided by libica can be accessed using the PKCS #11 API implemented by openCryptoki.

Chapter 5, “libica constants, type definitions, data structures, and return codes,” on page 73 lists the defines, typedefs, structs, and return codes for libica.

Chapter 6, “libica tools,” on page 77 contains tools to investigate the capabilities of your cryptographic hardware and how these capabilities are used by applications that use libica.

Chapter 7, “Examples,” on page 81 is a set of programming examples that use the libica APIs.

## Who should read this document

This document is intended for C programmers that want to access IBM System z hardware support for cryptographic methods.

In particular, this document addresses programmers who write hardware-specific plug-ins for cryptographic libraries such as openssl and openCryptoki.

## Assumptions

The following general assumptions are made about your background knowledge:

- You have an understanding of basic computer architecture, operating systems, and programs.
- You have an understanding of Linux and IBM System z terminology.
- You have knowledge about cryptographic applications and solution design, as well as the required cryptographic functions and algorithms.

# Distribution independence

This publication does not provide information that is specific to a particular Linux distribution.

The tools it describes are distribution independent.

# Other Linux on System z publications

You can find Linux on System z publications on developerWorks and on the IBM Knowledge Center.

These publications are available on developerWorks at

www.ibm.com/developerworks/linux/linux390/documentation_dev.html

- *Device Drivers, Features, and Commands*, SC33-8411
- *Using the Dump Tools*, SC33-8412
- *How to Improve Performance with PAV*, SC33-8414
- *How to use FC-attached SCSI devices with Linux on System z*, SC33-8413
- *How to use Execute-in-Place Technology with Linux on z/VM®*, SC34-2594
- *How to Set up a Terminal Server Environment on z/VM*, SC34-2596
- *Kernel Messages*, SC34-2599
- *libica Programmer's Reference*, SC34-2602
- *Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*, SC33-8294
- *Exploiting Enterprise PKCS #11 using openCryptoki*, SC34-2713
- *Linux on System z Troubleshooting*, SC34-2612
- *Linux Health Checker User's Guide*, SC34-2609

These publications are available on the IBM Knowledge Center at

ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_lib.html

- *libica Programmer's Reference*, SC34-2602
- *Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*, SC33-8294
- *Exploiting Enterprise PKCS #11 using openCryptoki*, SC34-2713
- *Linux Health Checker User's Guide*, SC34-2609
- *Linux on System z Troubleshooting*, SC34-2612
- *Kernel Messages*, SC34-2599

# Chapter 1. General information about libica

The libica library provides hardware support (and software fallbacks if the hardware is not available) for cryptographic functions. Version numbering libica 2.4 is used throughout this document, which is valid for all available libica versions 2.4.x., because the changes in versions later than 2.4.0 are not relevant for user documentation.

The cryptographic adapters are used for asymmetric encryption and decryption. The CPACF instructions are used for symmetric encryption and decryption, pseudo random number generation, message authentication, and Secure Hashing. For some of these functions, if the hardware is not available or failed, libica uses the low-level cryptographic functions of OpenSSL, if available.

This product includes software that is developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org). This product includes cryptographic software that is written by Eric Young (eay@cryptsoft.com).

The libica library is part of the openCryptoki project in SourceForge. It is primarily used by OpenSSL through the IBM OpenSSL CA engine or by openCryptoki through the ica_s390 token. A higher level of security can be achieved by using it through the PKCS11 API implemented by openCryptoki.

The libica library works only on IBM System z hardware.

IBM reserves the right to change or modify this API at any time. However, an effort is made to keep the API compatible with later versions within a major release.

You can use the **icastats** utility to obtain statistics about cryptographic processes. The **icainfo** command shows whether libica is using cryptographic hardware or software fallback for each specific libica function. See “icastats - Show use of libica functions” on page 78 and “icainfo - Show available libica functions” on page 77 for more information.

## libica examples

There is a list of sample programs in the libica source for each API, as well as instructions about how to use the functions.

You can find the open source version of libica at:

`http://sourceforge.net/projects/opencryptoki/files/libica`

Sample programs area also in Chapter 7, “Examples,” on page 81.

## System z cryptographic hardware support

The following lists different types of cryptographic hardware support that might be available in a System z server.

### IBM CP Assist for Cryptographic Function (CPACF):

DES, TDES, AES128, AES192, AES256, SHA-1, SHA224, SHA256, SHA384, SHA512, PRNG

### Cryptographic cards:

**Accelerator:** RSA (CRT, MOD-EXPO) 1024, 2048 and 4096 bit key size

**CCA Co-processor:** RSA (CRT, MOD-EXPO) 1024, 2048 and 4096 bit key size, RNG

# Check the prerequisites: cryptographic adapter and device driver

To exploit hardware support of asymmetric cryptographic operations, you need a loaded device driver and an installed IBM cryptographic adapter.

## Loading the Linux zcrypt device driver

You also need an installed Linux kernel that includes the zcrypt device driver.

To check, enter the command:

```
$ lszcrypt
card06: CEX3A
```

If the following error message is displayed, load the zcrypt device driver main module:

```
error - cryptographic device driver zcrypt is not loaded!
```

The zcrypt device driver is no longer monolithic as in older distributions where the module was called z90crypt. The device driver is now loaded as separate modules, where the main module is called ap. There is, however, an alias name z90crypt that links to the ap main module.

To load the device driver ap main module, use the following command:

**`modprobe ap`**

See your Linux distribution documentation for how to load the module persistently.

## Checking the cryptographic adapter availability

Check whether you have plugged in and enabled your IBM cryptographic adapter and validate your model and type configuration (accelerator or coprocessor). Use the **`lszcrypt`** command to retrieve basic status information.

To check, enter the command:

```
$ lszcrypt
card06: CEX3A
```

Use the **`chzcrypt`** command to enable (online state) or disable (offline state) the IBM crypto adapter:

```
$ chzcrypt -e 0x06    // set card06 online
$ chzcrypt -d 0x06    // set card06 offline
```

For more information about the IBM crypto adapter with Linux on System z, see *Device Drivers, Features, and Commands*, SC33-8411 available at

`www.ibm.com/developerworks/linux/linux390/documentation_dev.html`

# Chapter 2. Installing and using libica version 2.4

View the contained subtopics for information about where to obtain the libica version 2.4 library (any 2.4.x version), and how to install it.

## Installing libica version 2.4 from the libica RPM

To make use of the libica hardware support of cryptographic functions, it is necessary to install the libica version 2.4 package. Obtain the current libica version 2.4.x from the SourceForge website.

The website is at:

`http://sourceforge.net/projects/opencryptoki/files/libica`

### Before you begin

Follow the installation instructions on the mentioned website to download the libica version 2.4 package and then follow the instructions in this topic or in topic "Installing libica version 2.4 from the source package" to install libica version 2.4.

### Procedure

The libica library is available as an RPM named `libica-`*`<version>`*. See your Linux distribution documentation for how to install an RPM. To check whether the libica library is installed, issue, for example:

```
# rpm -qa | grep -i libica
```

## Installing libica version 2.4 from the source package

If you prefer you can install the source package.

### Procedure

1. Download the latest libica version 2.4 sources from:

   `http://sourceforge.net/projects/opencryptoki/files/libica`
2. Extract the tar archive. There should be a new directory named `libica-2.x.x`.
3. Change to that directory and execute the following scripts and commands:

   ```
   $ ./bootstrap
   $ ./configure
   $ make
   $ make install
   ```

   where:

   **bootstrap**
   Initial setup, basic configurations

   **configure**
   Check configurations and build the Makefile

   **make** Compile and link

 

**make install**
Install the libraries

## Using libica version 2.4

The function prototypes are provided in the header file, `include/ica_api.h`.

Applications using these functions must link libica and libcrypto. The libcrypto library is available from the OpenSSL package. You must have OpenSSL in order to run libica version 2.4 programs.

## libica version 1, version 2, version 2.1.0, and up to version 2.4 coexistence

Some of the libica version 1 APIs are available in libica version 2, libica version 2.1.0, up to libica version 2.4.

Some of them, such as those APIs that work with an environment other than Linux on IBM System z, were removed and are not present in libica version 2 or later versions. If your application program has calls to libica version 1 APIs, check to see whether these APIs are available in libica version 2.4. If they are, these API calls still work. However, we suggest that you convert your application to use the equivalent libica version 2.4 functions. See Chapter 3, “libica version 2.4 application programming interfaces,” on page 7.

libica key generation is restricted to the limits imposed by the OpenSSL implementation. Thus, the value of a public exponent passed to libica cannot be greater than the maximum value that would fit in an unsigned long integer.

# Chapter 3. libica version 2.4 application programming interfaces

A list of application programming interfaces (APIs) for libica version 2.4.

Table 1 lists the APIs for libica version 2.4.

*Table 1. libica version 2.4 APIs*

| Function | libica version 2.4 API name | Key length in bits | Supported on | | | CPACF function | Software fallback |
|---|---|---|---|---|---|---|---|
| | | | z9® | z10™ | z196 | | |
| **Open and close adapter functions** | | | | | | | |
| Open adapter handle | "ica_open_adapter" on page 10 | N/A | Yes | Yes | Yes | No | N/A |
| Close adapter handle | "ica_close_adapter" on page 10 | N/A | Yes | Yes | Yes | No | N/A |
| **Secure hash operations** | | | | | | | |
| Secure hash using the SHA-1 algorithm. | "ica_sha1" on page 11 | N/A | Yes | Yes | Yes | Yes | Yes |
| Secure hash using the SHA-224 algorithm. | "ica_sha224" on page 12 | N/A | No | Yes | Yes | Yes | Yes |
| Secure hash using the SHA-256 algorithm. | "ica_sha256" on page 13 | N/A | Yes | Yes | Yes | Yes | Yes |
| Secure hash using the SHA-384 algorithm. | "ica_sha384" on page 14 | N/A | No | Yes | Yes | Yes | Yes |
| Secure hash using the SHA-512 algorithm. | "ica_sha512" on page 15 | N/A | No | Yes | Yes | Yes | Yes |
| **Random number generation** | | | | | | | |
| Generate a pseudo random number. | "ica_random_number_generate" on page 16 | N/A | Yes | Yes | Yes | Yes | Yes |
| **RSA key generation functions** | | | | | | | |
| Generate RSA keys in modulus/exponent format. | "ica_rsa_key_generate_mod_expo" on page 17 | N/A | Yes | Yes | Yes | No | Software only |
| Generate RSA keys in CRT format. | "ica_rsa_key_generate_crt" on page 17 | N/A | Yes | Yes | Yes | No | Software only |
| **RSA encryption and decryption operations** | | | | | | | |
| RSA encryption and decryption operation using a key in modulus/exponent format. | "ica_rsa_mod_expo" on page 18 | Depending on supported key size of Crypto Express feature | Yes | Yes | Yes | No | Key length maximum 4 K bits |
| RSA encryption and decryption operation using a key in Chinese-Remainder Theorem (CRT) format. | "ica_rsa_crt" on page 19 | Depending on supported key size of Crypto Express feature | Yes | Yes | Yes | No | Key length maximum 4 K bits |

 

*Table 1. libica version 2.4 APIs (continued)*

| Function | libica version 2.4 API name | Key length in bits | Supported on | | | CPACF function | Software fallback |
|---|---|---|---|---|---|---|---|
| | | | z9® | z10™ | z196 | | |
| **DES functions** | | | | | | | |
| DES with Cipher Block Chaining mode | "ica_des_cbc" on page 20 | 56 | Yes | Yes | Yes | Yes | Yes |
| DES with CBC-Cipher text stealing mode | "ica_des_cbc_cs" on page 21 | 56 | Yes | Yes | Yes | Yes | Yes |
| DES with Cipher Feedback mode | "ica_des_cfb" on page 22 | 56 | No | No | Yes | Yes | No |
| DES with CMAC mode | "ica_des_cmac" on page 23 | 56 | No | No | Yes | Yes | No |
| DES with CMAC mode process intermediate chunks | "ica_des_cmac_intermediate" on page 24 | 56 | No | No | Yes | Yes | No |
| DES with CMAC mode process last chunk | "ica_des_cmac_last" on page 25 | 56 | No | No | Yes | Yes | No |
| DES with Counter mode | "ica_des_ctr" on page 26 | 56 | No | No | Yes | Yes | No |
| DES with Counter mode, using a list of counters | "ica_des_ctrlist" on page 28 | 56 | No | No | Yes | Yes | No |
| DES with Electronic Codebook mode. | "ica_des_ecb" on page 29 | 56 | Yes | Yes | Yes | Yes | Yes |
| DES with Output Feedback mode | "ica_des_ofb" on page 29 | 56 | No | No | Yes | Yes | No |
| **TDES/3DES functions** | | | | | | | |
| TDES with Cipher Block Chaining mode | "ica_3des_cbc" on page 32 | 168 | Yes | Yes | Yes | Yes | Yes |
| TDES with CBC-Cipher text Stealing mode | "ica_3des_cbc_cs" on page 32 | 168 | Yes | Yes | Yes | Yes | Yes |
| TDES with Cipher Feedback mode | "ica_3des_cfb" on page 34 | 168 | No | No | Yes | Yes | No |
| TDES with CMAC mode | "ica_3des_cmac" on page 35 | 168 | No | No | Yes | Yes | No |
| TDES with CMAC mode process intermediate chunks | "ica_3des_cmac_intermediate" on page 36 | 168 | No | No | Yes | Yes | No |
| TDES with CMAC mode process last chunk | "ica_3des_cmac_last" on page 37 | 168 | No | No | Yes | Yes | No |
| TDES with Counter mode | "ica_3des_ctr" on page 38 | 168 | No | No | Yes | Yes | No |
| TDES with Counter mode, using a list of counters | "ica_3des_ctrlist" on page 39 | 168 | No | No | Yes | Yes | No |
| TDES with Electronic Codebook mode | "ica_3des_ecb" on page 40 | 168 | Yes | Yes | Yes | Yes | Yes |
| TDES with Output Feedback mode | "ica_3des_ofb" on page 41 | 168 | No | No | Yes | Yes | No |
| **AES functions** | | | | | | | |
| AES with Cipher Block Chaining mode. | "ica_aes_cbc" on page 43 | 128, 192, 256 | Yes | Yes | Yes | Yes | Yes |

Table 1. libica version 2.4 APIs (continued)

| Function | libica version 2.4 API name | Key length in bits | Supported on z9® | z10™ | z196 | CPACF function | Software fallback |
|---|---|---|---|---|---|---|---|
| AES with CBC-Cipher text stealing mode. | "ica_aes_cbc_cs" on page 44 | 128, 192, 256 | Yes | Yes | Yes | Yes | Yes |
| AES with Counter with Cipher Block Chaining - Message Authentication Code mode. | "ica_aes_ccm" on page 45 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with Cipher Feedback mode. | "ica_aes_cfb" on page 47 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with CMAC mode | "ica_aes_cmac" on page 48 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with CMAC mode process intermediate chunks | "ica_aes_cmac_intermediate" on page 49 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with CMAC mode process last chunk | "ica_aes_cmac_last" on page 50 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with Counter mode. | "ica_aes_ctr" on page 51 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with Counter mode, using a list of counters | "ica_aes_ctrlist" on page 52 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with Electronic Codebook mode. | "ica_aes_ecb" on page 53 | 128, 192, 256 | Yes | Yes | Yes | Yes | Yes |
| AES with Galois / Counter mode. | "ica_aes_gcm" on page 54 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with Output Feedback mode. | "ica_aes_ofb" on page 56 | 128, 192, 256 | No | No | Yes | Yes | No |
| AES with XEX-based Tweaked CodeBook mode (TCB) with CipherText Stealing (CTS). | "ica_aes_xts" on page 57 | 128, 256 | No | No | Yes | Yes | No |
| **Information retrieval functions** | | | | | | | |
| Return version information for libica. | "ica_get_version" on page 59 | N/A | Yes | Yes | Yes | No | N/A |
| Return a list of crypto mechanisms supported by libica. | "ica_get_functionlist" on page 59 | N/A | Yes | Yes | Yes | No | N/A |

## Open and close adapter functions

These functions open or close the crypto adapter. It is recommended to open the crypto adapter before using any of the libica crypto functions, and to close it after the last usage of the libica crypto functions. However, in this version of the libica only the RSA-related functions **ica_rsa_mod_expo** and **ica_rsa_crt** require a valid adapter handle as input. A pointer to the value DRIVER_NOT_LOADED indicates an invalid adapter handle. The parameter *ica_adapter_handle_t* is a redefine of int.

These functions are included in: `include/ica_api.h`.

## ica_open_adapter

### Purpose

Opens an adapter.

### Format

```
unsigned int ica_open_adapter(ica_adapter_handle_t *adapter_handle);
```

### Parameters

**`ica_adapter_handle_t *adapter_handle`**
Pointer to the file descriptor for the adapter or to DRIVER_NOT_LOADED if opening the crypto adapter failed.

Opening an adapter succeeds if a cryptographic device is accessible for reading and writing. By default, cryptographic access must be available with the **/dev/z90crypt** path name for the adapter open request to succeed. If the environment variable LIBICA_CRYPT_DEVICE is set to a valid path name of an accessible cryptographic device, accessing the device with that path name takes precedence over the default path names.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_close_adapter

### Purpose

Closes an adapter.

### Comments

This API closes a device handle.

### Format

```
unsigned int ica_close_adapter(ica_adapter_handle_t adapter_handle);
```

### Parameters

**`ica_adapter_handle_t adapter_handle`**
Pointer to a previously opened device handle.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# Secure hash operations

These functions are included in: `include/ica_api.h`.

These functions perform secure hash on input data using the chosen algorithm of SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512.

SHA context structs contain information about how much of the actual work was already performed. Also, it contains the part of the hash that is already produced. For the user, it is only interesting in cases where the message is not hashed at once, because the context is needed for further operations.

# ica_sha1

## Purpose

Performs a secure hash operation on the input data using the SHA-1 algorithm.

## Format

```
unsigned int ica_sha1(unsigned int message_part,
  unsigned int input_length,
  unsigned char *input_data,
  sha_context_t *sha_context,
  unsigned char *output_data);
```

## Required hardware support

KIMD-SHA-1, or KLMD-SHA-1

## Parameters

**`unsigned int message_part`**
The message chaining state. This parameter must be one of the following values:

**SHA_MSG_PART_ONLY**
A single hash operation

**SHA_MSG_PART_FIRST**
The first part

**SHA_MSG_PART_MIDDLE**
The middle part

**SHA_MSG_PART_FINAL**
The last part

**`unsigned int input_length`**
Length in bytes of the input data to be hashed using the SHA-1 algorithm.

**`unsigned char *input_data`**
Pointer to the input data to be hashed. This pointer must not be zero. So even in case of zero size message data, it must be set to a valid value.

**`sha_context_t *sha_context`**
Pointer to the SHA-1 context structure used to store intermediate values needed when chaining is used. The contents are ignored for message part SHA_MSG_PART_ONLY and SHA_MSG_PART_FIRST. This structure must contain the returned value of the preceding call to **ica_sha1** for message part SHA_MSG_PART_MIDDLE and SHA_MSG_PART_FINAL. For message part SHA_MSG_PART_FIRST and SHA_MSG_PART_FINAL, the returned value can be used for a chained call of **ica_sha1**. Therefore, the application must not modify the contents of this structure in between chained calls.

**`unsigned char *output_data`**
Pointer to the buffer to contain the resulting hash data. The resulting output data has a length of **SHA_HASH_LENGTH**. Make sure that the buffer is at least this size.

## Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_sha224

## Purpose

Performs a secure hash operation on the input data using the SHA-224 algorithm.

## Format

```
unsigned int ica_sha224(unsigned int message_part,
  unsigned int input_length,
  unsigned char *input_data,
  sha256_context_t *sha256_context,
  unsigned char *output_data);
```

## Required hardware support

KIMD-SHA-256, or KLMD-SHA-256

## Parameters

**`unsigned int message_part`**
The message chaining state. This parameter must be one of the following values:

**SHA_MSG_PART_ONLY**
A single hash operation

**SHA_MSG_PART_FIRST**
The first part

**SHA_MSG_PART_MIDDLE**
The middle part

**SHA_MSG_PART_FINAL**
The last part

**`unsigned int input_length`**
Length in bytes of the input data to be hashed using the SHA-224 algorithm.

**`unsigned char *input_data`**
Pointer to the input data to be hashed. This pointer must not be zero. So even in case of zero size message data, it must be set to a valid value.

**`sha256_context_t *sha256_context`**
Pointer to the SHA-256 context structure used to store intermediate values needed when chaining is used. The contents are ignored for message part SHA_MSG_PART_ONLY and SHA_MSG_PART_FIRST. This structure must contain the returned value of the preceding call to **ica_sha224** for message part SHA_MSG_PART_MIDDLE and SHA_MSG_PART_FINAL. For message part SHA_MSG_PART_FIRST and SHA_MSG_PART_FINAL, the returned value can be used for a chained call of **ica_sha224**. Therefore, the application must not modify the contents of this structure in between chained calls.

**Note:** Due to the algorithm used by SHA-224, a SHA-256 context must be used.

**`unsigned char *output_data`**
: Pointer to the buffer to contain the resulting hash data. The resulting output data has a length of **SHA224_HASH_LENGTH**. Make sure that the buffer is at least this size.

### Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

## ica_sha256

### Purpose

Performs a secure hash on the input data using the SHA-256 algorithm.

### Format

```
unsigned int ica_sha256(unsigned int message_part,
  unsigned int input_length,
  unsigned char *input_data,
  sha256_context_t *sha256_context,
  unsigned char *output_data);
```

### Required hardware support

KIMD-SHA-256, or KLMD-SHA-256

### Parameters

**`unsigned int message_part`**
: The message chaining state. This parameter must be one of the following values:

  **SHA_MSG_PART_ONLY**
  : A single hash operation

  **SHA_MSG_PART_FIRST**
  : The first part

  **SHA_MSG_PART_MIDDLE**
  : The middle part

  **SHA_MSG_PART_FINAL**
  : The last part

**`unsigned int input_length`**
: Length in bytes of the input data to be hashed using the SHA-256 algorithm.

**`unsigned char *input_data`**
: | Pointer to the input data to be hashed. This pointer must not be zero. So even | in case of zero size message data, it must be set to a valid value.

**`sha256_context_t *sha256_context`**
: Pointer to the SHA-256 context structure used to store intermediate values needed when chaining is used. The contents are ignored for message part SHA_MSG_PART_ONLY and SHA_MSG_PART_FIRST. This structure must contain the returned value of the preceding call to **ica_sha256** for message part SHA_MSG_PART_MIDDLE and SHA_MSG_PART_FINAL. For message part SHA_MSG_PART_FIRST and SHA_MSG_PART_FINAL, the returned value can be used for a chained call of **ica_sha256**. Therefore, the application must not modify the contents of this structure in between chained calls.

`unsigned char *output_data`
: Pointer to the buffer to contain the resulting hash data. The resulting output data has a length of **SHA256_HASH_LENGTH**. Make sure that the buffer is at least this size.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_sha384

### Purpose

Performs a secure hash on the input data using the SHA-384 algorithm.

### Format

```
unsigned int ica_sha384(unsigned int message_part,
  uint64_t input_length,
  unsigned char *input_data,
  sha512_context_t *sha512_context,
  unsigned char *output_data);
```

### Required hardware support

KIMD-SHA-512, or KLMD-SHA-512

### Parameters

**`unsigned int message_part`**
The message chaining state. This parameter must be one of the following values:
**SHA_MSG_PART_ONLY**
A single hash operation
**SHA_MSG_PART_FIRST**
The first part
**SHA_MSG_PART_MIDDLE**
The middle part
**SHA_MSG_PART_FINAL**
The last part

**`uint64_t input_length`**
Length in bytes of the input data to be hashed using the SHA-384 algorithm.

**`unsigned char *input_data`**
| Pointer to the input data to be hashed. This pointer must not be zero. So even
| in case of zero size message data, it must be set to a valid value.

**`sha512_context_t *sha512_context`**
Pointer to the SHA-512 context structure used to store intermediate values needed when chaining is used. The contents are ignored for message part SHA_MSG_PART_ONLY and SHA_MSG_PART_FIRST. This structure must contain the returned value of the preceding call to **ica_sha384** for message part SHA_MSG_PART_MIDDLE and SHA_MSG_PART_FINAL. For message part SHA_MSG_PART_FIRST and SHA_MSG_PART_FINAL, the returned value can be used for a chained call of **ica_sha384**. Therefore, the application must not modify the contents of this structure in between chained calls.

**Note:** Due to the algorithm used by SHA-384, a SHA-512 context must be used.

**`unsigned char *output_data`**
Pointer to the buffer to contain the resulting hash data. The resulting output data has a length of **SHA384_HASH_LENGTH**. Make sure that the buffer is at least this size.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_sha512

### Purpose

Performs a secure hash operation on input data using the SHA-512 algorithm.

### Format

```
unsigned int ica_sha512(unsigned int message_part,
  uint64_t input_length,
  unsigned char *input_data,
  sha512_context_t *sha512_context,
  unsigned char *output_data);
```

### Required hardware support

KIMD-SHA-512, or KLMD-SHA-512

### Parameters

**`unsigned int message_part`**
The message chaining state. This parameter must be one of the following values:

**SHA_MSG_PART_ONLY**
A single hash operation

**SHA_MSG_PART_FIRST**
The first part

**SHA_MSG_PART_MIDDLE**
The middle part

**SHA_MSG_PART_FINAL**
The last part

**`uint64_t input_length`**
Length in bytes of the input data to be hashed using the SHA-512 algorithm.

**`unsigned char *input_data`**
| Pointer to the input data to be hashed. This pointer must not be zero. So even
| in case of zero size message data, it must be set to a valid value.

**`sha512_context_t *sha512_context`**
Pointer to the SHA-512 context structure used to store intermediate values needed when chaining is used. The contents are ignored for message part SHA_MSG_PART_ONLY and SHA_MSG_PART_FIRST. This structure must contain the returned value of the preceding call to **ica_sha512** for message part SHA_MSG_PART_MIDDLE and SHA_MSG_PART_FINAL. For message part SHA_MSG_PART_FIRST and SHA_MSG_PART_FINAL, the returned value can

be used for a chained call of **ica_sha512**. Therefore, the application must not modify the contents of this structure in between chained calls.

**`unsigned char *output_data`**
Pointer to the buffer to contain the resulting hash data. The resulting output data has a length of **SHA512_HASH_LENGTH**. Make sure that the buffer is at least this size.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## Pseudo random number generation function

This function is included in: `include/ica_api.h`.

This function generates pseudo random data. Parameter **ouput_data* is a pointer to a buffer of byte length *output_length*. *output_length* number of bytes of pseudo random data is placed in the buffer pointed to by *output_data*.

libica initialization tries to seed the CPACF random generator. To get the seed, device `/dev/hwrng` is opened. Device `/dev/hwrng` provides true random data from crypto adapters over the crypto device driver (main module name is **`ap`**, with an alias name **`z90crypt`**, which is linking to **`ap`**). If that fails, the initialization mechanism uses device `/dev/urandom`. Within the initialization, a byte counter *s390_byte_count* is set to 0. If the CPACF pseudo random generator is available, after 4096 bytes of the pseudo random number are generated, the random number generator is seeded again. If the CPACF pseudo random generator is not available, random numbers are read from `/dev/urandom`.

### ica_random_number_generate

#### Purpose

Generates a pseudo random number.

#### Format

```
unsigned int ica_random_number_generate(unsigned int output_length,
  unsigned char *output_data);
```

#### Required hardware support

KMC-PRNG

#### Parameters

**`unsigned int output_length`**
Length in bytes of the *output_data* buffer, and the length of the generated pseudo random number.

**`unsigned char *output_data`**
Pointer to the buffer to receive the generated pseudo random number.

#### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# RSA key generation functions

These functions are included in: `include/ica_api.h`.

These functions generate an RSA public/private key pair. These functions are performed using software through OpenSSL. Hardware is not used.

## ica_rsa_key_generate_mod_expo

### Purpose

Generates RSA keys in modulus/exponent format.

### Comments

For specific information about some of these parameters, see the considerations in "Data structures" on page 74.

### Format

```
unsigned int ica_rsa_key_generate_mod_expo(ica_adapter_handle_t adapter_handle,
  unsigned int modulus_bit_length,
  ica_rsa_key_mod_expo_t *public_key,
  ica_rsa_key_mod_expo_t *private_key);
```

### Parameters

**`ica_adapter_handle_t adapter_handle`**
Pointer to a previously opened device handle.

**`unsigned int modulus_bit_length`**
Length in bits of the modulus. This value should comply with the length of the keys (in bytes), according to this calculation:

```
key_length = (modulus_bits + 7) / 8
```

**`ica_rsa_key_mod_expo_t *public_key`**
Pointer to where the generated public key is to be placed. If the *exponent* element in the public key is not set, it is randomly generated. A poorly chosen *exponent* could result in the program looping endlessly. Common public exponents are 3 and 65537.

**`ica_rsa_key_mod_expo_t *private_key`**
Pointer to where the generated private key in modulus/exponent format is to be placed. The length of both the private and public keys should be set in bytes. This value should comply with the length of the keys (in bytes), according to this calculation:

```
key_length = (modulus_bits + 7) / 8
```

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_rsa_key_generate_crt

### Purpose

Generates RSA keys in Chinese-Remainder Theorem (CRT) format.

### Comments

For specific information about some of these parameters, see the considerations in "Data structures" on page 74.

### Format

```
unsigned int ica_rsa_key_generate_crt(ica_adapter_handle_t adapter_handle,
  unsigned int modulus_bit_length,
  ica_rsa_key_mod_expo_t *public_key,
  ica_rsa_key_crt_t *private_key);
```

### Parameters

**`ica_adapter_handle_t adapter_handle`**
: Pointer to a previously opened device handle.

**`unsigned int modulus_bit_length`**
: Length in bits of the modulus part of the key. This value should comply with the length of the keys (in bytes), according to this calculation:

```
key_length = (modulus_bits + 7) / 8
```

**`ica_rsa_key_mod_expo_t *public_key`**
: Pointer to where the generated public key is to be placed. If the *exponent* element in the public key is not set, it is randomly generated. A poorly chosen *exponent* can result in the program looping endlessly. Common public exponents are 3 and 65537.

**`ica_rsa_key_crt_t *private_key`**
: Pointer to where the generated private key in CRT format is to be placed. Length of both private and public keys should be set in bytes. This value should comply with the length of the keys (in bytes), according to this calculation

```
key_length = (modulus_bits + 7) / 8
```

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# RSA encrypt and decrypt operations

These functions are included in: `include/ica_api.h`.

These functions perform a modulus/exponent operation using an RSA key whose type is either *ica_rsa_key_mod_expo_t* or *ica_rsa_key_crt_t*.

## ica_rsa_mod_expo

### Purpose

Performs an RSA encryption or decryption operation using a key in modulus/exponent format.

### Comments

Make sure that your message is padded before using this function.

## Format

```
unsigned int ica_rsa_mod_expo(ica_adapter_handle_t adapter_handle,
  unsigned char *input_data,
  ica_rsa_key_mod_expo_t *rsa_key,
  unsigned char *output_data);
```

## Parameters

**ica_adapter_handle_t adapter_handle**
Pointer to a previously opened device handle.

**unsigned char *input_data**
Pointer to the input data to be encrypted or decrypted. This data must be in big endian format. Make sure that the input data is not longer than the bit length of the key. The byte length for the input data and the key must be the same. Right align the input data inside the data block.

**ica_rsa_key_mod_expo_t *rsa_key**
Pointer to the key to be used, in modulus/exponent format.

**unsigned char *output_data**
Pointer to the location where the output results are to be placed. This buffer has to be at least the same size as *input_data* and therefore at least the same size as the size of the modulus.

## Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_rsa_crt

## Purpose

Performs an RSA encryption or decryption operation using a key in CRT format.

## Comments

Make sure that your message is padded before using this function.

## Format

```
unsigned int ica_rsa_crt(ica_adapter_handle_t adapter_handle,
  unsigned char *input_data,
  ica_rsa_key_crt_t *rsa_key,
  unsigned char *output_data);
```

## Parameters

**ica_adapter_handle_t adapter_handle**
Pointer to a previously opened device handle.

**unsigned char *input_data**
Pointer to the input data to be encrypted or decrypted. This data must be in big endian format. Make sure that the input data is not longer than the bit length of the key. The byte length for the input data and the key must be the same. Right align the input data inside the data block.

**ica_rsa_key_crt_t *rsa_key**
Pointer to the key to be used, in CRT format.

**unsigned char *output_data**
Pointer to the location where the output results are to be placed. This buffer must be as large as the *input_data*, and as large as the length of the *modulus* specified in *rsa_key*.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## DES functions

These functions are included in: `include/ica_api.h`.

These functions perform encryption and decryption and computation or verification of message authentication codes using a DES (DEA) key. A DES key has a size of 8 bytes. Each byte of a DES key contains one parity bit, such that each 64-bit DES key contains only 56 security-relevant bits. The cipher block size for DES is 8 bytes.

To securely apply DES encryption to messages that are longer than the cipher block size, modes of operation can be used to chain multiple encryption, decryption, or authentication operations. Most modes of operation require an initialization vector as additional input. As long as the messages are encrypted or decrypted using such a mode of operation, and have a size that is a multiple of a particular block size (mostly the cipher block size), the functions encrypting or decrypting according to a mode of operation also compute an output vector. This output vector can be used as the initialization vector of a chained encryption or decryption operation in the same mode with the same block size and the same key.

When decrypting a cipher text, these values used for the decryption function must match the corresponding settings of the encryption function that transformed the plain text into the cipher text:
- The mode of operation
- The key
- The initialization vector (if applicable)
- For the **ica_des_cfb** function, the *lcfb* parameter

### ica_des_cbc

#### Purpose

Encrypt or decrypt data with a DES key using Cipher Block Chaining (CBC) mode, as described in NIST Special Publication 800-38A Chapter 6.2.

#### Format

```
unsigned int ica_des_cbc(const unsigned char *in_data,
   unsigned char *out_data,
   unsigned long data_length,
   const unsigned char *key,
   unsigned char *iv,
   unsigned int direction);
```

#### Required hardware support

KMC-DEA

## Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. This buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be a multiple of the cipher block size (a multiple of 8 bytes for DES).

**`const unsigned char *key`**
Pointer to a valid DES key of 8 bytes in length.

**`unsigned char *iv`**
Pointer to a valid initialization vector of cipher block size number of bytes (8 bytes for DES). This vector is overwritten by this function. The result value in *iv* can be used as the initialization vector for a chained **ica_des_cbc** or **ica_des_cbc_cs** call with the same key.

**`unsigned int direction`**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

# ica_des_cbc_cs

## Purpose

Encrypt or decrypt data with a DES key using Cipher Block Chaining with Ciphertext Stealing (CBC-CS) mode, as described in NIST Special Publication 800-38A, Chapter 6.2 and the Addendum to NIST Special Publication 800-38A on *Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode*.

**ica_des_cbc_cs** can be used to encrypt or decrypt the last chunk of a message consisting of multiple chunks, where all chunks except the last one are encrypted or decrypted by chained calls to **ica_des_cbc**. To do this, the resulting *iv* of the last call to **ica_des_cbc** is fed into the *iv* of the **ica_des_cbc_cs** call, provided that the chunk is greater than the cipher block size (8 bytes for DES).

## Format

```
unsigned int ica_des_cbc_cs(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned char *iv,
  unsigned int direction,
  unsigned int variant);
```

### Required hardware support

KMC-DEA

### Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as the *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. This buffer must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be greater than or equal to the cipher block size (8 bytes for DES).

**`const unsigned char *key`**
Pointer to a valid DES key of 8 bytes in length.

**`unsigned char *iv`**
Pointer to a valid initialization vector of cipher block size number of bytes. This vector is overwritten during the function. For *variant* equal to 1 or *variant* equal to 2, the result value in *iv* can be used as the initialization vector for a chained **ica_des_cbc** or **ica_des_cbc_cs** call with the same key, if *data_length* is a multiple of the cipher block size.

**`unsigned int direction`**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

**`unsigned int variant`**
- **1** Use variant CBC-CS1 of the Addendum to NIST Special Publication 800-38A to encrypt or decrypt the message: always keep last two blocks in order.
- **2** Use variant CBC-CS2 of the Addendum to NIST Special Publication 800-38A to encrypt or decrypt the message: switch order of the last two blocks if *data_length* is not a multiple of the cipher block size (a multiple of 8 bytes for DES).
- **3** Use variant CBC-CS3 of the Addendum to NIST Special Publication 800-38A to encrypt or decrypt the message: always switch order of the last two blocks.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_des_cfb

### Purpose

Encrypt or decrypt data with a DES key using Cipher Feedback (CFB) mode, as described in NIST Special Publication 800-38A Chapter 6.3.

### Format

```
unsigned int ica_des_cfb(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned char *iv,
  unsigned int lcfb,
  unsigned int direction);
```

### Required hardware support

KMF-DEA

### Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as the *data_length* parameter.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as the *data_length* parameter.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**`const unsigned char *key`**
Pointer to a valid DES key of 8 bytes in length.

**`unsigned char *iv`**
Pointer to a valid initialization vector of cipher block size bytes (8 bytes for DES). This vector is overwritten during the function. The result value in *iv* can be used as the initialization vector for a chained **ica_des_cfb** call with the same key, if *data_length* in the preceding call is a multiple of the *lcfb* parameter.

**`unsigned int lcfb`**
Length in bytes of the cipher feedback, which is a value greater than or equal to 1 and less than or equal to the cipher block size (8 bytes for DES).

**`unsigned int direction`**
**0** Use the decrypt function.
**1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_des_cmac

### Purpose

Authenticate data or verify the authenticity of data with a DES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_des_cmac** can be used to authenticate or verify the authenticity of a complete message.

### Format

```
unsigned int ica_des_cmac(const unsigned char *message,
  unsigned long message_length,
  unsigned char *mac,
  unsigned int mac_length,
  const unsigned char *key,
  unsigned int direction);
```

### Required hardware support

KMAC-DEA
PCC-Compute-Last_block-CMAC-Using-DEA

### Parameters

**`const unsigned char *message`**
: Pointer to a readable buffer of size greater than or equal to *message_length* bytes. This buffer contains a message to be authenticated or of which the authenticity is to be verified.

**`unsigned long message_length`**
: Length in bytes of the message to be authenticated or verified.

**`unsigned char *mac`**
: Pointer to a buffer of size greater than or equal to *mac_length* bytes. If *direction* is equal to 1, the buffer must be writable and a message authentication code for the message in *message* of size *mac_length* bytes is written to the buffer. If *direction* is equal to 0, the buffer must be readable and contain a message authentication code to be verified against the message in *message*.

**`unsigned int mac_length`**
: Length in bytes of the message authentication code *mac*, which is less than or equal to the cipher block size (8 bytes for DES). It is recommended to use a *mac_length* of 8.

**`const unsigned char *key`**
: Pointer to a valid DES key of 8 bytes in length.

**`unsigned int direction`**
: **0** Verify message authentication code.
: **1** Compute message authentication code for the message.

### Return codes

**0** Success

**EFAULT**
: If *direction* is equal to 0 and the verification of the message authentication code fails.

For return codes indicating exceptions, see “Return codes” on page 76.

## ica_des_cmac_intermediate

### Purpose

Authenticate data or verify the authenticity of data with a DES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_des_cmac_intermediate** and **ica_des_cmac_last** can be used when the message to be authenticated or to be verified using CMAC is supplied in multiple chunks. **ica_des_cmac_intermediate** is used to process all but the last chunk. All message chunks to be processed by **ica_des_cmac_intermediate** must have a size that is a multiple of the cipher block size (8 bytes for DES).

Note that **ica_des_cmac_intermediate** has no direction argument. This function can be used during authentication and during authenticity verification.

### Format

```
unsigned int ica_des_cmac_intermediate(const unsigned char *message,
  unsigned long message_length,
  const unsigned char *key,
  unsigned char *iv);
```

### Required hardware support

KMAC-DEA

### Parameters

**`const unsigned char *message`**
: Pointer to a readable buffer of size greater than or equal to *message_length* bytes. This buffer contains a non-final part of a message to be authenticated, or of which the authenticity is to be verified.

**`unsigned long message_length`**
: Length in bytes of the message part in *message*. This value must be a multiple of the cipher block size.

**`const unsigned char *key`**
: Pointer to a valid DES key of 8 bytes in length.

**`unsigned char *iv`**
: Pointer to a valid initialization vector of cipher block size bytes (8 bytes for DES). For the first message part, this parameter must be set to a string of zeros. For processing the *n*-th message part, this parameter must be the resulting *iv* value of the **ica_des_cmac_intermediate** function applied to the (*n-1*)-th message part. This vector is overwritten during the function. The result value in *iv* can be used as the initialization vector for a chained call to **ica_des_cmac_initermediate**, or to **ica_des_cmac_last** with the same key.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_des_cmac_last

### Purpose

Authenticate data or verify the authenticity of data with a DES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_des_cmac_last** can be used to authenticate or verify the authenticity of a complete message or of the final part of a message for which all preceding parts were processed with **ica_des_cmac_intermediate**.

### Format

```
unsigned int ica_des_cmac_last(const unsigned char *message,
  unsigned long message_length,
  unsigned char *mac,
  unsigned int mac_length,
  const unsigned char *key,
  unsigned char *iv,
  unsigned int direction);
```

### Required hardware support

KMAC-DEA
PCC-Compute-Last_block-CMAC-Using-DEA

### Parameters

**`const unsigned char *message`**
Pointer to a readable buffer of size greater than or equal to *message_length* bytes. This buffer contains a message or the final part of a message, to be either authenticated or of which the authenticity is to be verified.

**`unsigned long message_length`**
Length in bytes of the message to be authenticated or verified.

**`unsigned char *mac`**
Pointer to a buffer of size greater than or equal to *mac_length* bytes. If *direction* is equal to 1, the buffer must be writable and a message authentication code for the message in *message* of size *mac_length* bytes is written to the buffer. If *direction* is equal to 0, the buffer must be readable and contain a message authentication code that is verified against the message in *message*.

**`unsigned int mac_length`**
Length in bytes of the message authentication code *mac* that is less than or equal to the cipher block size (8 bytes for DES). It is recommended to use a *mac_length* of 8.

**`const unsigned char *key`**
Pointer to a valid DES key of 8 bytes in length.

**`unsigned char *iv`**
Pointer to a valid initialization vector of cipher block size number of bytes. If *iv* is NULL, *message* is assumed to be the complete message to be processed. Otherwise, *message* is the final part of a composite message to be processed and *iv* contains the output vector resulting from processing all previous parts with chained calls to **ica_des_cmac_intermediate** (the value returned in *iv* of the **ica_des_cmac_intermediate** call applied to the penultimate message part).

**`unsigned int direction`**
- **0** Verify message authentication code.
- **1** Compute message authentication code for the message.

### Return codes

**0** Success

**EFAULT**
If *direction* is equal to 0 and the verification of the message authentication code fails.

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_des_ctr

### Purpose

Encrypt or decrypt data with a DES key using Counter (CTR) mode, as described in NIST Special Publication 800-38A Chapter 6.5. With the counter mode, each message block of the same size as the cipher block (8 bytes for DES) is combined with a counter value of the same size during encryption and decryption.

Starting with an initial counter value to be combined with the first message block, subsequent counter values to be combined with subsequent message blocks are derived from preceding counter values by an increment function. The increment function used in **ica_des_ctr** is an arithmetic increment without carry on the *M* least significant bytes in the counter, where *M* is a parameter to **ica_des_ctr**.

## Format

```
unsigned int ica_des_ctr(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned char *ctr,
  unsigned int ctr_width,
  unsigned int direction);
```

## Required hardware support

KMCTR-DEA

## Parameters

**`const unsigned char *in_data`**
: Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
: Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
: Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**`const unsigned char *key`**
: Pointer to a valid DES key of 8 bytes in length.

**`unsigned char *ctr`**
: Pointer to a readable and writable buffer of the same size as the cipher block in bytes. *ctr* contains an initialization value for a counter function, and it is replaced by a new value. That new value can be used as the initialization value for a counter function in a chained **ica_des_ctr** call with the same key, if the *data_length* used in the preceding call is a multiple of the cipher block size.

**`unsigned int ctr_width`**
: A number *M* between 1 and the cipher block size. This value is used by the counter increment function, which increments a counter value by incrementing without carry the least significant *M* bytes of the counter value.

**`unsigned int direction`**
: **0** Use the decrypt function.
: **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_des_ctrlist

## Purpose

Encrypt or decrypt data with a DES key using Counter (CTR) mode, as described in NIST Special Publication 800-38A ,Chapter 6.5. With the counter mode, each message block of the same size as the cipher block is combined with a counter value of the same size during encryption and decryption.

The **ica_des_ctrlist** function assumes that a list *n* of precomputed counter values is provided, where *n* is the smallest integer that is less than or equal to the message size divided by the cipher block size. This function is used to optimally utilize IBM System z hardware support for non-standard counter functions.

## Format

```
unsigned int ica_des_ctrlist(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  const unsigned char *ctrlist,
  unsigned int direction);
```

## Required hardware support

KMCTR-DEA

## Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

Calls to **ica_des_ctrlist** with the same key can be chained if:
- With the possible exception of the last call in the chain the *data_length* used is a multiple of the cipher block size.
- The *ctrlist* argument of each chained call contains a list of counters that follows the counters used in the preceding call.

**`const unsigned char *key`**
Pointer to a valid DES key of 8 bytes in length.

**`const unsigned char *ctrlist`**
Pointer to a readable buffer of a size greater than or equal to *data_length*, and a multiple of the cipher block size (8 bytes for DES). *ctrlist* should contain a list of precomputed counter values, each of the same size as the cipher block.

**`unsigned int direction`**
**0** Use the decrypt function.
**1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_des_ecb

## Purpose

Encrypt or decrypt data with a DES key using Electronic Code Book (ECB) mode, as described in NIST Special Publication 800-38A Chapter 6.1.

## Format

```
unsigned int ica_des_ecb(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int direction);
```

## Required hardware support

KM-DEA

## Parameters

**const unsigned char *in_data**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**unsigned char *out_data**
Pointer to a writeable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**unsigned long data_length**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be a multiple of the cipher block size (8 bytes for DES).

**const unsigned char *key**
Pointer to a valid DES key of 8 bytes in length.

**unsigned int direction**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_des_ofb

## Purpose

Encrypt or decrypt data with a DES key using Output Feedback (OFB) mode, as described in NIST Special Publication 800-38A Chapter 6.4.

## Format

```
unsigned int ica_des_ofb(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
```

```
const unsigned char *key,
unsigned int key_length,
unsigned char *iv,
unsigned int direction);
```

### Required hardware support

KMO-DEA

### Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer that contains the resulting encrypted or decrypted message. The size of this buffer must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**`const unsigned char *key`**
Pointer to a valid DES key of 8 bytes in length.

**`unsigned char *iv`**
Pointer to a valid initialization vector of the same size as the cipher block in bytes (8 bytes for DES). This vector is overwritten during the function. If *data_length* is a multiple of the cipher block size (8 bytes for DES), the result value in *iv* can be used as the initialization vector for a chained **ica_des_ofb** call with the same key.

**`unsigned int direction`**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## Compatibility with earlier versions

In order to stay compatible with earlier versions of libica, the following DES interfaces remain supported:

```
unsigned int ica_des_encrypt(unsigned int mode,
 unsigned int data_length, unsigned char *input_data,
 ica_des_vector_t *iv, ica_des_key_single_t *des_key,
 unsigned char *output_data);
```

```
unsigned int ica_des_decrypt(unsigned int mode,
 unsigned int data_length, unsigned char *input_data,
 ica_des_vector_t *iv, ica_des_key_single_t *des_key,
 unsigned char *output_data);
```

Table 2 on page 31 shows libica version 2.0 DES functions calls, and their corresponding libica version 2.4 DES function calls.

*Table 2. Compatibility of libica version 2.0 DES functions calls to libica version 2.4 DES function calls*

| Calling this libica version 2.0 DES function | Corresponds to calling this libica version 2.4 DES function |
|---|---|
| `ica_des_encrypt(MODE_ECB, data_length,in_data,NULL, key, out_data);` | `ica_des_ecb(in_data,out_data,(long)data_length, key,1);` |
| `ica_des_encrypt(MODE_CBC,data_length,in_data,iv, key,out_data);` | `ica_des_cbc(in_data,out_data,(long)data_length, key,iv,1);` |
| `ica_des_decrypt(MODE_ECB,data_length,in_data,NULL, key,out_data);` | `ica_des_ecb(in_data,out_data,(long)data_length, key,0);` |
| `ica_des_decrypt(MODE_CBC,data_length,in_data,iv, key,out_data);` | `ica_des_cbc(in_data,out_data,(long)data_length, key,iv,0);` |

The functions **ica_des_encrypt** and **ica_des_decrypt** remain supported, but their use is discouraged in favor of **ica_des_ecb** and **ica_des_cbc**.

For a detailed description of the earlier APIs, see *libica Programmers Reference* version 2.0.

## TDES/3DES functions

These functions are included in: `include/ica_api.h`.

These functions perform encryption and decryption or computation and verification of message authentication codes using a triple-DES (3DES, TDES or TDEA) key. A 3DES key consists of a concatenation of three DES keys, each of which has a size of 8 bytes. Note that each byte of a DES key contains one parity bit, such that each 64-bit DES key contains only 56 security-relevant bits. The cipher block size for 3DES is 8 bytes.

3DES is known in two variants: a two key variant and a three key variant. This library implements only the three key variant. The two key variant can be derived from functions for the three key variant by using the same key as the first and third key.

To securely apply 3DES encryption to messages that are longer than the cipher block size, modes of operation can be used to chain multiple encryption, decryption, or authentication operations. Most modes of operation require an initialization vector as additional input. As long as the messages are encrypted or decrypted using such a mode of operation and have a size that is a multiple of a particular block size (mostly the cipher block size), the functions encrypting or decryption according to that mode of operation also compute an output vector that can be used as the initialization vector of a chained encryption or decryption operation in the same mode with the same block size and the same key.

Note that when decrypting a cipher text, the mode of operation, the key, the initialization vector (if applicable), and for **ica_3des_cfb** the *lcfb* value used for the decryption function must match the corresponding settings of the encryption function that was used to transform the plain text into the cipher text.

# ica_3des_cbc

## Purpose

Encrypt or decrypt data with an 3DES key using Cipher Block Chaining (CBC) mode, as described in NIST Special Publication 800-38A Chapter 6.2.

## Format

```
unsigned int ica_3des_cbc(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned char *iv,
  unsigned int direction);
```

## Required hardware support

KMC-TDEA-192

## Parameters

**`const unsigned char *in_data`**
: Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
: Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
: Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be a multiple of the cipher block size (8 bytes for 3DES).

**`const unsigned char *key`**
: Pointer to a valid 3DES key of 24 bytes in length.

**`unsigned char *iv`**
: Pointer to a valid initialization vector of cipher block size number of bytes. This vector is overwritten during the function. The result value in *iv* can be used as the initialization vector for a chained **ica_3des_cbc** or **ica_3des_cbc_cs** call with the same key.

**`unsigned int direction`**
: **0** Use the decrypt function.
: **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_3des_cbc_cs

## Purpose

Encrypt or decrypt data with a 3DES key using Cipher Block Chaining with Ciphertext Stealing (CBC-CS) mode, as described in NIST Special Publication

800-38A Chapter 6.2 and the Addendum to NIST Special Publication 800-38A on *Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode*.

**ica_3des_cbc_cs** can be used to encrypt or decrypt the last chunk of a message consisting of multiple chunks, where all chinks except the last one are encrypted or decrypted by chained calls to **ica_3des_cbc**. To do this, the resulting *iv* of the last call to **ica_3des_cbc** is fed into the *iv* of the **ica_3des_cbc_cs** call, provided that the chunk is greater than the cipher block size (8 bytes for 3DES).

## Format

```
unsigned int ica_3des_cbc_cs(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned char *iv,
  unsigned int direction,
  unsigned int variant);
```

## Required hardware support

KMC-TDEA-192

## Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be greater than or equal to the cipher block size (8 bytes for 3DES).

**`const unsigned char *key`**
Pointer to a valid 3DES key of 24 bytes in length.

**`unsigned char *iv`**
Pointer to a valid initialization vector of the same size as the cipher block in bytes. This vector is overwritten during the function. For *variant* equal to 1 or *variant* equal to 2, the result value in *iv* can be used as the initialization vector for a chained **ica_3des_cbc** or **ica_3des_cbc_cs** call with the same key, if *data_length* is a multiple of the cipher block size.

**`unsigned int direction`**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

**`unsigned int variant`**
- **1** Use variant CBC-CS1 of the Addendum to NIST Special Publication 800-38A to encrypt or decrypt the message: always keep last two blocks in order.
- **2** Use variant CBC-CS2 of the Addendum to NIST Special Publication

800-38A to encrypt or decrypt the message: switch order of the last two blocks if *data_length* is not a multiple of the cipher block size (a multiple of 8 bytes for 3DES).

**3** Use variant CBC-CS3 of the Addendum to NIST Special Publication 800-38A to encrypt or decrypt the message: always switch order of the last two blocks.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_3des_cfb

### Purpose

Encrypt or decrypt data with a 3DES key using Cipher Feedback (CFB) mode, as described in NIST Special Publication 800-38A Chapter 6.3.

### Format

```
unsigned int ica_3des_cfb(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned char *iv,
  unsigned int lcfb,
  unsigned int direction);
```

### Required hardware support

KMF-TDEA-192

### Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**`const unsigned char *key`**
Pointer to a valid 3DES key of 24 bytes in length.

**`unsigned char *iv`**
Pointer to a valid initialization vector of cipher block size number of bytes (8 bytes for 3DES). This vector is overwritten during the function. The result value in *iv* can be used as the initialization vector for a chained **ica_3des_cfb** call with the same key, if the *data_length* in the preceding call is a multiple of *lcfb*.

**`unsigned int lcfb`**
Length in bytes of the cipher feedback, which is a value greater than or equal to 1 and less than or equal to the cipher block size (8 bytes for 3DES).

**`unsigned int direction`**

**0** Use the decrypt function.

**1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_3des_cmac

### Purpose

Authenticate data or verify the authenticity of data with an 3DES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_3des_cmac** can be used to authenticate or verify the authenticity of a complete message.

### Format

```
unsigned int ica_3des_cmac(const unsigned char *message,
  unsigned long message_length,
  unsigned char *mac,
  unsigned int mac_length,
  const unsigned char *key,
  unsigned int direction);
```

### Required hardware support

KMAC-TDEA-192
PCC-Compute-Last_block-CMAC-Using-TDEA-192

### Parameters

**`const unsigned char *message`**
Pointer to a readable buffer of size greater than or equal to *message_length* bytes. This buffer contains a message to be authenticated, or of which the authenticity is to be verified.

**`unsigned long message_length`**
Length in bytes of the message to be authenticated or verified.

**`unsigned char *mac`**
Pointer to a buffer of size greater than or equal to *mac_length* bytes. If *direction* is equal to 1, the buffer must be writable and a message authentication code for the message in *message* of size *mac_length* bytes is written to the buffer. If *direction* is equal to 0, the buffer must be readable and contain a message authentication code to be verified against the message in *message*.

**`unsigned int mac_length`**
Length in bytes of the message authentication code *mac*, which is less than or equal to the cipher block size (8 bytes for 3DES). It is recommended to use a *mac_length* of 8.

**`const unsigned char *key`**
Pointer to a valid 3DES key of 24 bytes in length.

**`unsigned int direction`**

**0** Verify message authentication code.

**1** Compute message authentication code for the message.

### Return codes

**0** Success

**EFAULT**
: If *direction* is equal to 0 and the verification of the message authentication code fails.

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_3des_cmac_intermediate

### Purpose

Authenticate data or verify the authenticity of data with an 3DES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_3des_cmac_intermediate** and **ica_3des_cmac_last** can be used when the message to be authenticated or to be verified using CMAC is supplied in multiple chunks. **ica_3des_cmac_intermediate** is used to process all but the last chunk. All message chunks to be processed by **ica_3des_cmac_intermediate** must have a size that is a multiple of the cipher block size (a multiple of 8 bytes for 3DES).

Note that **ica_3des_cmac_intermediate** has no direction argument. This function can be used during authentication and during authenticity verification.

### Format

```
unsigned int ica_3des_cmac_intermediate(const unsigned char *message,
  unsigned long message_length,
  const unsigned char *key,
  unsigned char *iv);
```

### Required hardware support

KMAC-TDEA-192

### Parameters

**`const unsigned char *message`**
: Pointer to a readable buffer of size greater than or equal to *message_length* bytes. This buffer contains a non-final part of a message to be authenticated, or of which the authenticity is to be verified.

**`unsigned long message_length`**
: Length in bytes of the message part in *message*. This value must be a multiple of the cipher block size.

**`const unsigned char *key`**
: Pointer to a valid 3DES key of 24 bytes in length.

**`unsigned char *iv`**
: Pointer to a valid initialization vector of size cipher block size (8 bytes for 3DES). For the first message part, this parameter must be set to a string of zeros. For processing the *n*-th message part, this parameter must be the resulting *iv* value of the **ica_3des_cmac_intermediate** applied to the (*n-1*)-th message part. This vector is overwritten during the function. The result value in *iv* can be used as the initialization vector for a chained call to **ica_3des_cmac_initermediate** or to **ica_3des_cmac_last** with the same key.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_3des_cmac_last

## Purpose

Authenticate data or verify the authenticity of data with an 3DES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_3des_cmac_last** can be used to authenticate or verify the authenticity of a complete message or of the final part of a message, for which all preceding parts were processed with **ica_3des_cmac_intermediate**.

## Format

```
unsigned int ica_3des_cmac_last(const unsigned char *message,
  unsigned long message_length,
  unsigned char *mac,
  unsigned int mac_length,
  const unsigned char *key,
  unsigned char *iv,
  unsigned int direction);
```

## Required hardware support

KMAC-TDEA,-192
PCC-Compute-Last_block-CMAC-Using-TDEA-192

## Parameters

**`const unsigned char *message`**
Pointer to a readable buffer of size greater than or equal to *message_length* bytes. It contains a message or the final part of a message to be authenticated, or of which the authenticity is to be verified.

**`unsigned long message_length`**
Length in bytes of the message to be authenticated or verified.

**`unsigned char *mac`**
Pointer to a buffer of size greater than or equal to *mac_length* bytes. If *direction* is equal to 1, the buffer must be writable and a message authentication code for the message in *message* of size *mac_length* bytes is written to the buffer. If *direction* is equal to 0, the buffer must be readable and contain a message authentication code that is to be verified against the message in *message*.

**`unsigned int mac_length`**
Length in bytes of the message authentication code *mac* in bytes that is less than or equal to the cipher block size (8 bytes for 3DES). It is recommended to use a *mac_length* of 8.

**`const unsigned char *key`**
Pointer to a valid 3DES key of 24 bytes in length.

**`unsigned char *iv`**
Pointer to a valid initialization vector of cipher block size number of bytes. If *iv* is NULL, *message* is assumed to be the complete message to be processed. Otherwise, *message* is the final part of a composite message to be processed and *iv* contains the output vector resulting from processing all previous parts with chained calls to **ica_des_cmac_intermediate** (the value returned in *iv* of the **ica_des_cmac_intermediate** call applied to the penultimate message part.

**`unsigned int direction`**

**0** Verify message authentication code.
**1** Compute message authentication code for the message.

## Return codes

**0** Success

**EFAULT**
: If *direction* is equal to 0 and the verification of the message authentication code fails.

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_3des_ctr

## Purpose

Encrypt or decrypt data with a triple-length DES key using Counter (CTR) mode, as described in NIST Special Publication 800-38A Chapter 6.5. With the counter mode, each message block of size cipher block size (8 bytes for 3DES) is combined with a counter value of the same size during encryption and decryption.

Starting with an initial counter value to be combined with the first message block, subsequent counter values to be combined with subsequent message blocks are derived from preceding counter values by an increment function. The increment function used in **ica_3des_ctr** is an arithmetic increment without carry on the *M* least significant bytes in the counter, where *M* is a parameter to **ica_3des_ctr**.

## Format

```
unsigned int ica_3des_ctr(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned char *ctr,
  unsigned int ctr_width,
  unsigned int direction);
```

## Required hardware support

KMCTR-TDEA-192

## Parameters

**`const unsigned char *in_data`**
: Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
: Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
: Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**`const unsigned char *key`**
: Pointer to a valid 3DES key of 24 bytes in length.

**`unsigned char *ctr`**
: Pointer to a readable and writable buffer of the same size as the cipher block in bytes. *ctr* contains an initialization value for a counter function that is replaced by a new value. The new value can be used as an initialization value for a counter function in a chained **ica_3des_ctr** call with the same key, if the *data_length* used in the preceding call is a multiple of the cipher block size.

**`unsigned int ctr_width`**
: A number *M* between 1 and the cipher block size. The value is used by the counter increment function, which increments a counter value by incrementing without carry the least significant *M* bytes of the counter value.

**`unsigned int direction`**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_3des_ctrlist

### Purpose

Encrypt or decrypt data with an 3DES key using Counter (CTR) mode, as described in NIST Special Publication 800-38A ,Chapter 6.5. With the counter mode, each message block of the same size as the cipher block is combined with a counter value of the same size during encryption and decryption.

The **ica_3des_ctrlist** function assumes that a list *n* of precomputed counter values is provided where *n* is the smallest integer that is less than or equal to the message size divided by the cipher block size. This function is used to optimally utilize IBM System z hardware support for non-standard counter functions.

### Format

```
unsigned int ica_3des_ctrlist(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  const unsigned char *ctrlist,
  unsigned int direction);
```

### Required hardware support

KMCTR-TDEA-192

### Parameters

**`const unsigned char *in_data`**
: Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
: Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
: Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

Calls to **ica_3des_ctrlist** with the same key can be chained if:
- With the possible exception of the last call in the chain the *data_length* used is a multiple of the cipher block size.
- The *ctrlist* argument of each chained call contains a list of counters that follows the counters used in the preceding call.

**const unsigned char *key**
Pointer to a valid 3DES key of 24 bytes in length.

**const unsigned char *ctrlist**
Pointer to a readable buffer that is both of size greater than or equal to *data_length*, and a multiple of the cipher block size (8 bytes for 3DES). *ctrlist* should contain a list of precomputed counter values, each of the same size as the cipher block.

**unsigned int direction**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

## ica_3des_ecb

### Purpose

Encrypt or decrypt data with an 3DES key using Electronic Code Book (ECB) mode, as described in NIST Special Publication 800-38A Chapter 6.1.

### Format

```
unsigned int ica_3des_ecb(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int direction);
```

### Required hardware support

KM-DEA-192

### Parameters

**const unsigned char *in_data**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**unsigned char *out_data**
Pointer to a writeable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**unsigned long data_length**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be a multiple of the cipher block size (8 bytes for 3DES).

**const unsigned char *key**
Pointer to a valid 3DES key of 24 bytes in length.

**unsigned int direction**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

# ica_3des_ofb

## Purpose

Encrypt or decrypt data with an 3DES key using Output Feedback (OFB) mode, as described in NIST Special Publication 800-38A Chapter 6.4.

## Format

```
unsigned int ica_3des_ofb(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned char *iv,
  unsigned int direction);
```

## Required hardware support

KMO-TDEA-192

## Parameters

**`const unsigned char *in_data`**
: Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
: Pointer to a writable buffer that contains the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
: Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**`const unsigned char *key`**
: Pointer to a valid 3DES key of 24 bytes in length.

**`unsigned char *iv`**
: Pointer to a valid initialization vector of the same size as the cipher block in bytes (8 bytes for 3DES). This vector is overwritten during the function. If *data_length* is a multiple of the cipher block size (a multiple of 8 for 3DES), the result value in *iv* can be used as the initialization vector for a chained **ica_3des_ofb** call with the same key.

**`unsigned int direction`**
: **0** Use the decrypt function.
: **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

## Compatibility with earlier versions

In order to stay compatible with earlier versions of libica, the following 3DES interfaces remain supported:

```
unsigned int ica_3des_encrypt(unsigned int mode,
 unsigned int data_length, unsigned char *input_data,
 ica_des_vector_t *iv, ica_des_key_triple_t *des_key,
 unsigned char *output_data);
```

```
unsigned int ica_3des_decrypt(unsigned int mode,
 unsigned int data_length, unsigned char *input_data,
 ica_des_vector_t *iv, ica_des_key_triple_t *des_key,
 unsigned char *output_data);
```

Table 3 shows libica version 2.0 TDES functions calls, and their corresponding libica version 2.4 TDES function calls.

*Table 3. Compatibility of libica version 2.0 TDES functions calls to libica version 2.4 TDES function calls*

| Calling this libica version 2.0 TDES function | Corresponds to calling this libica version 2.4 TDES function |
|---|---|
| `ica_3des_encrypt(MODE_ECB, data_length,in_data,NULL, key, out_data);` | `ica_3des_ecb(in_data,out_data,(long)data_length, key,1);` |
| `ica_3des_encrypt(MODE_CBC,data_length,in_data,iv, key,out_data);` | `ica_3des_cbc(in_data,out_data,(long)data_length, key,iv,1);` |
| `ica_3des_decrypt(MODE_ECB,data_length,in_data,NULL, key,out_data);` | `ica_3des_ecb(in_data,out_data,(long)data_length, key,0);` |
| `ica_3des_decrypt(MODE_CBC,data_length,in_data,iv, key,out_data);` | `ica_3des_cbc(in_data,out_data,(long)data_length,, key,iv,0);` |

The functions **ica_3des_encrypt** and **ica_3des_decrypt** remain supported, but their use is discouraged in favor of **ica_3des_ecb** and **ica_3des_cbc**.

For a detailed description of the earlier APIs, see *libica Programmers Reference* version 2.0.

# AES functions

These functions are included in: `include/ica_api.h`.

These functions perform encryption and decryption or computation or verification of message authentication codes using an AES key. Supported key lengths are 16, 24 or 32 bytes for AES-128, AES-192 and AES-256 respectively. The cipher block size for AES is 16 bytes.

To securely apply AES encryption to messages that are longer than the cipher block size, modes of operation can be used to chain multiple encryption, decryption, or authentication operations. Most modes of operation require an initialization vector as additional input.

As long as the messages are encrypted or decrypted using such a mode of operation, have a size that is a multiple of a particular block size (mostly the cipher block size), the functions encrypting or decryption according to a mode of operation also compute an output vector. The output vector can be used as the initialization vector of a chained encryption or decryption operation in the same mode with the same block size and the same key.

Note that when decrypting a cipher text the mode of operation, the key, the initialization vector (if applicable), and for **ica_aes_cfb** the *lcfb* value used for the decryption function must match the corresponding settings of the encryption function that transformed the plain text into the cipher text.

# ica_aes_cbc

## Purpose

Encrypt or decrypt data with an AES key using Cipher Block Chaining (CBC) mode, as described in NIST Special Publication 800-38A Chapter 6.2.

## Format

```
unsigned int ica_aes_cbc(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned char *iv,
  unsigned int direction);
```

## Required hardware support

KMC-AES-128, KMC-AES-192, or KMC-AES-256

## Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be a multiple of the cipher block size (a multiple of 16 for AES).

**`const unsigned char *key`**
Pointer to a valid AES key.

**`unsigned int key_length`**
Length in bytes of the AES key. Supported sizes are 16, 24, and 32, for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned char *iv`**
Pointer to a valid initialization vector of the same size as the cipher block in bytes. This vector is overwritten during the function. The result value in *iv* can be used as the initialization vector for a chained **ica_aes_cbc** or **ica_aes_cbc_cs** call with the same key.

**`unsigned int direction`**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

# ica_aes_cbc_cs

## Purpose

Encrypt or decrypt data with an AES key using Cipher Block Chaining with Ciphertext Stealing (CBC-CS) mode, as described in NIST Special Publication 800-38A Chapter 6.2, and the Addendum to NIST Special Publication 800-38A on *Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode*.

**ica_aes_cbc_cs** can be used to encrypt or decrypt the last chunk of a message consisting of multiple chunks, where all chunks except the last one are encrypted or decrypted by chained calls to **ica_aes_cbc**. To do this, the resulting *iv* of the last call to **ica_aes_cbc** is fed into the *iv* of the **ica_aes_cbc_cs** call, provided that the chunk is greater than the cipher block size (greater than 16 bytes for AES).

## Format

```
unsigned int ica_aes_cbc_cs(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned char *iv,
  unsigned int direction,
  unsigned int variant);
```

## Required hardware support

KMC-AES-128, KMC-AES-192 or KMC-AES-256

## Parameters

**`const unsigned char *in_data`**
: Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
: Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
: Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be greater than or equal to the cipher block size (16 bytes for AES).

**`const unsigned char *key`**
: Pointer to a valid AES key.

**`unsigned int key_length`**
: Length in bytes of the AES key. Supported sizes are 16, 24, and 32, for AES-128, AES-192, and AES-256 respectively. . Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned char *iv`**
: Pointer to a valid initialization vector of cipher block size number of bytes. This vector is overwritten during the function. For *variant* equal to 1 or *variant*

equal to 2, the result value in *iv* can be used as the initialization vector for a chained **ica_aes_cbc** or **ica_aes_cbc_cs** call with the same key, if *data_length* is a multiple of the cipher block size.

**unsigned int direction**

**0** Use the decrypt function.

**1** Use the encrypt function.

**unsigned int variant**

**1** Use variant CBC-CS1 of the Addendum to NIST Special Publication 800-38A to encrypt or decrypt the message: always keep last two blocks in order.

**2** Use variant CBC-CS2 of the Addendum to NIST Special Publication 800-38A to encrypt or decrypt the message: switch order of the last two blocks if *data_length* is not a multiple of the cipher block size (a multiple of 16 bytes for AES).

**3** Use variant CBC-CS3 of the Addendum to NIST Special Publication 800-38A to encrypt or decrypt the message: always switch order of the last two blocks.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_aes_ccm

### Purpose

Encrypt and authenticate or decrypt data and check authenticity of data with an AES key using Counter with Cipher Block Chaining Message Authentication Code (CCM) mode, as described in NIST Special Publication 800-38C. Formatting and counter functions are implemented according to NIST 800-38C Appendix A.

### Format

```
unsigned int ica_aes_ccm(unsigned char *payload,
  unsigned long payload_length,
  unsigned char *ciphertext_n_mac,
  unsigned int mac_length,
  const unsigned char *assoc_data,
  unsigned long assoc_data_length,
  const unsigned char *nonce,
  unsigned int nonce_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned int direction);
```

### Required hardware support

KMCTR-AES-128, KMCTR-AES-192, or KMCTR-AES-256
KMAC-AES-128, KMAC-AES-192, or KMAC-AES-256

### Parameters

**unsigned char *payload**

Pointer to a buffer of size greater than or equal to *payload_length* bytes. If *direction* is equal to 1, the payload buffer must be readable and contain a payload message of size *payload_length* to be encrypted. If direction is equal to 0, the payload buffer must be writable. If the authentication verification succeeds, the decrypted message in the most significant *payload_length* bytes of *ciphertext_n_mac* is written to this buffer. Otherwise, the contents of this buffer is undefined.

**`unsigned long payload_length`**
Length in bytes of the message to be encrypted or decrypted. This value can be 0 unless *assoc_data_length* is equal to 0.

**`unsigned char *ciphertext_n_mac`**
Pointer to a buffer of size greater than or equal to *payload_length* plus *mac_length* bytes. If *direction* is equal to 1, the buffer must be writable and the encrypted message from *payload* followed by the message authentication code for the nonce, the payload, and associated data are written to that buffer. If direction is equal to 0, then the buffer is readable and contains an encrypted message of length *payload_length* followed by a message authentication code of length *mac_length*.

**`unsigned int mac_length`**
Length in bytes of the message authentication code. Valid values are: 4, 6, 8, 10, 12, and 16.

**`const unsigned char *assoc_data`**
Pointer to a readable buffer of size greater than or equal to *assoc_data_length* bytes. The associated data in the most significant *assoc_data_length* bytes is subject to the authentication code computation, but is not encrypted.

**`unsigned long assoc_data_length`**
Length of the associated data in *assoc_data*. This value can be 0 unless *payload_length* is equal to 0.

**`const unsigned char *nonce`**
Pointer to readable buffer of size greater than or equal to *nonce_length* bytes, which contains a nonce (number used once) of size *nonce_length* bytes.

**`unsigned int nonce_length`**
Length of the *nonce* in bytes. Valid values are greater than 6 and less than 14.

**`const unsigned char *key`**
Specifies a pointer to a valid AES key.

**`unsigned int key_length`**
Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128, AES-192 and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned int direction`**
**0** Use the decrypt function.
**1** Use the encrypt function.

## Return codes

**0** Success

**EFAULT**
If *direction* is equal to 0 and the verification of the message authentication code fails.

For return codes indicating exceptions, see “Return codes” on page 76.

# ica_aes_cfb

## Purpose

Encrypt or decrypt data with an AES key using Cipher Feedback (CFB) mode, as described in NIST Special Publication 800-38A Chapter 6.3.

## Format

```
unsigned int ica_aes_cfb(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned char *iv,
  unsigned int lcfb,
  unsigned int direction);
```

## Required hardware support

KMF-AES-128, KMF-AES-192, or KMF-AES-256

## Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**`const unsigned char *key`**
Pointer to a valid AES key.

**`unsigned int key_length`**
Length in bytes of the AES key. Supported sizes are 16, 24, and 32, for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned char *iv`**
Pointer to a valid initialization vector of the same size as the cipher block in bytes (16 bytes for AES). This vector is overwritten during the function. The result value in *iv* can be used as the initialization vector for a chained **ica_aes_cfb** call with the same key, if the *data_length* in the preceding call is a multiple of *lcfb*.

**`unsigned int lcfb`**
Length in bytes of the cipher feedback, which is a value greater than or equal to 1 and less than or equal to the cipher block size (16 bytes for AES).

**`unsigned int direction`**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

# ica_aes_cmac

## Purpose

Authenticate data or verify the authenticity of data with an AES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_aes_cmac** can be used to authenticate or verify the authenticity of a complete message.

## Format

```
unsigned int ica_aes_cmac(const unsigned char *message,
  unsigned long message_length,
  unsigned char *mac,
  unsigned int mac_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned int direction);
```

## Required hardware support

KMAC-AES-128, KMAC-AES-192 or KMAC-AES-256
PCC-Compute-Last_block-CMAC-Using-AES-128, PCC-Compute-Last_block-CMAC-Using-AES-192, or PCC-Compute-Last_block-CMAC-Using-AES-256

## Parameters

**`const unsigned char *message`**
Pointer to a readable buffer of size greater than or equal to *message_length* bytes. This buffer contains a message to be authenticated, or of which the authenticity is to be verified.

**`unsigned long message_length`**
Length in bytes of the message to be authenticated or verified.

**`unsigned char *mac`**
Pointer to a buffer of size greater than or equal to *mac_length* bytes. If *direction* is equal to 1, the buffer must be writable and a message authentication code for the message in message of size *mac_length* bytes is written to this buffer. If direction is equal to 0, this buffer must be readable and contain a message authentication code to be verified against the message in *message*.

**`unsigned int mac_length`**
Length in bytes of the message authentication code *mac* in bytes, which is less than or equal to the cipher block size (16 bytes for AES). It is recommended to use values greater than or equal to 8.

**`const unsigned char *key`**
Pointer to a valid AES key.

**`unsigned int key_length`**
Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned int direction`**
- **0** Verify message authentication code.
- **1** Compute message authentication code for the message.

## Return codes

**0** Success

**EFAULT**
: If *direction* is equal to 0 and the verification of the message authentication code fails.

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_aes_cmac_intermediate

## Purpose

Authenticate data or verify the authenticity of data with an AES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_aes_cmac_intermediate** and **ica_aes_cmac_last** can be used when the message to be authenticated or to be verified using CMAC is supplied in multiple chunks. **ica_aes_cmac_intermediate** is used to process all but the last chunk. All message chunks to be processed by **ica_aes_cmac_intermediate** must have a size that is a multiple of the cipher block size (a multiple of 16 bytes for AES).

Note that **ica_aes_cmac_intermediate** has no direction argument. This function can be used during authentication and during authenticity verification.

## Format

```
unsigned int ica_aes_cmac_intermediate(const unsigned char *message,
  unsigned long message_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned char *iv);
```

## Required hardware support

KMAC-AES-128, KMAC-AES-192, or KMAC-AES-256

## Parameters

**`const unsigned char *message`**
: Pointer to a readable buffer of size greater than or equal to *message_length* bytes. This buffer contains a non-final part of a message, to be authenticated or of which the authenticity is to be verified.

**`unsigned long message_length`**
: Length in bytes of the message part in *message*. This value must be a multiple of the cipher block size.

**`const unsigned char *key`**
: Pointer to a valid AES key.

**`unsigned int key_length`**
: Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned char *iv`**
: Pointer to a valid initialization vector of cipher block size number of bytes (16 bytes for AES). For the first message part, this parameter must be set to a string of zeros. For processing the *n*-th message part, this parameter must be the resulting *iv* value of the **ica_aes_cmac_intermediate** function applied to the (*n-1*)-th message part. This vector is overwritten during the function. The result

value in *iv* can be used as the initialization vector for a chained call to **ica_aes_cmac_initermediate** or to **ica_aes_cmac_last** with the same key.

### Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

## ica_aes_cmac_last

### Purpose

Authenticate data or verify the authenticity of data with an AES key using the Block Cipher Based Message Authentication Code (CMAC) mode, as described in NIST Special Publication 800-38B. **ica_aes_cmac_last** can be used to authenticate or verify the authenticity of a complete message, or of the final part of a message for which all preceding parts were processed with **ica_aes_cmac_intermediate**.

### Format

```
unsigned int ica_aes_cmac_last(const unsigned char *message,
  unsigned long message_length,
  unsigned char *mac,
  unsigned int mac_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned char *iv,
  unsigned int direction);
```

### Required hardware support

KMAC-AES-128, KMAC-AES-192 or KMAC-AES-256
PCC-Compute-Last_block-CMAC-Using-AES-128, PCC-Compute-Last_block-CMAC-Using-AES-192, or PCC-Compute-Last_block-CMAC-Using-AES-256

### Parameters

**`const unsigned char *message`**
Pointer to a readable buffer of size greater than or equal to *message_length* bytes. This buffer contains a message or the final part of a message to be authenticated, or of which the authenticity is to be verified.

**`unsigned long message_length`**
Length in bytes of the message to be authenticated or verified.

**`unsigned char *mac`**
Pointer to a buffer of size greater than or equal to *mac_length* bytes. If *direction* is equal to 1, the buffer must be writable and a message authentication code for the message in *message* of size *mac_length* bytes is written to the buffer. If *direction* is equal to 0, the buffer must be readable and contain a message authentication code that is verified against the message in *message*.

**`unsigned int mac_length`**
Length in bytes of the message authentication code *mac* in bytes, which is less than or equal to the cipher block size (16 bytes for AES). It is recommended to use values greater than or equal to 8.

**`const unsigned char *key`**
Pointer to a valid AES key.

**`unsigned int key_length`**
Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128,

AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned char *iv`**
Pointer to a valid initialization vector of cipher block size number of bytes. If *iv* is NULL, *message* is assumed to be the complete message to be processed. Otherwise, *message* is the final part of a composite message to be processed, and *iv* contains the output vector resulting from processing all previous parts with chained calls to **ica_aes_cmac_intermediate** (the value returned in *iv* of the **ica_aes_cmac_intermediate** call applied to the penultimate message part).

**`unsigned int direction`**
- **0** Verify message authentication code.
- **1** Compute message authentication code for the message.

### Return codes

**0** Success

**EFAULT**
If *direction* is equal to 0 and the verification of the message authentication code fails.

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_aes_ctr

### Purpose

Encrypt or decrypt data with an AES key using Counter (CTR) mode, as described in NIST Special Publication 800-38A Chapter 6.5. With the counter mode, each message block of size cipher block size (16 bytes for AES) is combined with a counter value of the same size during encryption and decryption.

Starting with an initial counter value to be combined with the first message block, subsequent counter values to be combined with subsequent message blocks are derived from preceding counter values by an increment function. The increment function used in **ica_aes_ctr** is an arithmetic increment without carry on the *M* least significant bytes in the counter where *M* is a parameter to **ica_aes_ctr**.

### Format

```
unsigned int ica_aes_ctr(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned char *ctr,
  unsigned int ctr_width,
  unsigned int direction);
```

### Required hardware support

KMCTR-AES-128, KMCTR-AES-192, or KMCTR-AES-256

### Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**unsigned char *out_data**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**unsigned long data_length**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**const unsigned char *key**
Pointer to a valid AES key.

**unsigned int key_length**
Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**unsigned char *ctr**
Pointer to a readable and writable buffer of the same size as the cipher block in bytes. *ctr* contains an initialization value for a counter function, and it is replaced by a new value. That new value can be used as an initialization value for a counter function in a chained **ica_aes_ctr** call with the same key, if the *data_length* used in the preceding call is a multiple of the cipher block size.

**unsigned int ctr_width**
A number *M* between 1 and the cipher block size. The value is used by the counter increment function, which increments a counter value by incrementing without carry the least significant *M* bytes of the counter value.

**unsigned int direction**
- **0** Use the decrypt function.
- **1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_aes_ctrlist

### Purpose

Encrypt or decrypt data with an AES key using Counter (CTR) mode, as described in NIST Special Publication 800-38A ,Chapter 6.5. With the counter mode, each message block of the same size as the cipher block in bytes is combined with a counter value of the same size during encryption and decryption.

The **ica_aes_ctrlist** function assumes that a list *n* of precomputed counter values is provided, where *n* is the smallest integer that is less than or equal to the message size divided by the cipher block size. This function optimally uses IBM System z hardware support for non-standard counter functions.

### Format

```
unsigned int ica_aes_ctrlist(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int key_length,
  const unsigned char *ctrlist,
  unsigned int direction);
```

## Required hardware support

KMCTR-DEAKMCTR-AES-128, KMCTR-AES-192, or KMCTR-AES-256

## Parameters

**`const unsigned char *in_data`**
: Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
: Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
: Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

  Calls to **ica_aes_ctrlist** with the same key can be chained if:
  - With the possible exception of the last call in the chain the *data_length* used is a multiple of the cipher block size.
  - The *ctrlist* argument of each chained call contains a list of counters that follows the counters used in the preceding call.

**`const unsigned char *key`**
: Pointer to a valid AES key.

**`unsigned int key_length`**
: Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`const unsigned char *ctrlist`**
: Pointer to a readable buffer that is both of a size greater than or equal to *data_length*, and a multiple of the cipher block size (16 bytes for AES). *ctrlist* should contain a list of precomputed counter values, each of the same size as the cipher block.

**`unsigned int direction`**
: **0** Use the decrypt function.
: **1** Use the encrypt function.

## Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

# ica_aes_ecb

## Purpose

Encrypt or decrypt data with an AES key using Electronic Code Book (ECB) mode, as described in NIST Special Publication 800-38A Chapter 6.1.

### Format

```
unsigned int ica_aes_ecb(const unsigned char *in_data,
  unsigned char *output,
  unsigned int data_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned int direction);
```

### Required hardware support

KM-AES-128, KM-AES-192, or KM-AES-256

### Parameters

**`const unsigned char *in_data`**
: Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
: Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
: Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. *data_length* must be a multiple of the cipher block size (a multiple of 16 for AES).

**`const unsigned char *key`**
: Pointer to a valid AES key.

**`unsigned int key_length`**
: Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned int direction`**
: **0** Use the decrypt function.
: **1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_aes_gcm

### Purpose

Encrypt data and authenticate data or decrypt data and check authenticity of data with an AES key using the Galois/Counter (GCM) mode, as described in NIST Special Publication 800-38D. If no message needs to be encrypted or decrypted and only authentication or authentication checks are requested, then this method implements the GMAC mode.

### Format

```
unsigned int ica_aes_gcm(unsigned char *plaintext,
  unsigned long plaintext_length,
  unsigned char *ciphertext,
  const unsigned char *iv,
  unsigned int iv_length,
```

```
const unsigned char *aad,
unsigned long aad_length,
unsigned char *tag,
unsigned int tag_length,
const unsigned char *key,
unsigned int key_length,
unsigned int direction);
```

## Required hardware support

KM-AES-128, KM-AES-192 or KM-AES-256
KIMD-GHASH
KMCTR-AES-128, KMCTR_AES-192 or KMCTR-AES-256

## Parameters

**`unsigned char *plaintext`**
Pointer to a buffer of size greater than or equal to *plaintext_length* bytes. If *direction* is equal to 1, the *plaintext* buffer must be readable and contain a payload message of size *plaintext_length* to be encrypted. If direction is equal to 0, the *plaintext* buffer must be writable and if the authentication verification succeeds, the decrypted message in the most significant *plaintext_length* bytes of *ciphertext* is written to the buffer. Otherwise, the contents of the buffer are undefined.

**`unsigned long plaintext_length`**
Length in bytes of the message to be encrypted or decrypted. This value can be 0 unless *aad_length* is equal to 0. The value must be greater than or equal to 0 and less than (2**36) - 32.

**`unsigned char *ciphertext`**
Pointer to a buffer of size greater than or equal to *plaintext_length* bytes. If *direction* is equal to 1, then this buffer must be writable and the encrypted message from *plaintext* is written to that buffer. If *direction* is equal to 0, then this buffer is readable and contains an encrypted message of length *plaintext_length*.

**`const unsigned char *iv`**
Pointer to a readable buffer of size greater than or equal to *iv_length* bytes, which contains an initialization vector of size *iv_length*.

**`unsigned int iv_length`**
Length in bytes of the initialization vector in *iv*. The value must be greater than 0 and less than 2**61. A length of 12 is recommended.

**`const unsigned char *aad`**
Pointer to a readable buffer of size greater than or equal to *aad_length* bytes. The additional authenticated data in the most significant *aad_length* bytes is subject to the message authentication code computation, but is not encrypted.

**`unsigned int aad_length`**
Length in bytes of the additional authenticated data in *aad*. The value must be greater than or equal to 0 and less than 2**61.

**`unsigned char *tag`**
Pointer to a buffer of size greater than or equal to *tag_length* bytes. If *direction* is equal to 1, this buffer must be writable, and a message authentication code for the additional authenticated data in *aad* and the plain text in *plaintext* of size *tag_length* bytes is written to this buffer. If *direction* is equal to 0, this buffer must be readable and contain a message authentication code to be verified against the additional authenticated data in *aad* and the decrypted cipher text from *ciphertext*.

**unsigned int tag_length**
Length in bytes of the message authentication code *tag* in bytes. Valid values are: 4, 8, 12, 13, 14, 15, and 16.

**const unsigned char *key**
Pointer to a valid AES key.

**unsigned int key_length**
Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**unsigned int direction**
- **0** Verify message authentication code and decrypt encrypted payload.
- **1** Encrypt payload and compute message authentication code for the additional authenticated data and the payload.

### Return codes

**0** Success

**EFAULT**
If *direction* is equal to 0 and the verification of the message authentication code fails.

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_aes_ofb

### Purpose

Encrypt or decrypt data with an AES key using Output Feedback (OFB) mode, as described in NIST Special Publication 800-38A Chapter 6.4.

### Format

```
unsigned int ica_aes_ofb(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key,
  unsigned int key_length,
  unsigned char *iv,
  unsigned int direction);
```

### Required hardware support

KMO-AES-128, KMO-AES-192, or KMO-AES-256

### Parameters

**const unsigned char *in_data**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**unsigned char *out_data**
Pointer to a writable buffer that to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**unsigned long data_length**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*.

**`const unsigned char *key`**
Pointer to a valid AES key.

**`unsigned int key_length`**
Length in bytes of the AES key. Supported sizes are 16, 24, and 32 for AES-128, AES-192, and AES-256 respectively. Therefore, you can use the definitions: **AES_KEY_LEN128**, **AES_KEY_LEN192**, and **AES_KEY_LEN256**.

**`unsigned char *iv`**
Pointer to a valid initialization vector of the same size as the cipher block, in bytes (16 bytes for AES). This vector is overwritten during the function. If *data_length* is a multiple of the cipher block size (16 bytes for AES), the result value in *iv* can be used as the initialization vector for a chained **ica_aes_ofb** call with the same key.

**`unsigned int direction`**
**0** Use the decrypt function.
**1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_aes_xts

### Purpose

Encrypt or decrypt data with an AES key using the XEX Tweakable Bloc Cipher with Ciphertext Stealing (XTS) mode, as described in NIST Special Publication 800-38E and IEEE standard 1619-2007.

### Format

```
unsigned int ica_aes_xts(const unsigned char *in_data,
  unsigned char *out_data,
  unsigned long data_length,
  const unsigned char *key1,
  const unsigned char *key2,
  unsigned int key_length,
  unsigned char *tweak,
  unsigned int direction);
```

### Required hardware support

KM-XTS-AES-128, or KM-XTS-AES-256
PCC-Compute-XTS-Parameter-Using-AES-128, or PCC-Compute-XTS-Parameter-Using-AES-256

### Parameters

**`const unsigned char *in_data`**
Pointer to a readable buffer that contains the message to be encrypted or decrypted. The size of the message in bytes is *data_length*. The size of this buffer must be at least as large as *data_length*.

**`unsigned char *out_data`**
Pointer to a writable buffer to contain the resulting encrypted or decrypted message. The size of this buffer in bytes must be at least as large as *data_length*.

**`unsigned long data_length`**
Length in bytes of the message to be encrypted or decrypted, which resides at the beginning of *in_data*. The minimal value of data_length is 16.

**`const unsigned char *key1`**
Pointer to a buffer containing a valid AES key. *key1* is used for the actual encryption of the message buffer, combined with some vector computed from the *tweak* value (Key1 in IEEE Std 1619-2007).

**`const unsigned char *key2`**
Pointer to a buffer containing a valid AES key *key2* is used to encrypt the tweak (Key2 in IEEE Std 1619-2007).

**`unsigned int key_length`**
The length in bytes of the AES key. XTS supported AES key sizes are 16 and 32, for AES-128 and AES-256 respectively. Therefore, you can use:

2 * **AES_KEY_LEN128** and 2 * **AES_KEY_LEN256**.

**`unsigned char *tweak`**
Pointer to a valid 16-byte tweak value (as in IEEE standard 1619-2007). This tweak is overwritten during the function. If *data_length* is a multiple of the cipher block size (a multiple of 16 for AES), the result value in *tweak* can be used as the *tweak* value for a chained **ica_aes_xts** call with the same key pair.

**`unsigned int direction`**
**0** Use the decrypt function.
**1** Use the encrypt function.

### Return codes

**0** Success

For return codes indicating exceptions, see “Return codes” on page 76.

## Compatibility with earlier versions

In order to stay compatible with earlier versions of libica, the following AES interfaces remain supported:

```
unsigned int ica_aes_encrypt(unsigned int mode,
 unsigned int data_length, unsigned char *input_data,
 ica_aes_vector_t *iv, unsigned int key_length, unsigned char *aes_key,
 unsigned char *output_data);
```

```
unsigned int ica_aes_decrypt(unsigned int mode,
 unsigned int data_length, unsigned char *input_data,
 ica_aes_vector_t *iv, unsigned int key_length, unsigned char *aes_key,
 unsigned char *output_data);
```

Table 4 shows libica version 2.0 AES functions calls, and their corresponding libica version 2.4 AES function calls.

*Table 4. Compatibility of libica version 2.0 AES functions calls to libica version 2.4 AES function calls*

| Calling this libica version 2.0 AES function | Corresponds to calling this libica version 2.4 AES function |
|---|---|
| `ica_aes_encrypt(MODE_ECB, data_length,in_data,NULL, key_length,key,out_data);` | `ica_aes_ecb(in_data,out_data,(long)data_length, key,key_length,1);` |
| `ica_aes_encrypt(MODE_CBC,data_length,in_data,iv, key_length,key,out_data);` | `ica_des_cbc(in_data,out_data,(long)data_length, key,key_length,iv,1);` |
| `ica_aes_decrypt(MODE_ECB,data_length,in_data,NULL, key_length,key,out_data);` | `ica_aes_ecb(in_data,out_data,(long)data_length, key,key_length,0);` |

Table 4. Compatibility of libica version 2.0 AES functions calls to libica version 2.4 AES function calls (continued)

| Calling this libica version 2.0 AES function | Corresponds to calling this libica version 2.4 AES function |
|---|---|
| `ica_aes_decrypt(MODE_CBC,data_length,in_data,iv, key_length,key,out_data);` | `ica_aes_cbc(in_data,out_data,(long)data_length, key,key_length,iv,0);` |

The functions **ica_aes_encrypt** and **ica_aes_decrypt** remain supported, but their use is discouraged in favor of **ica_aes_ecb** and **ica_aes_cbc**.

For a detailed description of the earlier APIs, see *libica Programmers Reference* version 2.0.

# Information retrieval function

These functions are included in: `include/ica_api.h`.

## ica_get_version

### Purpose

Return libica version information.

### Format

```
unsigned int ica_get_version(libica_version_info *version_info);
```

### Parameters

**`libica_version_info *version_info`**
Pointer to a *libica_version_info* structure. The structure is filled with the current libica version information.

### Return codes

**0** Success

For return codes indicating exceptions, see "Return codes" on page 76.

## ica_get_functionlist

### Purpose

Returns a list of crypto mechanisms supported by libica.

### Format

```
unsigned int ica_get_functionlist(libica_func_list_element *mech_list,
unsigned int *mech_list_len);
```

### Parameters

**`libica_func_list_element *mech_list`**
Null or pointer to an array of at least as many *libica_func_list_element* structures as denoted in the **mech_list_len* argument. If the value in the **mech_list_len* argument is equal to or greater than the number of mechanisms available in libica then the *libica_func_list_element* structures in **mech_list* are filled (in the order of the array indices) with information for the supported otherwise the **mech_list* argument remains unchanged.

**`unsigned int *mech_list_len`**

Pointer to an integer which contain the actual number of array elements (number of structures). If **mech_list* was NULL the contents of **mech_list_len* will be replaced by the number of mechanisms available in libica.

## Return codes

**0** Success

**EINVAL**

The value in **mech_list* is to small

For return codes indicating exceptions, see "Return codes" on page 76.

## Recommended usage

First call **`ica_get_functionlist`** with a NULL mechanism list, then allocate the mechanism list according to number of mechanisms in libica returned by that function, and then call **`ica_get_functionlist`** with the allocated mechanism list.

# Chapter 4. Accessing libica functions through the PKCS #11 (openCryptoki)

Learn how the cryptographic functions provided by libica can be accessed using the PKCS #11 API implemented by openCryptoki is described in this section.

For more information about PKCS #11 standard, see `PKCS #11 Cryptographic Token Interface Standard`

## openCryptoki overview

openCryptoki consists of an implementation of the PKCS #11 API, a slot manager, an API for slot token dynamic link libraries (STDLLs), and a set of STDLLs (or tokens). The libica token is such a STDLL introduced into openCryptoki.

The openCryptoki base library (`libopencryptoki.so`) provides the generic API as outlined in the PKCS #11 specification (version 2.20). This library also loads token-specific modules (STDLLs) that provide the token specific implementation of the PKCS #11 API and cryptographic functions (for example, session management, object management, and crypto algorithms). For a description of the PKCS #11 version 2.20 standard, refer to the following URL: PKCS #11 Cryptographic Token Interface Standard

A global configuration file (`/etc/opencryptoki/opencryptoki.conf`) is provided which describes the available tokens. This configuration file can be customized for the individual tokens. The openCryptoki package contains man pages that describe the format of the configuration files. For more information, see “Adjusting the openCryptoki configuration file” on page 65.

The libica token is a plug-in into the openCryptoki token library, providing support for several cryptographic algorithms.

### Slot manager

The slot manager (**`pkcsslotd`**) runs as a daemon. Upon start-up, it creates a shared memory segment and reads the openCryptoki configuration file to acquire the available token and slot information. The openCryptoki API attaches to this memory segment to retrieve token information. Thus, the slot manager provides the openCryptoki API with the token information when required. An application in turn links to or loads the openCryptoki API.

### Slot token dynamic link libraries (STDLLs)

The libica token is an example of an STDLL within openCryptoki. STDLLs are plug-in modules to the openCryptoki (main) API. They provide token-specific functions that implement the interfaces. Specific devices can be supported by building an appropriate STDLL. Figure 1 on page 63 illustrates the stack and the process flow in a System z environment.

The STDLLs require local disk space to store persistent data, such as token information, personal identification numbers (PINs) and token objects. This information is stored in a separate directory for each token (for example in `/var/lib/opencryptoki/lite` for the libica token). Within each of these directories there is a sub-directory `TOK_OBJ` that contains the token objects (token key store). Each private token object is represented by an encrypted file. Most of these directories are created during installation of openCryptoki.

## The `pkcsconf` command line program

openCryptoki provides a command line program (`/usr/sbin/pkcsconf`) to configure and administer tokens that are supported within the system. The `pkcsconf` capabilities include token initialization, and security officer (SO) PIN and user PIN initialization and maintenance.

`pkcsconf` operations that address a specific token must specify the slot that contains the token with the **`-c`** option. You can view the list of tokens present within the system by specifying the **`-t`** option (without **`-c`** option). For example, the following code shows the options for the `pkcsconf` command and displays slot information for the system:

```
# pkcsconf ?
usage: pkcsconf [-itsmlIupPh] [-c slotnumber -U user-PIN -S SO-PIN -n new PIN]
```

The available options have the following meanings:

**-i** display PKCS11 info

**-t** display token info

**-s** display slot info

**-m** display mechanism list

**-l** display slot description

**-I** initialize token

**-u** initialize user PIN

**-p** set the user PIN

**-P** set the SO PIN

**-h | --help | ?**
show pkcsconf help information

**-c** specify the token slot for the operation

**-U** the current user PIN (for use when changing the user pin with -u and -p options); if not specified, user will be prompted

**-S** the current Security Officer (SO) pin (for use when changing the SO pin with -P option); if not specified, user will be prompted

**-n** the new pin (for use when changing either the user pin or the SO pin with -u, -p or -P options); if not specified, user will be prompted

For more information about the `pkcsconf` command, see the `pkcsconf` man page.

Figure 1 on page 63 illustrates the stack and the process flow:

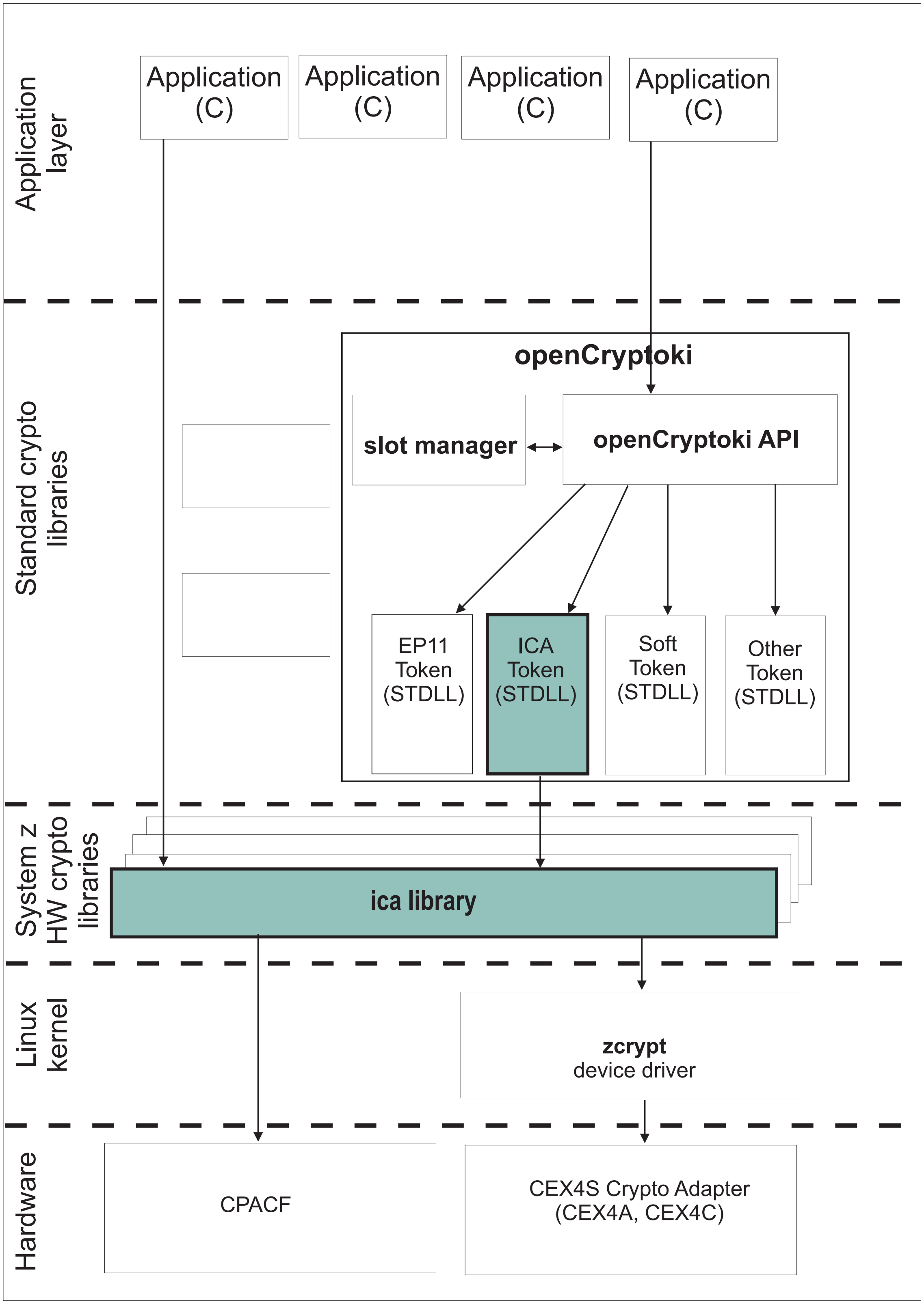


*Figure 1. Stack and process flow*

## Functions provided by openCryptoki with the ica token

The PKCS #11 functions that manage tokens, slots, and sessions are described in the PKCS #11 standard.

For an overview of the algorithms supported by the ica token, see "Supported mechanisms for the ica token" on page 70.

The PKCS #11 standard describes the exact API for the mentioned mechanisms. For more information, see

`http://www.rsa.com/rsalabs/`

For more details about how to use openCryptoki, see "Using openCryptoki" on page 70.

## Installing openCryptoki

openCryptoki is shipped with the Linux on System z distributions. Follow the instructions in this section to install openCryptoki.

Check whether you have already installed openCryptoki in your current environment:

```
$ rpm -qa | grep -i opencryptoki
```

**Note:** This command example is distribution dependent. `opencryptoki` must in certain distribution be specified as `openCryptoki` (case-sensitive).

You should see all installed openCryptoki packages. If required packages are missing, use the installation tool of your Linux distribution to install the appropriate openCryptoki RPM.

**Note:** You must remove any previous package of openCryptoki, before you can install the new package version 3.1.

### Installing from the RPM

The current distributions already provide the openCryptoki binary RPMs.

The openCryptoki version 3.1 or higher packages, are delivered by the distributors. Distributors build these packages as RPM packages for delivering them to customers.

Customers can install these openCryptoki RPM packages by using the installation tool of their selected distribution.

If you received openCryptoki as an *RPM* package, follow the *RPM* installation process that is described in the *RPM* man page. This process is the preferred installation method.

### Installing from the source package

If you prefer, you can install openCryptoki from the source package.

As an alternative, for example for development purposes, you can get the latest openCryptoki version (inclusive latest patches) from the sourceforge repository (sourceforge.net/projects/opencryptoki) and build it yourself. But this version is not serviced. It is suitable for non-production systems and early feature testing, but you should not use it for production.

1. Download the latest version of the openCryptoki sources from:

   `http://sourceforge.net/projects/opencryptoki/files/opencryptoki/v3.1/`
2. Decompress and extract the compressed tape archive (TGZ file). There is a new directory named `opencryptoki`.
3. Change to that directory and issue the following scripts and commands:

   ```
   $ ./bootstrap
   $ ./configure
   $ make
   $ make install
   ```

   The scripts or commands perform the following functions:

   **bootstrap**
   Initial setup, basic configurations

   **configure**
   Check configurations and build the makefile

   **make** Compile and link

   **make install**
   Install the libraries

**Note:** When installing openCryptoki from the source package, the location of some installed files will differ from the location of files installed from an RPM.

# Configuring openCryptoki

After a successful installation of openCryptoki, you need to perform certain configuration and customization tasks to enable the exploitation of the libica functions from applications. Especially, you need to set up tokens and daemons and then initialize the tokens.

openCryptoki, and in particular the slot manager, can handle several tokens, which can have different support for different hardware devices or software solutions. As shown in Figure 1 on page 63, libica interacts with the libica library host part. libica can operate with the Crypto Express4S (CEX4S) adapter (CEX4A and CEX4C) for symmetric and asymmetric cryptographic functions.

For a complete configuration of openCryptoki, finish the tasks as described in the contained subtopics:
- “Adjusting the openCryptoki configuration file”
- “Configuring the ica token” on page 68
- “Initializing the token” on page 68
- “How to recognize the ica token” on page 69

Finally, to control your configuration results, follow the instructions provided in “How to recognize the ica token” on page 69.

## Adjusting the openCryptoki configuration file

A preconfigured list of all available tokens that are ready to register to the openCryptoki slot daemon is required before the openCryptoki daemon can start.

This list is provided by the global configuration file. Read this topic for information on how to adapt this file according to your installation.

Table 5 provides an overview of supported libraries (tokens) that may be in place after you have successfully installed openCryptoki. The list may vary for different distributions and is dependent from the installed RPM packages.

Also, Linux on System z does not support the TPM token library.

A token is only available, if the token library is installed, and the appropriate software and hardware support pertaining to the stack of the token is also installed.

A token needs not be available, even if the corresponding token library is installed. Display the list of available tokens by using the command:

```
$ pkcsconf -t
```

*Table 5. openCryptoki libraries*

| Library | Explanation |
| --- | --- |
| /usr/lib64/opencryptoki/libopencryptoki.so | openCryptoki base library |
| /usr/lib64/opencryptoki/stdll/libpkcs11_ica.so | ica token library |
| /usr/lib64/opencryptoki/stdll/libpkcs11_sw.so | software token library |
| /usr/lib64/opencryptoki/stdll/libpkcs11_tpm.so | TPM token library |
| /usr/lib64/opencryptoki/stdll/libpkcs11_cca.so | CCA token library |
| /usr/lib64/opencryptoki/stdll/libpkcs11_ep11.so | EP11 token library |
| /usr/lib64/opencryptoki/stdll/libpkcs11_icsf.so | ICSF token library |

**Note:** An analogous set of libraries is available for 32 bit compatibility mode.

Sample configuration file:

```
------------- content of opencryptoki.conf ---------
version opencryptoki-3.1

# The following defaults are defined:
#       hwversion = 0.0
#       firmwareversion = 0.0
#       description = Linux
#       manufacturer = IBM
#
# The slot definitions below may be overriden and/or customized.
# For example:
#       slot 0
#       {
#          stdll = libpkcs11_cca.so
#          description = "OCK CCA Token"
#          manufacturer = "MyCompany Inc."
#          hwversion = 2.32
#          firmwareversion = 1.0
#       }
# See man(5) opencryptoki.conf for further information.
#
slot 0
{
stdll = libpkcs11_tpm.so
}

slot 1
{
stdll = libpkcs11_ica.so
description = "ICA Token"
manufacturer = "IBM"
hwversion = 1.0
firmwareversion = 1.0
}

slot 2
{
stdll = libpkcs11_cca.so
}

slot 3
{
stdll = libpkcs11_sw.so
}

slot 4
{
stdll = libpkcs11_ep11.so
confname = ep11tok.conf
}
---------------------------- end ----------------------------------
```

**Note:**

- The standard path for slot token dynamic link libraries (STDLLs) is: `/usr/lib64/opencryptoki/stdll/`.

Use one of the following command to start the slot-daemon, which reads out the configuration information and sets up the tokens:

```
$ pkcsslotd start
$ service pkcsslotd start
```

For a permanent solution, for example, for an automatic start-up of the slot-daemon, refer to the distribution documentation.

## Configuring the ica token

You need to connect the libica library to the ica token. For this purpose, you should check the slot entry definition in the openCryptoki configuration file.

Each token has its own token directory, which is used by openCryptoki to store token-specific information (like for example, key objects, user PIN, or SO PIN). The ica token directory is `/var/lib/opencryptoki/lite/`.

**Note:** This configuration is token-based. It applies to all applications that use this ica token.

### Defining the slot entry for the ica token in openCryptoki

Normally, the default openCryptoki configuration file `opencryptoki.conf` already provides a slot entry for the ica token. It is preconfigured to slot #1. Check this default entry to find out whether you can use it as is. If it is missing, then define a slot entry that sets the `stdll` attribute to `libpkcs11_ica.so`.

## Initializing the token

Once the configuration files of openCryptoki and the ica token are set up, and the **pkcsslotd** daemon is started, the ica token must be initialized.

**Note:** PKCS #11 defines two users for each token: a security officer (SO) whose responsibility is the administration of the token, and a standard user (User) who wants to use the token to perform cryptographic operations. openCryptoki requires that for both the SO and the User a log-in PIN is defined as part of the token initialization.

The following command provides some useful slot information:

```
# pkcsconf -s

Slot #0 Info
        Description: EP11 Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 4.0
        Firmware Version: 2.11
Slot #1 Info
        Description: ICA Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 4.0
        Firmware Version: 2.10
```

Find your preferred token in the details list and select the correct slot number. This number is used in the next initialization steps to identify your token:

```
$ pkcsconf -I -c <slot> // Initialize the Token and setup a Token Label

$ pkcsconf -P -c <slot> // change the SO PIN (recommended)

$ pkcsconf -u -c <slot> // Initialize the User PIN (SO PIN required)

$ pkcsconf -p -c <slot> // change the User PIN (optional)
```

**pkcsconf -I**
: During token initialization, you are asked for a token label. Provide a meaningful name, because you might need this reference for identification purposes.

**pkcsconf -P**
: For security reasons, openCryptoki requires that you change the default SO PIN (87654321) to a different value. Use the `pkcsconf -P` option to change the SO PIN.

**pkcsconf -u**
: When you enter the user PIN initialization you are asked for the newly set SO PIN. The length of the user PIN must be 4 - 8 characters.

**pkcsconf -p**
: You must at least once change the user PIN with `pkcsconf -p` option. After you completed the PIN setup, the token is prepared and ready for use.

**Note:** An initialization (`pkcsconf -u` option) with 12345678 will work without any issues. However, this is not recommended, because this pattern is checked internally and marked as default PIN. Therefore, change to a user PIN that is different from 12345678.

## How to recognize the ica token

You can use the **pkcsconf -t** command to display a table that shows all available tokens. You can check the slot and token information, and the PIN status at any time.

The following information provided by the **pkcsconf -t** command about the ica token is returned in the *Token Info* section, where, for example, `Token #1 Info` displays information about the token plugged into slot number 1.

```
$ pkcsconf -t

Token #1 Info:
       Label: IBM ICA  PKCS #11
 Manufacturer: IBM Corp.
 Model: IBM ICA
 Serial Number: 123
 Flags: 0x880045 (RNG|LOGIN_REQUIRED|CLOCK_ON_TOKEN|USER_PIN_TO_BE_CHANGED|
                       SO_PIN_TO_BE_CHANGED)
 Sessions: 0/-2
 R/W Sessions: -1/-2
 PIN Length: 4-8
 Public Memory: 0xFFFFFFFF/0xFFFFFFFF
 Private Memory: 0xFFFFFFFF/0xFFFFFFFF
 Hardware Version: 1.0
 Firmware Version: 1.0
 Time: 14:16:45
```

The most important information is as follows:

- The token **Label** you assigned at the initialization phase (`IBM ICA PKCS #11`, in the example). You can initialize or change a token label by using the `pkcsconf -I` command.
- The **Model** name is unique and designates the token that is in use.
- The **Flags** provide information about the token initialization status, the PIN status, and features such as ***Random Number Generator*** (RNG). They also provide information about requirements, such as ***Login required***, which means that there is at least one mechanism that requires a session log-in to use that cryptographic function.

The flag USER_PIN_TO_BE_CHANGED indicates that the user PIN must be changed before the token can be used. The flag SO_PIN_TO_BE_CHANGED indicates that the SO PIN must be changed before administration commands can be used.

For more information about the flags provided in this output, see the description of the TOKEN_INFO structure and the Token Information Flags in the PKCS #11 Cryptographic Token Interface Standard.

- The **PIN length** range declared for this token.

## Using openCryptoki

How you can get status information about openCryptoki is described in this section.

For a list of code samples, refer to "Coding samples (C)" on page 154.

### Supported mechanisms for the ica token

View a list of the supported mechanisms for the ica token in the openCryptoki implementation.

Use the following command to retrieve a complete list of algorithms (or mechanisms) that are supported by the token:

```
$ pkcsconf -m -c <slot>
Mechanism #2
        Mechanism: 0x131 (CKM_DES3_KEY_GEN)
        Key Size: 24-24
        Flags: 0x8001 (CKF_HW|CKF_GENERATE)
...
Mechanism #10
        Mechanism: 0x132 (CKM_DES3_ECB)
        Key Size: 24-24
        Flags: 0x60301 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_WRAP|CKF_UNWRAP)
Mechanism #11
        Mechanism: 0x133 (CKM_DES3_CBC)
        Key Size: 24-24
        Flags: 0x60301 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_WRAP|CKF_UNWRAP)
...
```

The list displays all mechanisms supported by this token. The mechanism ID and name corresponds to the PKCS #11 specification. Each mechanism provides its supported key size and the some further properties such as hardware support and mechanism information flags. These flags provide information about the PKCS #11 functions that may use the mechanism. Typical functions are for example, *encrypt*, *decrypt*, *wrap key*, *unwrap key*, *sign*, or *verify*.

*Table 6. Supported mechanism list for the ica token.*

| Mechanisms | ica token | supported with openCryptoki version |
|---|---|---|
| CKM_RSA_PKCS_KEY_PAIR_GEN | x | 2.4 |
| CKM_RSA_PKCS | x | 2.4 |
| CKM_RSA_X_509 | x | 2.4 |
| CKM_MD5_RSA_PKCS | x | 2.4 |
| CKM_SHA1_RSA_PKCS | x | 2.4 |
| CKM_SHA256_RSA_PKCS | x | 2.4.3.1 |

Table 6. Supported mechanism list for the ica token (continued).

| Mechanisms | ica token | supported with openCryptoki version |
|---|---|---|
| CKM_SHA384_RSA_PKCS | x | 2.4.3.1 |
| CKM_SHA512_RSA_PKCS | x | 2.4.3.1 |
| | | |
| CKM_DES_OFB64 | x | 3.0 |
| CKM_DES_KEY_GEN | x | 2.4 |
| CKM_DES_ECB | x | 2.4 |
| CKM_DES_CFB8 | x | 3.0 |
| CKM_DES_CFB64 | x | 3.0 |
| CKM_DES_CBC | x | 2.4 |
| CKM_DES_CBC_PAD | x | 2.4 |
| | | |
| CKM_DES3_MAC | x | 3.0 |
| CKM_DES3_MAC_GENERAL | x | 3.0 |
| CKM_DES3_KEY_GEN | x | 2.4 |
| CKM_DES3_ECB | x | 2.4 |
| CKM_DES3_CBC | x | 2.4 |
| CKM_DES3_CBC_PAD | x | 2.4 |
| | | |
| CKM_MD5 | x | 2.4 |
| CKM_MD5_HMAC | x | 2.4 |
| CKM_MD5_HMAC_GENERAL | x | 2.4 |
| | | |
| CKM_SHA_1 | x | 2.4 |
| CKM_SHA_1_HMAC | x | 2.4 |
| CKM_SHA_1_HMAC_GENERAL | x | 2.4 |
| | | |
| CKM_SHA256 | x | 2.4 |
| CKM_SHA256_HMAC | x | 2.4 |
| CKM_SHA256_HMAC_GENERAL | x | 2.4 |
| | | |
| CKM_SHA384 | x | 2.4 |
| CKM_SHA384_HMAC | x | 2.4.3.1 |
| CKM_SHA384_HMAC_GENERAL | x | 2.4.3.1 |
| | | |
| CKM_SHA512 | x | 2.4 |
| CKM_SHA512_HMAC | x | 2.4.3.1 |
| CKM_SHA512_HMAC_GENERAL | x | 2.4.3.1 |
| | | |
| CKM_SSL3_PRE_MASTER_KEY_GEN | x | 2.4 |

*Table 6. Supported mechanism list for the ica token (continued).*

| Mechanisms | ica token | supported with openCryptoki version |
| --- | --- | --- |
| CKM_SSL3_MASTER_KEY_DERIVE | x | 2.4 |
| CKM_SSL3_KEY_AND_MAC_DERIVE | x | 2.4 |
| CKM_SSL3_MD5_MAC | x | 2.4 |
| CKM_SSL3_SHA1_MAC | x | 2.4 |
| | | |
| CKM_AES_OFB | x | 3.0 |
| CKM_AES_MAC | x | 3.0 |
| CKM_AES_MAC_GENERAL | x | 3.0 |
| CKM_AES_KEY_GEN | x | 2.4 |
| CKM_AES_ECB | x | 2.4 |
| CKM_AES_CFB8 | x | 3.0 |
| CKM_AES_CFB64 | x | 3.0 |
| CKM_AES_CFB128 | x | 3.0 |
| CKM_AES_CBC | x | 2.4 |
| CKM_AES_CBC_PAD | x | 2.4 |
| CKM_AES_CTR | x | 2.4 |

# Chapter 5. libica constants, type definitions, data structures, and return codes

Use these constants, type definitions, data structures, and return codes when you program with the libica APIs.

The APIs are described in Chapter 3, "libica version 2.4 application programming interfaces," on page 7. To use them, include `ica_api.h` in your programs.

## libica constants

The constants listed in this topic are provided and valid for the current libica version.

Use these constants instead of the equivalent libica version 1 constants. There is no difference in their values.

```
#define ica_adapter_handle_t int
#define SHA_HASH_LENGTH 20
#define SHA1_HASH_LENGTH SHA_HASH_LENGTH
#define SHA224_HASH_LENGTH 28
#define SHA256_HASH_LENGTH 32
#define SHA384_HASH_LENGTH 48
#define SHA512_HASH_LENGTH 64
#define ica_aes_key_t ica_key_t
#define ICA_ENCRYPT 1
#define ICA_DECRYPT 0
```

## Type definitions

These type definitions are available to ensure compatibility with libica version 1 types.

```
typedef ica_des_vector_t ICA_DES_VECTOR;
typedef ica_des_key_single_t ICA_KEY_DES_SINGLE;
typedef ica_des_key_triple_t ICA_KEY_DES_TRIPLE;
typedef ica_aes_vector_t ICA_AES_VECTOR;
typedef ica_aes_key_single_t ICA_KEY_AES_SINGLE;
typedef ica_aes_key_len_128_t ICA_KEY_AES_LEN128;
typedef ica_aes_key_len_192_t ICA_KEY_AES_LEN192;
typedef ica_aes_key_len_256_t ICA_KEY_AES_LEN256;
typedef sha_context_t SHA_CONTEXT;
typedef sha256_context_t SHA256_CONTEXT;
typedef sha512_context_t SHA512_CONTEXT;
typedef unsigned char ica_des_vector_t[8];
typedef unsigned char ica_des_key_single_t[8];
typedef unsigned char ica_key_t[8];
typedef unsigned char ica_aes_vector_t[16];
typedef unsigned char ica_aes_key_single_t[8];
typedef unsigned char ica_aes_key_len_128_t[16];
```

```
typedef unsigned char ica_aes_key_len_192_t[24];
typedef unsigned char ica_aes_key_len_256_t[32];
```

## Data structures

These structures are used in the API of the current libica version.

For the definitions of older functions, see previous versions of this book. The older functions are no longer recommended for use, but they are supported.

```
typedef struct {
unsigned int key_length;
unsigned char* modulus;
unsigned char* exponent;
} ica_rsa_key_mod_expo_t;

typedef struct {
unsigned int key_length;
unsigned char* p;
unsigned char* q;
unsigned char* dp;
unsigned char* dq;
unsigned char* qInverse;
} ica_rsa_key_crt_t;

typedef struct {
unsigned int mech_mode_id;
unsigned int flags;
unsigned int property;
} libica_func_list_element;
```

* `mech_mode_id`: Unique mechanism ID for each mechanism implemented in libica

```
#define SHA1 1
#define SHA224 2
#define SHA256 3
#define SHA384 4
#define SHA512 5
#define DES_ECB 20
#define DES_CBC 21
#define DES_CBC_CS 22
#define DES_OFB 23
#define DES_CFB 24
#define DES_CTR 25
#define DES_CTRLST 26
#define DES_CBC_MAC 27
#define DES_CMAC 28
#define DES3_ECB 41
#define DES3_CBC 42
#define DES3_CBC_CS 43
#define DES3_OFB 44
#define DES3_CFB 45
#define DES3_CTR 46
#define DES3_CTRLST 47
#define DES3_CBC_MAC 48
#define DES3_CMAC 49
#define AES_ECB 60
#define AES_CBC 61
#define AES_CBC_CS 62
#define AES_OFB 63
#define AES_CFB 64
#define AES_CTR 65
#define AES_CTRLST 66
#define AES_CBC_MAC 67
#define AES_CMAC 68
#define AES_CCM 69
#define AES_GCM 70
```

```
#define AES_XTS 71
#define P_RNG 80
#define RSA_ME 90
#define RSA_CRT 91
#define RSA_KEY_GEN_ME 92
#define RSA_KEY_GEN_CRT 93
```

For more details regarding these mechanism please refer to the openCryptoki v 2.20 specification.

**`* flags`**
: This flag represents the type of hardware/software support for each mechanism.

**`#define ICA_FLAG_SHW 4`**
: Static hardware support (operations on CPACF). Hardware support will be available unless a hardware error occurs.

**`#define ICA_FLAG_DHW 2`**
: Dynamic hardware support (operations on crypto cards). Hardware support will be available unless the hardware is reconfigured.

**`#define ICA_FLAG_SW 1`**
: Software support. If both static and dynamic hardware support as well as software support are available, then software support is used as fall back if hardware support fails.

**`* property`**
: This property field is optional depending on the mechanism. It is used to declare mechanism specific parameters, such as key sizes for RSA and AES.

For RSA mechanisms:

**`- bit 0`**
: 512 bit key size support

**`- bit 1`**
: 1024 bit key size support

**`- bit 2`**
: 2048 bit key size support

**`- bit 3`**
: 4096 bit key size support

For AES mechanisms:

**`- bit 0`**
: 128 bit key size support

**`- bit 1`**
: 192 bit key size support

**`- bit 2`**
: 256 bit key size support

For all non-RSA/AES mechanisms this field is empty.

Take note of these considerations:

- The buffers pointed to by members of type *unsigned char ** must be manually allocated and deallocated by the user.
- Key parts must always be right-aligned in their fields.

- All buffers pointed to by members *modulus* and *exponent* in struct *ica_rsa_key_mod_expo_t* must be of length *key_length*.
- All buffers pointed to by members *p*, *q*, *dp*, *dq*, and *qInverse* in struct *ica_rsa_key_crt_t* must be of size *key_length* / 2 or larger.
- In the struct *ica_rsa_key_crt_t*, the buffers *p*, *dp*, and *qInverse* must contain 8 bytes of zero padding in front of the actual values.
- If an exponent is set in struct *ica_rsa_key_mod_expo_t* as part of a public key for key generation, be aware that due to a restriction in OpenSSL, the public exponent cannot be larger than a size of unsigned long. Therefore, you must have zeros left-padded in the buffer pointed to by *exponent* in the struct *ica_rsa_key_mod_expo_t* struct. Be aware that this buffer also must be of size *key_length*.
- This *key_length* value should be calculated from the length of the modulus in bits, according to this calculation:

  ```
  key_length = (modulus_bits + 7) / 8
  ```

```
typedef struct {
	uint64_t runningLength;
	unsigned char shaHash[LENGTH_SHA_HASH];
} sha_context_t;

typedef struct {
	uint64_t runningLength;
	unsigned char sha256Hash[LENGTH_SHA256_HASH];
} sha256_context_t;

typedef struct {
	uint64_t runningLengthHigh;
	uint64_t runningLengthLow;
	unsigned char sha512Hash[LENGTH_SHA512_HASH];
} sha512_context_t;

typedef struct {
	unsigned int major_version;
	unsigned int minor_version;
	unsigned int fixpack_version;
} libica_version_info;
```

## Return codes

The current libica functions use the standard Linux return codes listed in this topic.

**0** Success

**EFAULT**
The message authentication failed.

**EINVAL**
Incorrect parameter

**EIO** I/O error

**EPERM**
Operation not permitted by Hardware (CPACF).

**ENODEV**
No such device

**ENOMEM**
Not enough memory

**errno** When libica calls **open**, **close**, **begin_sigill_section**, or OpenSSL function **RSA_generate_key**, the error codes of these programs are returned.

# Chapter 6. libica tools

The libica package includes tools to investigate the capabilities of your cryptographic hardware and how these capabilities are used by applications that use libica.

## icainfo - Show available libica functions

Use the **icainfo** command to find out which libica functions are available on your Linux system.

### Format

**icainfo syntax**

```
►►──icainfo──┬──────┬──────────────────────────────────────────────►◄
             ├─ -v ─┤
             └─ -h ─┘
```

Where:

**-v or --version**
: Displays the version number of **icainfo**, then exits.

**-h or --help**
: Displays help information for the command.

### Examples

To obtain an overview of the supported algorithms with modes of operations and how they are implemented on your Linux system (hardware, software, or both), enter:

```
# icainfo
```

View the output produced by this command:

```
The following CP Assist for Cryptographic Function (CPACF) operations are
supported by libica on this system:

 function      | # hardware | #software
---------------+------------+--------------

 SHA-1         |    yes     |      yes
 SHA-224       |    yes     |      yes
 SHA-256       |    yes     |      yes
 SHA-384       |    yes     |      yes
 SHA-512       |    yes     |      yes
   P_RNG       |    yes     |      yes
  RSA ME       |    yes     |      yes
 RSA CRT       |    yes     |      yes
 DES ECB       |    yes     |      yes
 DES CBC       |    yes     |      yes
```

| | | |
|---|---|---|
| DES CBC CS | yes | no |
| DES OFB | yes | no |
| DES CFB | yes | no |
| DES CTR | yes | no |
| DES CTRLST | yes | no |
| DES CBC MAC | yes | no |
| DES CMAC | yes | no |
| 3DES ECB | yes | yes |
| 3DES CBC | yes | yes |
| 3DES CBC CS | yes | no |
| 3DES OFB | yes | no |
| 3DES CFB | yes | no |
| 3DES CTR | yes | no |
| 3DES CTRLIST | yes | no |
| 3DES CBC MAC | yes | no |
| 3DES CMAC | yes | no |
| AES ECB | yes | yes |
| AES CBC | yes | yes |
| AES CBC CS | yes | no |
| AES OFB | yes | no |
| AES CFB | yes | no |
| AES CTR | yes | no |
| AES CTRLST | yes | no |
| AES CBC MAC | yes | no |
| AES CMAC | yes | no |
| AES CCM | yes | no |
| AES GCM | yes | no |
| AES XTS | yes | no |

## icastats - Show use of libica functions

Use the **icastats** utility to find out whether libica uses hardware acceleration features or works with software fallbacks. **icastats** collects the statistical data per user and not per system.

The command also shows which specific functions of libica are used. For a standard user, **icastats** shows a statistics table with all crypto operations that are used by the user's processes. For the root user, **icastats** provides statistics for all users, or processes, on the system.

The shared memory segment that holds the statistic data is created when a user starts **icastats** or when a program is started, that performs cryptographic operations using libica. Once the shared memory segment exists, it can only be removed by one of the delete options (**-d** or **-D**) provided with the **icastats** utility. Thus, this function collects crypto statistics independently from the process context for continuing availability of data. All cryptographic operations using libica are counted into the statistics.

**Note:** Before deleting the shared memory segment, ensure that there are no running applications that are using this memory segment.

## Format

### icastats syntax

```
►►──icastats──┬────────────────────┬─────────────────────────────────────►◄
              ├─ -A────────────────┤
              ├─ -d────────────────┤
              ├─ -D────────────────┤
              ├─ -r────────────────┤
              ├─ -R────────────────┤
              ├─ -S────────────────┤
              ├─ -U < username> ───┤
              ├─ -h────────────────┤
              └─ -v────────────────┘
```

Where:

**`-A or --all`**
: Shows the statistic tables from all users (for root users only).

**`-d or --delete`**
: Removes the user specific shared memory segment.

**`-D or --delete-all`**
: Removes all shared memory segments (for root users only).

**`-r or --reset`**
: Resets the user statistic data table.

**`-R or --reset-all`**
: Resets all statistic data tables from all users (for root users only).

**`-S or --summary`**
: Shows accumulated statistics from all users (for root users only).

**`-U <username> or --user <username>`**
: Shows statistic data for a dedicated user (for root users only).

**`-h or --help`**
: Displays help information for the command.

**`-v or --version`**
: Displays the version number of **`icastats`**, then exits.

## Examples

To display the current use of libica functions issue:

```
# icastats
```

View an excerpt of a sample output produced by this command:

```
Function     |      # Hardware          |       # Software
-------------+--------------------------+------------------------
                ENC    CRYPT     DEC    |  ENC    CRYPT    DEC
-------------+--------------------------+------------------------
SHA-1        |          100             |            0
SHA-224      |          256             |            0
SHA-256      |            0             |            0
```

```
SHA-384      |         0          |          0
SHA-512      |         0          |          0
P_RNG        |         0          |         100
RSA ME       | 0                0 |  121          22
RSA CRT      | 0                0 |    0           0
DES ECB      | 0                0 |    0           0
DES CBC      | 0                0 |    0           0
...
3DES ECB     | 0                0 |    0           0
3DES CBC     | 0                0 |    0           0
...
AES ECB      | 0                0 |    0           0
AES CBC      | 0                0 |    0           0
...
AES GCM AUTH | 0                0 |    0           0
```

## Logging and error handling

Access failures to the shared memory segments that are used by the **icastats** utility, are logged once via the syslog interface. After a failed attempt to access the shared memory segment, the library no longer collects any statistic data for this application (related to application lifetime and user).

**Example** of syslog message:

```
<date> <machine> <application>: failed to create or access shared memory segment.
```

The **icastats** utility prints an error messages if it cannot create, access, or remove the shared memory segment.

**Note:** The log message may indicate a permission problem with the shared memory segment. An administrator can remove the defect memory segment. The next call of **icastats** should create a new memory segment automatically.

# Chapter 7. Examples

These sample program segments illustrate the use of the libica APIs.

These examples are released under the Common Public License - V1.0, which is stated in full at the end of this chapter. See "Common Public License - V1.0" on page 151.

View a list of examples for libica, and the makefile used to create the library.



## DES with ECB mode example

This program prints the version of libica and then encrypts the contents of a character array (plain_data[]) using DES in ECE mode and a key stored in another character array (des_key[]). The program then decrypts the result and prints it as a string. Intermediate results are written as hex dumps.

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 *
 * Copyright IBM Corp. 2011
 *
 */

#include <stdio.h>
#include <string.h>
#include <errno.h>

#include <ica_api.h>

#define DES_CIPHER_BLOCK_SIZE 8

/* Prints hex values to standard out. */
static void dump_data(unsigned char *data, unsigned long length);
/* Prints a description of the return value to standard out. */
static int handle_ica_error(int rc);

int main(char **argv, int argc)
```

```
{
 int rc;
 libica_version_info version;

 /* This example uses a static key. In real life you would
  * use your real DES key, which is negotiated between the
  * encrypting and the decrypting entity.
  *
  * Note: DES key size is cipher block size (DES_CIPHER_BLOCK_SIZE)
  */
 unsigned char des_key[] = {
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 };

 /* This is the plain data, you want to encrypt. For the
  * encryption mode, used in this example, it is necessary,
  * that the length of the encrypted data is a multiple of
  * cipher block size (DES_CIPHER_BLOCK_SIZE).
  */
 unsigned char plain_data[] = {
  0x55, 0x73, 0x69, 0x6e, 0x67, 0x20, 0x6c, 0x69,
  0x62, 0x69, 0x63, 0x61, 0x20, 0x69, 0x73, 0x20,
  0x73, 0x6d, 0x61, 0x72, 0x74, 0x20, 0x61, 0x6e,
  0x64, 0x20, 0x65, 0x61, 0x73, 0x79, 0x21, 0x00,
 };

 unsigned char cipher_data[sizeof(plain_data)];
 unsigned char decrypt_data[sizeof(plain_data)];

 /* Print out libica version.
  */
 ica_get_version(&version);
 printf("libica version %i.%i.%i\n\n",
        version.major_version,
        version.minor_version,
        version.fixpack_version);

 /* Dump key and plain data to standard output, just for
  * a visual control.
  */
 printf("DES key:\n");
 dump_data(des_key, DES_CIPHER_BLOCK_SIZE);
 printf("plain data:\n");
 dump_data(plain_data, sizeof(plain_data));

 /* Encrypt plain data to cipher data, using libica API.
  */
 rc = ica_des_ecb(plain_data, cipher_data, sizeof(plain_data),
    des_key,
    ICA_ENCRYPT);

 /* Error handling (if necessary).
  */
 if (rc)
  return handle_ica_error(rc);

 /* Dump encrypted data.
  */
 printf("encrypted data:\n");
 dump_data(cipher_data, sizeof(plain_data));

 /* Decrypt cipher data to decrypted data, using libica API.
  * Note: The same DES key must be used for encryption and decryption.
  */
 rc = ica_des_ecb(cipher_data, decrypt_data, sizeof(plain_data),
    des_key,
    ICA_DECRYPT);
```

```
 /* Error handling (if necessary).
  */
 if (rc)
  return handle_ica_error(rc);

 /* Dump decrypted data.
  * Note: Please compare output with the plain data, they are the same.
  */
 printf("decrypted data:\n");
 dump_data(decrypt_data, sizeof(plain_data));

 /* Surprise... :-)
  * Note: The following will only work in this example!
  */
 printf("%s\n", decrypt_data);
}

static void dump_data(unsigned char *data, unsigned long length)
{
 unsigned char *ptr;
 int i;

 for (ptr = data, i = 1; ptr < (data+length); ptr++, i++) {
  printf("0x%02x ", *ptr);
  if ((i % DES_CIPHER_BLOCK_SIZE) == 0)
   printf("\n");
 }
 if (i % DES_CIPHER_BLOCK_SIZE)
  printf("\n");
}

static int handle_ica_error(int rc)
{
 switch (rc) {
 case 0:
  printf("OK\n");
  break;
 case EINVAL:
  printf("Incorrect parameter.\n");
  break;
 case EPERM:
  printf("Operation not permitted by Hardware (CPACF).\n");
  break;
 case EIO:
  printf("I/O error.\n");
  break;
 default:
  printf("unknown error.\n");
 }

 return rc;
}
```

## SHA-256 example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2005, 2009, 2011 */
/* (C) COPYRIGHT International Business Machines Corp. 2005, 2009          */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
```

```
#include <string.h>
#include "ica_api.h"

#define NUM_FIPS_TESTS 3

unsigned char FIPS_TEST_DATA[NUM_FIPS_TESTS][64] = {
  // Test 0: "abc"
  { 0x61,0x62,0x63 },
  // Test 1: "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
  {
0x61,0x62,0x63,0x64,0x62,0x63,0x64,0x65,0x63,0x64,0x65,0x66,0x64,0x65,0x66,0x67,
0x65,0x66,0x67,0x68,0x66,0x67,0x68,0x69,0x67,0x68,0x69,0x6a,0x68,0x69,0x6a,0x6b,
0x69,0x6a,0x6b,0x6c,0x6a,0x6b,0x6c,0x6d,0x6b,0x6c,0x6d,0x6e,0x6c,0x6d,0x6e,0x6f,
0x6d,0x6e,0x6f,0x70,0x6e,0x6f,0x70,0x71,
  },
  // Test 2: 1,000,000 'a' -- don't actually use this... see the special case
  // in the loop below.
  {
0x61,
  },
};

unsigned int FIPS_TEST_DATA_SIZE[NUM_FIPS_TESTS] = {
  // Test 0: "abc"
  3,
  // Test 1: "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
  56,
  // Test 2: 1,000,000 'a'
  1000000,
};

unsigned char FIPS_TEST_RESULT[NUM_FIPS_TESTS][LENGTH_SHA256_HASH] =
{
  // Hash for test 0: "abc"
  {
0xBA,0x78,0x16,0xBF,0x8F,0x01,0xCF,0xEA,0x41,0x41,0x40,0xDE,0x5D,0xAE,0x22,0x23,
0xB0,0x03,0x61,0xA3,0x96,0x17,0x7A,0x9C,0xB4,0x10,0xFF,0x61,0xF2,0x00,0x15,0xAD,
  },
  // Hash for test 1: "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
  {
0x24,0x8D,0x6A,0x61,0xD2,0x06,0x38,0xB8,0xE5,0xC0,0x26,0x93,0x0C,0x3E,0x60,0x39,
0xA3,0x3C,0xE4,0x59,0x64,0xFF,0x21,0x67,0xF6,0xEC,0xED,0xD4,0x19,0xDB,0x06,0xC1,
  },
  // Hash for test 2: 1,000,000 'a'
  {
0xCD,0xC7,0x6E,0x5C,0x99,0x14,0xFB,0x92,0x81,0xA1,0xC7,0xE2,0x84,0xD7,0x3E,0x67,
0xF1,0x80,0x9A,0x48,0xA4,0x97,0x20,0x0E,0x04,0x6D,0x39,0xCC,0xC7,0x11,0x2C,0xD0,
  },
};

void dump_array(unsigned char *ptr, unsigned int size)
{
  unsigned char *ptr_end;
  unsigned char *h;
  int i = 1, trunc = 0;

  if (size > 64) {
    trunc = size - 64;
    size = 64;
  }
  h = ptr;
  ptr_end = ptr + size;
  while (h < ptr_end) {
    printf("0x%02x ", *h);
    h++;
    if (i == 8) {
      if (h != ptr_end)
```

```
        printf("\n");
      i = 1;
    } else {
     ++i;
    }
  }
  printf("\n");
  if (trunc > 0)
    printf("... %d bytes not printed\n", trunc);
}


int old_api_sha256_test(void)
{
  ICA_ADAPTER_HANDLE adapter_handle;
  SHA256_CONTEXT Sha256Context;
  int rc = 0, i = 0;
  unsigned char input_data[1000000];
  unsigned int  output_hash_length = LENGTH_SHA256_HASH;
  unsigned char output_hash[LENGTH_SHA256_HASH];

  rc = icaOpenAdapter(0, &adapter_handle);
  if (rc != 0) {
    printf("icaOpenAdapter failed and returned %d (0x%x).\n", rc, rc);
    if (rc == ENODEV)
      printf("The usual cause of this on zSeries is that the CPACF instruction is not available.\n");
    return 2;
  }

  for (i = 0; i < NUM_FIPS_TESTS; i++) {
    // Test 2 is a special one, because we want to keep the size of the
    // executable down, so we build it special, instead of using a static
    if (i != 2)
      memcpy(input_data, FIPS_TEST_DATA[i], FIPS_TEST_DATA_SIZE[i]);
    else
      memset(input_data, 'a', FIPS_TEST_DATA_SIZE[i]);

    printf("\nOriginal data for test %d:\n", i);
    dump_array(input_data, FIPS_TEST_DATA_SIZE[i]);

    rc = icaSha256(adapter_handle,
                   SHA_MSG_PART_ONLY,
                   FIPS_TEST_DATA_SIZE[i],
                   input_data,
                   LENGTH_SHA256_CONTEXT,
                   &Sha256Context,
                   &output_hash_length,
                   output_hash);

    if (rc != 0) {
      printf("icaSha256 failed with errno %d (0x%x).\n", rc, rc);
      return 2;
    }

    if (output_hash_length != LENGTH_SHA256_HASH) {
      printf("icaSha256 returned an incorrect output data length, %u (0x%x).\n",
             output_hash_length, output_hash_length);
      return 2;
    }

    printf("\nOutput hash for test %d:\n", i);
    dump_array(output_hash, output_hash_length);
    if (memcmp(output_hash, FIPS_TEST_RESULT[i], LENGTH_SHA256_HASH) != 0) {
       printf("This does NOT match the known result.\n");
    } else {
       printf("Yes, it's what it should be.\n");
    }
```

```
}

// This test is the same as test 2, except that we use the SHA256_CONTEXT and
// break it into calls of 1024 bytes each.
printf("\nOriginal data for test 2(chunks = 1024) is calls of 1024 'a's at a time\n");
i = FIPS_TEST_DATA_SIZE[2];
while (i > 0) {
  unsigned int shaMessagePart;
  memset(input_data, 'a', 1024);

  if (i == FIPS_TEST_DATA_SIZE[2])
    shaMessagePart = SHA_MSG_PART_FIRST;
  else if (i <= 1024)
    shaMessagePart = SHA_MSG_PART_FINAL;
  else
    shaMessagePart = SHA_MSG_PART_MIDDLE;

  rc = icaSha256(adapter_handle,
             shaMessagePart,
             (i < 1024) ? i : 1024,
             input_data,
             LENGTH_SHA256_CONTEXT,
             &Sha256Context,
             &output_hash_length,
             output_hash);

  if (rc != 0) {
    printf("icaSha256 failed with errno %d (0x%x) on iteration %d.\n", rc, rc, i);
    return 2;
  }

  i -= 1024;
}

if (output_hash_length != LENGTH_SHA256_HASH) {
  printf("icaSha256 returned an incorrect output data length, %u (0x%x).\n",
         output_hash_length, output_hash_length);
  return 2;
}

printf("\nOutput hash for test 2(chunks = 1024):\n");
dump_array(output_hash, output_hash_length);
if (memcmp(output_hash, FIPS_TEST_RESULT[2], LENGTH_SHA256_HASH) != 0) {
   printf("This does NOT match the known result.\n");
} else {
   printf("Yes, it's what it should be.\n");
}

// This test is the same as test 2, except that we use the SHA256_CONTEXT and
// break it into calls of 64 bytes each.
printf("\nOriginal data for test 2(chunks = 64) is calls of 64 'a's at a time\n");
i = FIPS_TEST_DATA_SIZE[2];
while (i > 0) {
  unsigned int shaMessagePart;
  memset(input_data, 'a', 64);

  if (i == FIPS_TEST_DATA_SIZE[2])
    shaMessagePart = SHA_MSG_PART_FIRST;
  else if (i <= 64)
    shaMessagePart = SHA_MSG_PART_FINAL;
  else
    shaMessagePart = SHA_MSG_PART_MIDDLE;

  rc = icaSha256(adapter_handle,
             shaMessagePart,
             (i < 64) ? i : 64,
             input_data,
```

```
                LENGTH_SHA256_CONTEXT,
                &Sha256Context,
                &output_hash_length,
                output_hash);

    if (rc != 0) {
      printf("icaSha256 failed with errno %d (0x%x) on iteration %d.\n", rc, rc, i);
      return 2;
    }

    i -= 64;
  }

  if (output_hash_length != LENGTH_SHA256_HASH) {
    printf("icaSha256 returned an incorrect output data length, %u (0x%x).\n",
           output_hash_length, output_hash_length);
    return 2;
  }

  printf("\nOutput hash for test 2(chunks = 64):\n");
  dump_array(output_hash, output_hash_length);
  if (memcmp(output_hash, FIPS_TEST_RESULT[2], LENGTH_SHA256_HASH) != 0) {
     printf("This does NOT match the known result.\n");
  } else {
     printf("Yes, it's what it should be.\n");
  }

  printf("\nAll SHA256 tests completed successfully\n");

  icaCloseAdapter(adapter_handle);

  return 0;
}

int new_api_sha256_test(void)
{
 sha256_context_t sha256_context;
 int rc = 0, i = 0;
 unsigned char input_data[1000000];
 unsigned int  output_hash_length = LENGTH_SHA256_HASH;
 unsigned char output_hash[LENGTH_SHA256_HASH];

 for (i = 0; i < NUM_FIPS_TESTS; i++) {
  // Test 2 is a special one, because we want to keep the size of the
  // executable down, so we build it special, instead of using a static
  if (i != 2)
   memcpy(input_data, FIPS_TEST_DATA[i], FIPS_TEST_DATA_SIZE[i]);
  else
   memset(input_data, 'a', FIPS_TEST_DATA_SIZE[i]);

  printf("\nOriginal data for test %d:\n", i);
  dump_array(input_data, FIPS_TEST_DATA_SIZE[i]);

  rc = ica_sha256(SHA_MSG_PART_ONLY, FIPS_TEST_DATA_SIZE[i], input_data,
    &sha256_context, output_hash);

  if (rc != 0) {
   printf("icaSha256 failed with errno %d (0x%x).\n", rc, rc);
   return rc;
  }

  printf("\nOutput hash for test %d:\n", i);
  dump_array(output_hash, output_hash_length);
  if (memcmp(output_hash, FIPS_TEST_RESULT[i], LENGTH_SHA256_HASH) != 0)
   printf("This does NOT match the known result.\n");
  else
   printf("Yes, it's what it should be.\n");
```

```
}

// This test is the same as test 2, except that we use the SHA256_CONTEXT and
// break it into calls of 1024 bytes each.
printf("\nOriginal data for test 2(chunks = 1024) is calls of 1024"
       " 'a's at a time\n");
i = FIPS_TEST_DATA_SIZE[2];
while (i > 0) {
 unsigned int sha_message_part;
 memset(input_data, 'a', 1024);

 if (i == FIPS_TEST_DATA_SIZE[2])
  sha_message_part = SHA_MSG_PART_FIRST;
 else if (i <= 1024)
  sha_message_part = SHA_MSG_PART_FINAL;
 else
  sha_message_part = SHA_MSG_PART_MIDDLE;

 rc = ica_sha256(sha_message_part, (i < 1024) ? i : 1024,
   input_data, &sha256_context, output_hash);

 if (rc != 0) {
  printf("ica_sha256 failed with errno %d (0x%x) on"
         " iteration %d.\n", rc, rc, i);
  return rc;
 }
 i -= 1024;
}

printf("\nOutput hash for test 2(chunks = 1024):\n");
dump_array(output_hash, output_hash_length);
if (memcmp(output_hash, FIPS_TEST_RESULT[2], LENGTH_SHA256_HASH) != 0)
 printf("This does NOT match the known result.\n");
else
 printf("Yes, it's what it should be.\n");

// This test is the same as test 2, except that we use the
// SHA256_CONTEXT and break it into calls of 64 bytes each.
printf("\nOriginal data for test 2(chunks = 64) is calls of 64 'a's at"
       " a time\n");
i = FIPS_TEST_DATA_SIZE[2];
while (i > 0) {
 unsigned int sha_message_part;
 memset(input_data, 'a', 64);

 if (i == FIPS_TEST_DATA_SIZE[2])
  sha_message_part = SHA_MSG_PART_FIRST;
 else if (i <= 64)
  sha_message_part = SHA_MSG_PART_FINAL;
 else
  sha_message_part = SHA_MSG_PART_MIDDLE;

 rc = ica_sha256(sha_message_part, (i < 64) ? i : 64,
   input_data, &sha256_context, output_hash);

 if (rc != 0) {
  printf("ica_sha256 failed with errno %d (0x%x) on iteration"
         " %d.\n", rc, rc, i);
  return rc;
 }
 i -= 64;
}

printf("\nOutput hash for test 2(chunks = 64):\n");
dump_array(output_hash, output_hash_length);
if (memcmp(output_hash, FIPS_TEST_RESULT[2], LENGTH_SHA256_HASH) != 0)
 printf("This does NOT match the known result.\n");
```

```
 else
  printf("Yes, it's what it should be.\n");

 printf("\nAll SHA256 tests completed successfully\n");

 return 0;
}

int main(int argc, char **argv)
{
 int rc = 0;
 rc = old_api_sha256_test();
 if (rc) {
  printf("old_api_sha256_test: returned rc = %i\n", rc);
  return rc;
 }

 rc = new_api_sha256_test();
 if (rc) {
  printf("new_api_sha256_test: returned rc = %i\n", rc);
  return rc;
 }

 return rc;
}
```

## Pseudo random number generation example

This example uses the libica version 1 API. Examples for using the libica version 2.4 API for random number generation are located in other examples, such as the DES with CTR mode example.

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2010, 2011 */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include "ica_api.h"

unsigned char R[512];

extern int errno;

void dump_array(unsigned char *ptr, unsigned int size)
{
   unsigned char *ptr_end;
   unsigned char *h;
   int i = 1;


   h = ptr;
   ptr_end = ptr + size;
   while (h < (unsigned char *)ptr_end) {
      printf("0x%02x ",(unsigned char ) *h);
      h++;
      if (i == 8) {
         printf("\n");
         i = 1;
      } else {
         ++i;
      }
```

```
   }
   printf("\n");
}

int main(int ac, char **av)
{
   int rc;
   ICA_ADAPTER_HANDLE adapter_handle;

   rc = icaOpenAdapter(0, &adapter_handle);
   if (rc != 0) {
      printf("icaOpenAdapter failed and returned %d (0x%x).\n", rc, rc);
   }

   rc = icaRandomNumberGenerate(adapter_handle, sizeof R, R);
   if (rc != 0) {
      printf("icaRandomNumberGenerate failed and returned %d (0x%x).\n", rc, rc);
#ifdef __s390__
      if (rc == ENODEV)
        printf("The usual cause of this on zSeries is that the CPACF instruction is not available.\n");
#endif
   }
   else {
      printf("\nHere it is:\n");
   }

   dump_array(R, sizeof R);

   if (!rc) {
      printf("\nWell, does it look random?\n\n");
   }

   icaCloseAdapter(adapter_handle);

   return 0;
}
```

## Key generation example

This example uses the various key generation APIs, as well as those to open and close an adapter, and random number generation.

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* (C) COPYRIGHT International Business Machines Corp. 2001, 2009          */
#include <sys/errno.h>
#include <fcntl.h>
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "ica_api.h"

#define KEY_BYTES ((key_bits + 7) / 8)
#define KEY_BYTES_MAX 256

extern int errno;

void dump_array(char *ptr, int size)
{
 char *ptr_end;
 char *h;
 int i = 1;
```

```
 h = ptr;
 ptr_end = ptr + size;
 while (h < ptr_end) {
  printf("0x%02x ",(unsigned char ) *h);
  h++;
  if (i == 8) {
   printf("\n");
   i = 1;
  } else {
   ++i;
  }
 }
 printf("\n");
}


int main(int argc, char **argv)
{
 ICA_ADAPTER_HANDLE adapter_handle;
 ICA_KEY_RSA_CRT crtkey;
 ICA_KEY_RSA_MODEXPO wockey, wockey2;
 unsigned char decrypted[KEY_BYTES_MAX], encrypted[KEY_BYTES_MAX],
        original[KEY_BYTES_MAX];
 int rc;
 unsigned int length, length2;
 unsigned int exponent_type = RSA_PUBLIC_FIXED, key_bits = 1024;

 length = sizeof wockey;
 length2 = sizeof wockey2;
 bzero(&wockey, sizeof wockey);
 bzero(&wockey2, sizeof wockey2);

 rc = icaOpenAdapter(0, &adapter_handle);
 if (rc != 0) {
  printf("icaOpenAdapter failed and returned %d (0x%x).\n", rc,
         rc);
 }
 exponent_type = RSA_PUBLIC_FIXED;
 printf("a fixed exponent . . .\n");
 rc = icaRandomNumberGenerate(adapter_handle, KEY_BYTES,
         wockey.keyRecord);
 if (rc != 0) {
        printf("icaRandomNumberGenerate failed and returned %d (0x%x)"
         ".\n", rc, rc);
  return -1;
 }
 wockey.nLength = KEY_BYTES / 2;
 wockey.expLength = sizeof(unsigned long);
  wockey.expOffset = SZ_HEADER_MODEXPO;
 wockey.keyRecord[wockey.expLength - 1] |= 1;
 if (argc > 1) {
  key_bits = atoi(argv[1]);
  if (key_bits > KEY_BYTES_MAX * 8) {
   printf("The maximum key length is %d bits.",
          KEY_BYTES_MAX * 8);
   exit(0);
  }
  wockey.modulusBitLength = key_bits;
  printf("Using %u-bit keys and ", key_bits);
  if (argc > 2) {
   switch (argv[2][0]) {
   case '3':
    exponent_type = RSA_PUBLIC_3;
    printf("exponent 3 . . .\n");
    wockey.expLength = 1;
    break;
```

```
   case '6':
    exponent_type = RSA_PUBLIC_65537;
    printf("exponent 65537 . . .\n");
    wockey.expLength = 3;
    break;
   case 'R':
   case 'r':
    exponent_type = RSA_PUBLIC_RANDOM;
    printf("a random exponent . . .\n");
    break;
   default:
    break;
   }
  }
 }

 rc = icaRandomNumberGenerate(adapter_handle, sizeof(original),
         original);
 if (rc != 0) {
  printf("icaRandomNumberGenerate failed and returned %d (0x%x)"
         ".\n", rc, rc);
  return rc;
 }
 original[0] = 0;

 rc = icaRsaKeyGenerateModExpo(adapter_handle, key_bits, exponent_type,
          &length, &wockey, &length2, &wockey2);
 if (rc != 0) {
  printf("icaRsaKeyGenerateModExpo failed and returned %d (0x%x)"
         ".\n", rc, rc);
  return rc;
 }

 printf("Public key:\n");
 dump_array((char *) wockey.keyRecord, 2 * KEY_BYTES);
 printf("Private key:\n");
 dump_array((char *) wockey2.keyRecord, 2 * KEY_BYTES);

 bzero(encrypted, KEY_BYTES);
 length = KEY_BYTES;
 printf("encrypt \n");
 rc = icaRsaModExpo(adapter_handle, KEY_BYTES, original, &wockey,
      &length, encrypted);
 if (rc != 0) {
  printf("icaRsaModExpo failed and returned %d (0x%x).\n", rc, rc);
  return rc;
 }
 bzero(decrypted, KEY_BYTES);
 length = KEY_BYTES;
 printf("decrypt \n");
 rc = icaRsaModExpo(adapter_handle, KEY_BYTES, encrypted, &wockey2,
      &length, decrypted);
 if (rc != 0) {
  printf("icaRsaModExpo failed and returned %d (0x%x).\n", rc,
         rc);
  return rc;
 }

 printf("Original:\n");
 dump_array((char *) original, KEY_BYTES);
 printf("Result of encrypt:\n");
 dump_array((char *) encrypted, KEY_BYTES);
 printf("Result of decrypt:\n");
 dump_array((char *) decrypted, KEY_BYTES);
 if (memcmp(original, decrypted, KEY_BYTES) != 0) {
  printf("This does not match the original plaintext.  Failure!\n");
  icaCloseAdapter(adapter_handle);
```

```
 return errno ? errno : -1;
} else {
 printf("Success!  The key pair checks out.\n");
 if (memcmp(original, encrypted, KEY_BYTES) == 0) {
  printf("But the ciphertext equals the plaintext."
         "That can't be good.\n");
  return -1;
 }
}
fflush(stdout);

length = sizeof wockey;
length2 = sizeof crtkey;
bzero(&wockey, sizeof wockey);
wockey.expLength = sizeof(unsigned long);
if (exponent_type == RSA_PUBLIC_FIXED) {
 wockey.keyType = KEYTYPE_MODEXPO;
 wockey.keyLength = sizeof wockey;
 wockey.modulusBitLength = key_bits;
 wockey.nLength = KEY_BYTES;
 wockey.expOffset = SZ_HEADER_MODEXPO;
 wockey.expLength = sizeof (unsigned long);
 wockey.nOffset = KEY_BYTES + wockey.expOffset;
 rc = icaRandomNumberGenerate(adapter_handle, KEY_BYTES,
         wockey.keyRecord);
 if (rc != 0) {
  printf("icaRandomNumberGenerate failed and returned %d"
         "(0x%x).\n", rc, rc);
  return rc;
 }
 wockey.keyRecord[wockey.expLength - 1] |= 1;
}
rc = icaRsaKeyGenerateCrt(adapter_handle, key_bits, exponent_type,
     &length, &wockey, &length2, &crtkey);
printf("wockey.modulusBitLength = %i, crtkey.modulusBitLength = %i"
       " \n", wockey.modulusBitLength, crtkey.modulusBitLength);
if (rc != 0) {
 printf("icaRsaKeyGenerateCrt failed and returned %d (0x%x)"
        ".\n", rc, rc);
 return rc;
}

printf("Public key:\n");
dump_array((char *) wockey.keyRecord, 2 * KEY_BYTES);
printf("Private key:\n");
dump_array((char *) crtkey.keyRecord, 5 * KEY_BYTES / 2 + 24);

bzero(encrypted, KEY_BYTES);
length = KEY_BYTES;
rc = icaRsaModExpo(adapter_handle, KEY_BYTES, original, &wockey,
     &length, encrypted);
if (rc != 0)
 printf("icaRsaModExpo failed and returned %d (0x%x).\n", rc, rc);

bzero(decrypted, KEY_BYTES);
length = KEY_BYTES;
rc = icaRsaCrt(adapter_handle, KEY_BYTES, encrypted, &crtkey, &length,
        decrypted);
if (rc != 0)
 printf("icaRsaCrt failed and returned %d (0x%x).\n", rc, rc);

printf("Original:\n");
dump_array((char *) original, KEY_BYTES);
printf("Result of encrypt:\n");
dump_array((char *) encrypted, KEY_BYTES);
printf("Result of decrypt:\n");
dump_array((char *) decrypted, KEY_BYTES);
```

```
if (memcmp(original, decrypted, KEY_BYTES) != 0) {
 printf("This does not match the original plaintext.  Failure!\n");
 icaCloseAdapter(adapter_handle);
 return errno ? errno : -1;
} else {
 printf("Success!  The key pair checks out.\n");
 if (memcmp(original, encrypted, KEY_BYTES) == 0) {
  printf("But the ciphertext equals the plaintext.  That can't be good.\n");
  return -1;
 }
}
fflush(stdout);

printf("TEST NEW API - MOD_EXPO\n");
rc = ica_close_adapter(adapter_handle);
printf("ica_close_adapter rc = %i\n", rc);

rc = ica_open_adapter(&adapter_handle);
if (rc)
 printf("Adapter not open\n");
else
 printf("Adapter open\n");

ica_rsa_key_mod_expo_t modexpo_public_key;
unsigned char modexpo_public_n[KEY_BYTES];
bzero(modexpo_public_n, KEY_BYTES);
unsigned char modexpo_public_e[KEY_BYTES];
bzero(modexpo_public_e, KEY_BYTES);
modexpo_public_key.modulus = modexpo_public_n;
modexpo_public_key.exponent = modexpo_public_e;
modexpo_public_key.key_length = KEY_BYTES;
if (exponent_type == RSA_PUBLIC_65537)
 *(unsigned long*)((unsigned char *)modexpo_public_key.exponent +
   modexpo_public_key.key_length -
   sizeof(unsigned long)) = 65537;
if (exponent_type == RSA_PUBLIC_3)
 *(unsigned long*)((unsigned char *)modexpo_public_key.exponent +
   modexpo_public_key.key_length -
   sizeof(unsigned long)) = 3;

ica_rsa_key_mod_expo_t modexpo_private_key;
unsigned char modexpo_private_n[KEY_BYTES];
bzero(modexpo_private_n, KEY_BYTES);
unsigned char modexpo_private_e[KEY_BYTES];
bzero(modexpo_private_e, KEY_BYTES);
modexpo_private_key.modulus = modexpo_private_n;
modexpo_private_key.exponent = modexpo_private_e;
modexpo_private_key.key_length = KEY_BYTES;

rc = ica_rsa_key_generate_mod_expo(adapter_handle,
       key_bits,
       &modexpo_public_key,
       &modexpo_private_key);
if (rc)
 printf("ica_rsa_key_generate_mod_expo rc = %i\n",rc);

printf("Public key:\n");
dump_array((char *) (char *)modexpo_public_key.exponent, KEY_BYTES);
dump_array((char *) (char *)modexpo_public_key.modulus, KEY_BYTES);
printf("Private key:\n");
dump_array((char *) (char *)modexpo_private_key.exponent, KEY_BYTES);
dump_array((char *) (char *)modexpo_private_key.modulus, KEY_BYTES);

bzero(encrypted, KEY_BYTES);
length = KEY_BYTES;
printf("encrypt \n");
rc = ica_rsa_mod_expo(adapter_handle, original, &modexpo_public_key,
```

```
        encrypted);

if (rc != 0) {
 printf("ica_rsa_mod_expo failed and returned %d (0x%x).\n", rc,
        rc);
 return rc;
}
bzero(decrypted, KEY_BYTES);
length = KEY_BYTES;
printf("decrypt \n");
rc = ica_rsa_mod_expo(adapter_handle, encrypted, &modexpo_private_key,
        decrypted);
if (rc != 0) {
 printf("ica_rsa_mod_expo failed and returned %d (0x%x).\n", rc,
        rc);
 return rc;
}

printf("Original:\n");
dump_array((char *) original, KEY_BYTES);
printf("Result of encrypt:\n");
dump_array((char *) encrypted, KEY_BYTES);
printf("Result of decrypt:\n");
dump_array((char *) decrypted, KEY_BYTES);
if (memcmp(original, decrypted, KEY_BYTES) != 0) {
 printf("This does not match the original plaintext.  Failure!\n");
 return -1;
} else {
 printf("Success!  The key pair checks out.\n");
 if (memcmp(original, encrypted, KEY_BYTES) == 0) {
  printf("But the ciphertext equals the plaintext.  That can't be good.\n");
  return -1;
 }
}
fflush(stdout);

printf("TEST NEW API - CRT\n");
ica_rsa_key_mod_expo_t public_key;
ica_rsa_key_crt_t private_key;

unsigned char public_n[KEY_BYTES];
bzero(public_n, KEY_BYTES);
unsigned char public_e[KEY_BYTES];
bzero(public_e, KEY_BYTES);
public_key.modulus = public_n;
public_key.exponent = public_e;
public_key.key_length = KEY_BYTES;

unsigned char private_p[(key_bits + 7) / (8 * 2) + 8];
bzero(private_p, KEY_BYTES + 1);
unsigned char private_q[(key_bits + 7) / (8 * 2)];
bzero(private_q, KEY_BYTES);
unsigned char private_dp[(key_bits + 7) / (8 * 2) + 8];
bzero(private_dp, KEY_BYTES + 1);
unsigned char private_dq[(key_bits + 7) / (8 * 2)];
bzero(private_dq, KEY_BYTES);
unsigned char private_qInverse[(key_bits + 7) / (8 * 2) + 8];
bzero(private_qInverse, KEY_BYTES + 1);
private_key.p = private_p;
private_key.q = private_q;
private_key.dp = private_dp;
private_key.dq = private_dq;
private_key.qInverse = private_qInverse;
private_key.key_length = (key_bits + 7) / 8;

if (exponent_type == RSA_PUBLIC_65537)
               *(unsigned long*)((unsigned char *)public_key.exponent +
```

```
                                public_key.key_length -
                                sizeof(unsigned long)) = 65537;
        if (exponent_type == RSA_PUBLIC_3)
                *(unsigned long*)((unsigned char *)public_key.exponent +
                                public_key.key_length -
                                sizeof(unsigned long)) = 3;

 rc = ica_rsa_key_generate_crt(adapter_handle, key_bits, &public_key,
          &private_key);
 if (rc != 0) {
  printf("ica_rsa_key_generate_crt failed and returned %d (0x%x)"
         ".\n", rc, rc);
  return rc;
 }

 printf("Public key:\n");
 dump_array((char *) (char *)&public_key, 2 * KEY_BYTES);
 printf("Private key:\n");
 dump_array((char *) (char *)&private_key, 5 * KEY_BYTES / 2 + 24);

 bzero(encrypted, KEY_BYTES);
 length = KEY_BYTES;
 rc = ica_rsa_mod_expo(adapter_handle, original, &public_key, encrypted);
 if (rc != 0) {
  printf("ica_rsa_mod_expo failed and returned %d (0x%x).\n",
         rc, rc);
  return rc;
 }
 bzero(decrypted, KEY_BYTES);
 length = KEY_BYTES;
 rc = ica_rsa_crt(adapter_handle, encrypted, &private_key, decrypted);
 if (rc != 0) {
  printf("icaRsaCrt failed and returned %d (0x%x).\n", rc, rc);
  return rc;
 }

 printf("Original:\n");
 dump_array((char *) original, KEY_BYTES);
 printf("Result of encrypt:\n");
 dump_array((char *) encrypted, KEY_BYTES);
 printf("Result of decrypt:\n");
 dump_array((char *) decrypted, KEY_BYTES);
 if (memcmp(original, decrypted, KEY_BYTES) != 0) {
  printf("This does not match the original plaintext."
         "Failure!\n");
 } else {
  printf("Success!  The key pair checks out.\n");
  if (memcmp(original, encrypted, KEY_BYTES) == 0) {
   printf("But the ciphertext equals the plaintext."
          "That can't be good.\n");
  }
 }
 fflush(stdout);
 ica_close_adapter(adapter_handle);
 return 0;
}
```

## RSA example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2001, 2009, 2011 */
```

```
#include <fcntl.h>
#include <memory.h>
#include <sys/errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "ica_api.h"

unsigned char pubkey1024[] =
                  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x03 };

unsigned char modulus1024[] =
                  { 0xec, 0x51, 0xab, 0xa1, 0xf8, 0x40, 0x2c, 0x08,
                    0x2e, 0x24, 0x52, 0x2e, 0x3c, 0x51, 0x6d, 0x98,
                    0xad, 0xee, 0xc7, 0x7d, 0x00, 0xaf, 0xe1, 0xa8,
                    0x61, 0xda, 0x32, 0x97, 0xb4, 0x32, 0x97, 0xe3,
                    0x52, 0xda, 0x28, 0x45, 0x55, 0xc6, 0xb2, 0x46,
                    0x65, 0x1b, 0x02, 0xcb, 0xbe, 0xf4, 0x2c, 0x6b,
                    0x2a, 0x5f, 0xe1, 0xdf, 0xe9, 0xe3, 0xbc, 0x47,
                    0xb7, 0x38, 0xb5, 0xa2, 0x78, 0x9d, 0x15, 0xe2,
                    0x59, 0x81, 0x77, 0x6b, 0x6b, 0x2e, 0xa9, 0xdb,
                    0x13, 0x26, 0x9c, 0xca, 0x5e, 0x0a, 0x1f, 0x3c,
                    0x50, 0x9d, 0xd6, 0x79, 0x59, 0x99, 0x50, 0xe5,
                    0x68, 0x1a, 0x98, 0xca, 0x11, 0xce, 0x37, 0x63,
                    0x58, 0x22, 0x40, 0x19, 0x29, 0x72, 0x4c, 0x41,
                    0x89, 0x0b, 0x56, 0x9e, 0x3e, 0xd5, 0x6d, 0x75,
                    0x9e, 0x3f, 0x8a, 0x50, 0xf1, 0x0a, 0x59, 0x4a,
                    0xc3, 0x59, 0x4b, 0xf6, 0xbb, 0xc9, 0xa5, 0x93 };



unsigned char Bp[] =
                  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0xa7, 0xcf, 0xa2, 0x18, 0x2c, 0xa9, 0xb4, 0xb9,
                    0xf5, 0x9e, 0xc9, 0x04, 0x16, 0xd9, 0xa6, 0x8b,
                    0x90, 0x4a, 0x19, 0x6d, 0x64, 0xb7, 0x17, 0x67,
                    0x53, 0xfa, 0x4e, 0x8d, 0xde, 0xa6, 0x94, 0x32,
                    0x5d, 0xcf, 0x58, 0x3e, 0x90, 0xbb, 0x30, 0x19,
                    0x96, 0x38, 0x95, 0xb6, 0xca, 0x2f, 0xfa, 0x22,
                    0x81, 0x65, 0x3b, 0x3c, 0x95, 0x9e, 0x79, 0x75,
                    0xe4, 0x93, 0x50, 0xf1, 0x88, 0x6b, 0xc1, 0x87 };

unsigned char Bq[] =
                  { 0xa0, 0x3a, 0x18, 0xa4, 0x1c, 0x3c, 0x49, 0x09,
                    0xd0, 0x84, 0x4a, 0x8c, 0x7c, 0xce, 0xdf, 0x9e,
                    0x90, 0x7d, 0xc4, 0xca, 0x7e, 0x2d, 0x3d, 0xbc,
                    0x09, 0x71, 0x79, 0xd0, 0xc0, 0xae, 0xa6, 0xc1,
                    0x9d, 0xf0, 0x16, 0xf0, 0x1f, 0x68, 0x9a, 0xc5,
                    0x2b, 0xf3, 0x5a, 0xfc, 0x2c, 0xf5, 0xa7, 0xec,
                    0xd9, 0xa2, 0xac, 0x49, 0xcc, 0x76, 0x9c, 0xd8,
                    0x4c, 0x59, 0x5e, 0x38, 0xd2, 0x85, 0xd3, 0x3b };
```

```
unsigned char Np[] =
                  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0xfb, 0xb7, 0x73, 0x24, 0x42, 0xfe, 0x8f, 0x16,
                    0xf0, 0x6e, 0x2d, 0x86, 0x22, 0x46, 0x79, 0xd1,
                    0x58, 0x6f, 0x26, 0x24, 0x17, 0x12, 0xa3, 0x1a,
                    0xfd, 0xf7, 0x75, 0xd4, 0xcd, 0xf9, 0xde, 0x4b,
                    0x8c, 0xb7, 0x04, 0x5d, 0xd9, 0x18, 0xc8, 0x26,
                    0x61, 0x54, 0xe0, 0x92, 0x2f, 0x47, 0xf7, 0x33,
                    0xc2, 0x17, 0xd8, 0xda, 0xe0, 0x6d, 0xb6, 0x30,
                    0xd6, 0xdc, 0xf9, 0x6a, 0x4c, 0xa1, 0xa2, 0x4b };

unsigned char Nq[] =
                  { 0xf0, 0x57, 0x24, 0xf6, 0x2a, 0x5a, 0x6d, 0x8e,
                    0xb8, 0xc6, 0x6f, 0xd2, 0xbb, 0x36, 0x4f, 0x6d,
                    0xd8, 0xbc, 0xa7, 0x2f, 0xbd, 0x43, 0xdc, 0x9a,
                    0x0e, 0x2a, 0x36, 0xb9, 0x21, 0x05, 0xfa, 0x22,
                    0x6c, 0xe8, 0x22, 0x68, 0x2f, 0x1c, 0xe8, 0x27,
                    0xc1, 0xed, 0x08, 0x7a, 0x43, 0x70, 0x7b, 0xe3,
                    0x46, 0x74, 0x02, 0x6e, 0xb2, 0xb1, 0xeb, 0x44,
                    0x72, 0x86, 0x0d, 0x55, 0x3b, 0xc8, 0xbc, 0xd9 };

unsigned char U[] =
                  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                    0x83, 0xf1, 0xca, 0x06, 0x58, 0x4a, 0x04, 0x5e,
                    0x96, 0xb5, 0x30, 0x32, 0x40, 0x36, 0x48, 0xb9,
                    0x02, 0x0c, 0xe3, 0x37, 0xb7, 0x51, 0xbc, 0x22,
                    0x26, 0x5d, 0x74, 0x03, 0x47, 0xd3, 0x33, 0x20,
                    0x8e, 0x75, 0x62, 0xf2, 0x9d, 0x4e, 0xc8, 0x7d,
                    0x5d, 0x8e, 0xb6, 0xd9, 0x69, 0x4a, 0x9a, 0xe1,
                    0x36, 0x6e, 0x1c, 0xbe, 0x8a, 0x14, 0xb1, 0x85,
                    0x39, 0x74, 0x7c, 0x25, 0xd8, 0xa4, 0x4f, 0xde };



unsigned char R[128];

unsigned char A[] =
                  { 0x00, 0x02, 0x08, 0x68, 0x30, 0x9a, 0x32, 0x08,
                    0x57, 0xb0, 0x28, 0xaa, 0x76, 0x30, 0x3d, 0x84,
                    0x5f, 0x92, 0x0d, 0x8e, 0x34, 0xe0, 0xd5, 0xcc,
                    0x36, 0x97, 0xed, 0x00, 0x00, 0x01, 0x02, 0x03,
                    0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
                    0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
                    0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
                    0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23,
                    0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b,
                    0x2c, 0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33,
                    0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3a, 0x3b,
                    0x3c, 0x3d, 0x3e, 0x3f, 0x40, 0x41, 0x42, 0x43,
                    0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x4b,
                    0x4c, 0x4d, 0x4e, 0x4f, 0x50, 0x51, 0x52, 0x53,
                    0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b,
                    0x5c, 0x5d, 0x5e, 0x5f, 0x60, 0x61, 0x62, 0x63 };

unsigned char Ciphertext[] =
                  { 0xb2, 0xb2, 0x82, 0xd7, 0x2c, 0x6f, 0x53, 0x29,
                    0xee, 0x4c, 0xd1, 0x77, 0xb7, 0x13, 0xf3, 0x1c,
                    0x51, 0x60, 0xd8, 0xa9, 0x4e, 0x52, 0x72, 0x43,
                    0x29, 0xfa, 0x51, 0xaa, 0xd8, 0xbc, 0x31, 0x21,
                    0xe0, 0xac, 0x9b, 0x4e, 0x0, 0x94, 0xac, 0x91,
                    0x7f, 0x1e, 0xfd, 0xfb, 0x1c, 0xfa, 0xa8, 0xe8,
                    0x56, 0x5a, 0x1, 0x17, 0xf1, 0x5f, 0x1, 0xba,
                    0xcd, 0x77, 0xa1, 0x8c, 0x74, 0x8a, 0xef, 0xfa,
                    0x64, 0x58, 0x79, 0x13, 0xaa, 0x54, 0x13, 0x2b,
                    0xaa, 0xe7, 0xc3, 0x50, 0x3b, 0x69, 0x3b, 0xb,
                    0x9a, 0xa9, 0x9d, 0x15, 0x8a, 0x6, 0x45, 0x71,
```

```
                    0x40, 0x7a, 0x80, 0x85, 0x4a, 0xbe, 0x68, 0x48,
                    0x6c, 0xe6, 0xdd, 0x96, 0xb0, 0xdc, 0xf4, 0x23,
                    0xa8, 0xea, 0x21, 0x9f, 0xbc, 0x6b, 0x15, 0xa4,
                    0x87, 0x6e, 0x93, 0x56, 0xae, 0xa7, 0x17, 0x4e,
                    0xd7, 0x14, 0xe4, 0x69, 0x4, 0xd5, 0x2e, 0x62 };

extern int errno;

void dump_array(unsigned char *ptr, unsigned int size)
{
   unsigned char *ptr_end;
   unsigned char *h;
   int i = 1;


   h = ptr;
   ptr_end = ptr + size;
   while (h < (unsigned char *)ptr_end) {
      printf("0x%02x ",(unsigned char ) *h);
      h++;
      if (i == 8) {
         printf("\n");
         i = 1;
      } else {
         ++i;
      }
   }
   printf("\n");
}

int main()
{
   ICA_ADAPTER_HANDLE adapter_handle;
   ICA_KEY_RSA_CRT icakey;
   ICA_KEY_RSA_MODEXPO wockey;
   caddr_t  key;
   caddr_t  my_result;
   caddr_t  my_result2;
   /* icaRsaModExpo_t rsawoc; */
   int   i;
   unsigned int  length;

   i = icaOpenAdapter(0, &adapter_handle);
   if (i != 0) {
      printf("icaOpenAdapter failed and returned %d (0x%x), errno=%d\n", i, i, errno);
      return i;
   }

    /*
     * encrypt with public key
     */

    printf("modulus size = %ld\n", (long)sizeof(modulus1024));
    bzero(&wockey, sizeof(wockey));
    wockey.keyType = KEYTYPE_MODEXPO;
    wockey.keyLength = sizeof(ICA_KEY_RSA_MODEXPO);
    wockey.modulusBitLength = sizeof(modulus1024) * 8;
    wockey.nLength = sizeof(modulus1024);
    wockey.expLength = sizeof(pubkey1024);

    key = (caddr_t)wockey.keyRecord;

    bcopy(&pubkey1024, key, sizeof(pubkey1024));
    wockey.expOffset = key - (char *) &wockey;
    key += sizeof(pubkey1024);
    bcopy(&modulus1024, key, sizeof(modulus1024));
    wockey.nOffset = key - (char *) &wockey;
```

```
    my_result = (caddr_t) malloc(sizeof(A));
    bzero(my_result, sizeof(A));
    length = sizeof(A);

 printf("wockey.modulusBitLength = %i\n", wockey.modulusBitLength);
    if ((i = icaRsaModExpo(adapter_handle, sizeof(A), A,
                          &wockey, &length, (unsigned char *)my_result)) != 0) {
      printf("icaRsaModExpo failed and returned %d (0x%x).\n", i, i);
    }

    printf("\n\n\n\n\n result of encrypt with public key\n");
    dump_array((unsigned char *)my_result,sizeof(A));
    printf("Ciphertext \n");
    dump_array(Ciphertext,sizeof(A));
    if (memcmp(my_result,Ciphertext,sizeof(A))){
       printf("Ciphertext mismatch\n");
       return 0;
    } else {
       printf("ENCRYPT WORKED\n");
    }

    bzero(&icakey, sizeof(icakey));

     /* Card level CRT operation */
    icakey.keyType = KEYTYPE_PKCSCRT;
    icakey.keyLength = sizeof(ICA_KEY_RSA_CRT);
    icakey.modulusBitLength = sizeof(modulus1024)*8;


    my_result2 = (caddr_t)malloc(sizeof(A));
    bzero(my_result2,sizeof(A));

    key = (caddr_t)icakey.keyRecord;
    /*
     * Bp is copied into the key */
    bcopy(Bp,key,sizeof(Bp));
    icakey.dpLength = sizeof(Bp);
    icakey.dpOffset =  key - (char *)&icakey;
    key += sizeof(Bp);
    /*
     * Bq is copied into the key */
    bcopy(Bq,key,sizeof(Bq));
    icakey.dqLength = sizeof(Bq);
    icakey.dqOffset =  key - (char *)&icakey;
    key += sizeof(Bq);
    /*
     * Np is copied into the key */
    bcopy(Np,key,sizeof(Np));
    icakey.pLength = sizeof(Np);
    icakey.pOffset =  key - (char *)&icakey;
    key += sizeof(Np);
    /*
     * Nq is copied into the key */
    bcopy(Nq,key,sizeof(Nq));
    icakey.qLength = sizeof(Nq);
    icakey.qOffset =  key - (char *)&icakey;
    key += sizeof(Nq);
    /*
     * U is copied into the key */
    bcopy(U,key,sizeof(U));
    icakey.qInvLength = sizeof(U);
    icakey.qInvOffset =  key - (char *)&icakey;
    key += sizeof(U);

/*    printf("size of Bp=%d\n",sizeof(Bp));
    printf("size of Bq=%d\n",sizeof(Bq));
```

```
    printf("size of Np=%d\n",sizeof(Np));
    printf("size of Nq=%d\n",sizeof(Nq));
    printf("size of U=%d\n",sizeof(U));
    printf("size of R=%d\n",sizeof(R));

    printf("icakey  private Key record\n");
    dump_array(&icakey,sizeof(ICA_KEY_RSA_CRT)); */


    length = sizeof(Ciphertext);
 icakey.modulusBitLength = length * 8;
 icakey.keyLength = length;
    if ((i = icaRsaCrt(adapter_handle, sizeof(Ciphertext), Ciphertext,
                       &icakey, &length, (unsigned char *)my_result2)) != 0) {
      printf("icaRsaCrt failed and returned %d (0x%x).\n", i, i);
    }

    printf("Result of decrypt\n");
    dump_array((unsigned char *)my_result2, sizeof(A));
    printf("original data\n");
    dump_array(A, sizeof(A));
    if( memcmp(A,my_result2,sizeof(A)) != 0) {
      printf("Results do not match.  Failure!\n");
      return -1;
    } else {
      printf("Results match!\n");
    }

   icaCloseAdapter(adapter_handle);

   return 0;
}
```

## DES with CTR mode example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2010, 2011 */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

#define NR_RANDOM_TESTS 100

void dump_array(unsigned char *ptr, unsigned int size)
{
 unsigned char *ptr_end;
 unsigned char *h;
 int i = 1;

 h = ptr;
 ptr_end = ptr + size;
 while (h < (unsigned char *)ptr_end) {
  printf("0x%02x ",(unsigned char ) *h);
  h++;
  if (i == 8) {
   printf("\n");
   i = 1;
  } else {
```

```
   ++i;
  }
 }
 printf("\n");
}

void dump_ctr_data(unsigned char *iv, unsigned int iv_length,
                   unsigned char *key, unsigned int key_length,
                   unsigned char *input_data, unsigned int data_length,
                   unsigned char *output_data)
{
 printf("IV \n");
 dump_array(iv, iv_length);
 printf("Key \n");
 dump_array(key, key_length);
 printf("Input Data\n");
 dump_array(input_data, data_length);
 printf("Output Data\n");
 dump_array(output_data, data_length);
}

int random_des_ctr(int iteration, int silent, unsigned int data_length, unsigned int iv_length)
{
 unsigned int key_length = sizeof(ica_des_key_single_t);
 if (data_length % sizeof(ica_des_vector_t))
  iv_length = sizeof(ica_des_vector_t);

 printf("Test Parameters for iteration = %i\n", iteration);
 printf("key length = %i, data length = %i, iv length = %i\n",
        key_length, data_length, iv_length);

 unsigned char iv[iv_length];
 unsigned char tmp_iv[iv_length];
 unsigned char key[key_length];
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];

 int rc = 0;
 rc = ica_random_number_generate(data_length, input_data);
 if (rc) {
  printf("random number generate returned rc = %i, errno = %i\n", rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(iv_length, iv);
 if (rc) {
  printf("random number generate returned rc = %i, errno = %i\n", rc, errno);
  return rc;
 }

 rc = ica_random_number_generate(key_length, key);
 if (rc) {
  printf("random number generate returned rc = %i, errno = %i\n", rc, errno);
  return rc;
 }
 memcpy(tmp_iv, iv, iv_length);

 rc = ica_des_ctr(input_data, encrypt, data_length, key, tmp_iv,
    32,1);
 if (rc) {
  printf("ica_des_ctr encrypt failed with rc = %i\n", rc);
  dump_ctr_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
  return rc;
 }
 if (!silent && !rc) {
  printf("Encrypt:\n");
```

```
  dump_ctr_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }

 memcpy(tmp_iv, iv, iv_length);
 rc = ica_des_ctr(encrypt, decrypt, data_length, key, tmp_iv,
    32, 0);
 if (rc) {
  printf("ica_des_ctr decrypt failed with rc = %i\n", rc);
  dump_ctr_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
  return rc;
 }


 if (!silent && !rc) {
  printf("Decrypt:\n");
  dump_ctr_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
 }
 return rc;
}

int main(int argc, char **argv)
{
 unsigned int silent = 0;
 unsigned int endless = 0;
 if (argc > 1) {
  if (strstr(argv[1], "silent"))
   silent = 1;
  if (strstr(argv[1], "endless"))
   endless = 1;
 }
 int rc = 0;
 int error_count = 0;
 int i = 0;
 unsigned int data_length = sizeof(ica_des_key_single_t);
 unsigned int iv_length = sizeof(ica_des_key_single_t);

 if (endless) {
  silent = 1;
  while (1) {
   printf("i = %i\n",i);
   rc = random_des_ctr(i, silent, 320, 320);
   if (rc) {
    printf("kat_des_ctr failed with rc = %i\n",
     rc);
    return rc;
   } else
    printf("kat_des_ctr finished successfuly\n");
   i++;
  }
 } else {
  for (i = 1; i < NR_RANDOM_TESTS; i++) {
   rc = random_des_ctr(i, silent, data_length, iv_length);
                if (rc) {
    printf("random_des_ctr failed with rc = %i\n",
           rc);
```

```
    error_count++;
   } else
    printf("random_des_ctr finished "
     "successfuly\n");
   if (!(data_length % sizeof(ica_des_key_single_t))) {
          /* Always when the full block size is reached use a
    * counter with the same size as the data */
           rc = random_des_ctr(i, silent,
          data_length, data_length);
           if (rc) {
                   printf("random_des_ctr failed with "
            "rc = %i\n", rc);
                   error_count++;
           } else
     printf("random_des_ctr finished "
      "successfuly\n");
   }
   data_length++;
  }
 }

 if (error_count)
  printf("%i testcases failed\n", error_count);
 else
  printf("All testcases finished successfully\n");

 return rc;
}
```

## Triple DES with CBC mode example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2010, 2011 */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

#define NR_RANDOM_TESTS 10000

void dump_array(unsigned char *ptr, unsigned int size)
{
 unsigned char *ptr_end;
 unsigned char *h;
 int i = 1;

 h = ptr;
 ptr_end = ptr + size;
 while (h < (unsigned char *)ptr_end) {
  printf("0x%02x ",(unsigned char ) *h);
  h++;
  if (i == 8) {
   printf("\n");
   i = 1;
  } else {
   ++i;
  }
 }
 printf("\n");
```

```
}

void dump_cbc_data(unsigned char *iv, unsigned int iv_length,
                   unsigned char *key, unsigned int key_length,
                   unsigned char *input_data, unsigned int data_length,
                   unsigned char *output_data)
{
 printf("IV \n");
 dump_array(iv, iv_length);
 printf("Key \n");
 dump_array(key, key_length);
 printf("Input Data\n");
 dump_array(input_data, data_length);
 printf("Output Data\n");
 dump_array(output_data, data_length);
}

int load_random_test_data(unsigned char *data, unsigned int data_length,
          unsigned char *iv, unsigned int iv_length,
          unsigned char *key, unsigned int key_length)
{
 int rc;
 rc = ica_random_number_generate(data_length, data);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
         rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(iv_length, iv);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
         rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(key_length, key);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
         rc, errno);
  return rc;
 }
 return rc;
}

int random_3des_cbc(int iteration, int silent, unsigned int data_length)
{
 unsigned int iv_length = sizeof(ica_des_vector_t);
 unsigned int key_length = sizeof(ica_des_key_triple_t);

 unsigned char iv[iv_length];
 unsigned char tmp_iv[iv_length];
 unsigned char key[key_length];
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];

 int rc = 0;
 memset(encrypt, 0x00, data_length);
 memset(decrypt, 0x00, data_length);

 load_random_test_data(input_data, data_length, iv, iv_length, key,
         key_length);
 memcpy(tmp_iv, iv, iv_length);

 printf("Test Parameters for iteration = %i\n", iteration);
 printf("key length = %i, data length = %i, iv length = %i\n",
        key_length, data_length, iv_length);
```

```
 rc = ica_3des_cbc(input_data, encrypt, data_length, key, tmp_iv, 1);
 if (rc) {
  printf("ica_3des_cbc encrypt failed with rc = %i\n", rc);
  dump_cbc_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }
 if (!silent && !rc) {
  printf("Encrypt:\n");
  dump_cbc_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }

 if (rc) {
  printf("3DES CBC test exited after encryption\n");
  return rc;
 }

 memcpy(tmp_iv, iv, iv_length);

 rc = ica_3des_cbc(encrypt, decrypt, data_length,  key, tmp_iv,
    0);
 if (rc) {
  printf("ica_3des_cbc decrypt failed with rc = %i\n", rc);
  dump_cbc_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
  return rc;
 }


 if (!silent && !rc) {
  printf("Decrypt:\n");
  dump_cbc_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
 }
 return rc;
}

int main(int argc, char **argv)
{
 // Default mode is 0. ECB,CBC and CFQ tests will be performed.
 unsigned int silent = 0;
 if (argc > 1) {
  if (strstr(argv[1], "silent"))
   silent = 1;
 }
 int rc = 0;
 int error_count = 0;
 int iteration;
 unsigned int data_length = sizeof(ica_des_vector_t);
 for(iteration = 1; iteration <= NR_RANDOM_TESTS; iteration++) {
  int silent = 1;
  rc = random_3des_cbc(iteration, silent, data_length);
  if (rc) {
   printf("random_3des_cbc failed with rc = %i\n", rc);
   error_count++;
   goto out;
  } else
   printf("random_3des_cbc finished successfuly\n");
```

```
  data_length += sizeof(ica_des_vector_t);
 }
out:
 if (error_count)
  printf("%i testcases failed\n", error_count);
 else
  printf("All testcases finished successfully\n");

 return rc;
}
```

## AES with CFB mode example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2010, 2011 */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

#define NR_TESTS 12
#define NR_RANDOM_TESTS 1000

/* CFB128 data -1- AES128 */
unsigned char NIST_KEY_CFB_E1[] = {
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
};

unsigned char NIST_IV_CFB_E1[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_CFB_E1[] = {
 0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
 0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
};

unsigned char NIST_TEST_DATA_CFB_E1[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
};

unsigned char NIST_TEST_RESULT_CFB_E1[] = {
 0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
 0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
};

unsigned int NIST_LCFB_E1 = 128 / 8;

/* CFB128 data -2- AES128 */
unsigned char NIST_KEY_CFB_E2[] = {
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
};

unsigned char NIST_IV_CFB_E2[] = {
 0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
```

```
 0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
};

unsigned char NIST_EXPECTED_IV_CFB_E2[] = {
 0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
 0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b,
};

unsigned char NIST_TEST_DATA_CFB_E2[] = {
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
};

unsigned char NIST_TEST_RESULT_CFB_E2[] = {
 0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
 0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b,
};

unsigned int NIST_LCFB_E2 = 128 / 8;

/* CFB8 data -3- AES128 */
unsigned char NIST_KEY_CFB_E3[] = {
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
};

unsigned char NIST_IV_CFB_E3[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_CFB_E3[] = {
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x3b,
};
unsigned char NIST_TEST_DATA_CFB_E3[] = {
 0x6b,
};

unsigned char NIST_TEST_RESULT_CFB_E3[] = {
 0x3b,
};
unsigned int NIST_LCFB_E3 = 8 / 8;

/* CFB8 data -4- AES128 */
unsigned char NIST_KEY_CFB_E4[] = {
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
};

unsigned char NIST_IV_CFB_E4[] = {
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x3b,
};

unsigned char NIST_EXPECTED_IV_CFB_E4[] = {
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x3b, 0x79,
};
unsigned char NIST_TEST_DATA_CFB_E4[] = {
 0xc1,
};

unsigned char NIST_TEST_RESULT_CFB_E4[] = {
 0x79,
};
```

```
unsigned int NIST_LCFB_E4 = 8 / 8;


/* CFB 128 data -5- for AES192 */
unsigned char NIST_KEY_CFB_E5[] = {
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b,
};

unsigned char NIST_IV_CFB_E5[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_CFB_E5[] = {
 0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
 0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
};

unsigned char NIST_TEST_DATA_CFB_E5[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
};

unsigned char NIST_TEST_RESULT_CFB_E5[] = {
 0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
 0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
};

unsigned int NIST_LCFB_E5 = 128 / 8;

/* CFB 128 data -6- for AES192 */
unsigned char NIST_KEY_CFB_E6[] = {
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b,
};

unsigned char NIST_IV_CFB_E6[] = {
 0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
 0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
};

unsigned char NIST_EXPECTED_IV_CFB_E6[] = {
 0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
 0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a,
};

unsigned char NIST_TEST_DATA_CFB_E6[] = {
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
};

unsigned char NIST_TEST_RESULT_CFB_E6[] = {
 0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
 0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a,
};

unsigned int NIST_LCFB_E6 = 128 / 8;

/* CFB 128 data -7- for AES192 */
unsigned char NIST_KEY_CFB_E7[] = {
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b,
};
```

```
unsigned char NIST_IV_CFB_E7[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_CFB_E7[] = {
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0xcd,
};

unsigned char NIST_TEST_DATA_CFB_E7[] = {
 0x6b,
};

unsigned char NIST_TEST_RESULT_CFB_E7[] = {
 0xcd,
};

unsigned int NIST_LCFB_E7 = 8 / 8;

/* CFB 128 data -8- for AES192 */
unsigned char NIST_KEY_CFB_E8[] = {
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b,
};

unsigned char NIST_IV_CFB_E8[] = {
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0xcd,
};

unsigned char NIST_EXPECTED_IV_CFB_E8[] = {
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0xcd, 0xa2,
};

unsigned char NIST_TEST_DATA_CFB_E8[] = {
 0xc1,
};

unsigned char NIST_TEST_RESULT_CFB_E8[] = {
 0xa2,
};

unsigned int NIST_LCFB_E8 = 8 / 8;



/* CFB128 data -9- for AES256 */
unsigned char NIST_KEY_CFB_E9[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CFB_E9[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_CFB_E9[] = {
 0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
 0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
};
```

```
unsigned char NIST_TEST_DATA_CFB_E9[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
};

unsigned char NIST_TEST_RESULT_CFB_E9[] = {
 0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
 0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
};

unsigned int NIST_LCFB_E9 = 128 / 8;

/* CFB128 data -10- for AES256 */
unsigned char NIST_KEY_CFB_E10[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CFB_E10[] = {
 0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
 0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
};

unsigned char NIST_EXPECTED_IV_CFB_E10[] = {
 0x39, 0xff, 0xed, 0x14, 0x3b, 0x28, 0xb1, 0xc8,
 0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b,
};

unsigned char NIST_TEST_DATA_CFB_E10[] = {
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
};

unsigned char NIST_TEST_RESULT_CFB_E10[] = {
 0x39, 0xff, 0xed, 0x14, 0x3b, 0x28, 0xb1, 0xc8,
 0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b,
};

unsigned int NIST_LCFB_E10 = 128 / 8;

/* CFB8 data -11- for AES256 */
unsigned char NIST_KEY_CFB_E11[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CFB_E11[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_CFB_E11[] = {
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0xdc,
};

unsigned char NIST_TEST_DATA_CFB_E11[] = {
 0x6b,
};

unsigned char NIST_TEST_RESULT_CFB_E11[] = {
 0xdc,
```

```
};

unsigned int NIST_LCFB_E11 = 8 / 8;

/* CFB8 data -12- for AES256 */
unsigned char NIST_KEY_CFB_E12[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CFB_E12[] = {
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0xdc,
};

unsigned char NIST_EXPECTED_IV_CFB_E12[] = {
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0xdc, 0x1f,
};

unsigned char NIST_TEST_DATA_CFB_E12[] = {
 0xc1,
};

unsigned char NIST_TEST_RESULT_CFB_E12[] = {
 0x1f,
};

unsigned int NIST_LCFB_E12 = 8 / 8;


void dump_array(unsigned char *ptr, unsigned int size)
{
 unsigned char *ptr_end;
 unsigned char *h;
 int i = 1;

 h = ptr;
 ptr_end = ptr + size;
 while (h < (unsigned char *)ptr_end) {
  printf("0x%02x ",(unsigned char ) *h);
  h++;
  if (i == 8) {
   printf("\n");
   i = 1;
  } else {
   ++i;
  }
 }
 printf("\n");
}

void dump_cfb_data(unsigned char *iv, unsigned int iv_length,
                   unsigned char *key, unsigned int key_length,
                   unsigned char *input_data, unsigned int data_length,
                   unsigned char *output_data)
{
 printf("IV \n");
 dump_array(iv, iv_length);
 printf("Key \n");
 dump_array(key, key_length);
 printf("Input Data\n");
 dump_array(input_data, data_length);
 printf("Output Data\n");
 dump_array(output_data, data_length);
```

```
}

void get_sizes(unsigned int *data_length, unsigned int *iv_length,
        unsigned int *key_length, unsigned int iteration)
{
 switch (iteration) {
  case 1:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E1);
   *iv_length = sizeof(NIST_IV_CFB_E1);
   *key_length = sizeof(NIST_KEY_CFB_E1);
   break;
  case 2:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E2);
   *iv_length = sizeof(NIST_IV_CFB_E2);
   *key_length = sizeof(NIST_KEY_CFB_E2);
   break;
  case 3:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E3);
   *iv_length = sizeof(NIST_IV_CFB_E3);
   *key_length = sizeof(NIST_KEY_CFB_E3);
   break;
  case 4:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E4);
   *iv_length = sizeof(NIST_IV_CFB_E4);
   *key_length = sizeof(NIST_KEY_CFB_E4);
   break;
  case 5:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E5);
   *iv_length = sizeof(NIST_IV_CFB_E5);
   *key_length = sizeof(NIST_KEY_CFB_E5);
   break;
  case 6:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E6);
   *iv_length = sizeof(NIST_IV_CFB_E6);
   *key_length = sizeof(NIST_KEY_CFB_E6);
   break;
  case 7:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E7);
   *iv_length = sizeof(NIST_IV_CFB_E7);
   *key_length = sizeof(NIST_KEY_CFB_E7);
   break;
  case 8:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E8);
   *iv_length = sizeof(NIST_IV_CFB_E8);
   *key_length = sizeof(NIST_KEY_CFB_E8);
   break;
  case 9:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E9);
   *iv_length = sizeof(NIST_IV_CFB_E9);
   *key_length = sizeof(NIST_KEY_CFB_E9);
   break;
  case 10:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E10);
   *iv_length = sizeof(NIST_IV_CFB_E10);
   *key_length = sizeof(NIST_KEY_CFB_E10);
   break;
  case 11:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E11);
   *iv_length = sizeof(NIST_IV_CFB_E11);
   *key_length = sizeof(NIST_KEY_CFB_E11);
   break;
  case 12:
   *data_length = sizeof(NIST_TEST_DATA_CFB_E12);
   *iv_length = sizeof(NIST_IV_CFB_E12);
   *key_length = sizeof(NIST_KEY_CFB_E12);
   break;
 }
```

```
}

void load_test_data(unsigned char *data, unsigned int data_length,
      unsigned char *result,
      unsigned char *iv, unsigned char *expected_iv,
      unsigned int iv_length,
      unsigned char *key, unsigned int key_length,
      unsigned int *lcfb, unsigned int iteration)
{
 switch (iteration) {
  case 1:
   memcpy(data, NIST_TEST_DATA_CFB_E1, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E1, data_length);
   memcpy(iv, NIST_IV_CFB_E1, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E1, iv_length);
   memcpy(key, NIST_KEY_CFB_E1, key_length);
   *lcfb = NIST_LCFB_E1;
   break;
  case 2:
   memcpy(data, NIST_TEST_DATA_CFB_E2, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E2, data_length);
   memcpy(iv, NIST_IV_CFB_E2, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E2, iv_length);
   memcpy(key, NIST_KEY_CFB_E2, key_length);
   *lcfb = NIST_LCFB_E2;
   break;
  case 3:
   memcpy(data, NIST_TEST_DATA_CFB_E3, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E3, data_length);
   memcpy(iv, NIST_IV_CFB_E3, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E3, iv_length);
   memcpy(key, NIST_KEY_CFB_E3, key_length);
   *lcfb = NIST_LCFB_E3;
   break;
  case 4:
   memcpy(data, NIST_TEST_DATA_CFB_E4, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E4, data_length);
   memcpy(iv, NIST_IV_CFB_E4, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E4, iv_length);
   memcpy(key, NIST_KEY_CFB_E4, key_length);
   *lcfb = NIST_LCFB_E4;
   break;
  case 5:
   memcpy(data, NIST_TEST_DATA_CFB_E5, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E5, data_length);
   memcpy(iv, NIST_IV_CFB_E5, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E5, iv_length);
   memcpy(key, NIST_KEY_CFB_E5, key_length);
   *lcfb = NIST_LCFB_E5;
   break;
  case 6:
   memcpy(data, NIST_TEST_DATA_CFB_E6, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E6, data_length);
   memcpy(iv, NIST_IV_CFB_E6, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E6, iv_length);
   memcpy(key, NIST_KEY_CFB_E6, key_length);
   *lcfb = NIST_LCFB_E6;
   break;
  case 7:
   memcpy(data, NIST_TEST_DATA_CFB_E7, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E7, data_length);
   memcpy(iv, NIST_IV_CFB_E7, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E7, iv_length);
   memcpy(key, NIST_KEY_CFB_E7, key_length);
   *lcfb = NIST_LCFB_E7;
   break;
```

```
  case 8:
   memcpy(data, NIST_TEST_DATA_CFB_E8, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E8, data_length);
   memcpy(iv, NIST_IV_CFB_E8, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E8, iv_length);
   memcpy(key, NIST_KEY_CFB_E8, key_length);
   *lcfb = NIST_LCFB_E8;
   break;
  case 9:
   memcpy(data, NIST_TEST_DATA_CFB_E9, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E9, data_length);
   memcpy(iv, NIST_IV_CFB_E9, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E9, iv_length);
   memcpy(key, NIST_KEY_CFB_E9, key_length);
   *lcfb = NIST_LCFB_E9;
   break;
  case 10:
   memcpy(data, NIST_TEST_DATA_CFB_E10, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E10, data_length);
   memcpy(iv, NIST_IV_CFB_E10, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E10, iv_length);
   memcpy(key, NIST_KEY_CFB_E10, key_length);
   *lcfb = NIST_LCFB_E10;
   break;
  case 11:
   memcpy(data, NIST_TEST_DATA_CFB_E11, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E11, data_length);
   memcpy(iv, NIST_IV_CFB_E11, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E11, iv_length);
   memcpy(key, NIST_KEY_CFB_E11, key_length);
   *lcfb = NIST_LCFB_E11;
   break;
  case 12:
   memcpy(data, NIST_TEST_DATA_CFB_E12, data_length);
   memcpy(result, NIST_TEST_RESULT_CFB_E12, data_length);
   memcpy(iv, NIST_IV_CFB_E12, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CFB_E12, iv_length);
   memcpy(key, NIST_KEY_CFB_E12, key_length);
   *lcfb = NIST_LCFB_E12;
   break;
 }

}

int kat_aes_cfb(int iteration, int silent)
{
 unsigned int data_length;
 unsigned int iv_length;
 unsigned int key_length;

 get_sizes(&data_length, &iv_length, &key_length, iteration);


 unsigned char iv[iv_length];
 unsigned char tmp_iv[iv_length];
 unsigned char expected_iv[iv_length];
 unsigned char key[key_length];
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];
 unsigned char result[data_length];

 int rc = 0;
 unsigned int lcfb;
 memset(encrypt, 0x00, data_length);
 memset(decrypt, 0x00, data_length);
```

```
load_test_data(input_data, data_length, result, iv, expected_iv,
        iv_length, key, key_length, &lcfb, iteration);
memcpy(tmp_iv, iv, iv_length);

printf("Test Parameters for iteration = %i\n", iteration);
printf("key length = %i, data length = %i, iv length = %i,"
       " lcfb = %i\n", key_length, data_length, iv_length, lcfb);

if (iteration == 3)
rc = ica_aes_cfb(input_data, encrypt, lcfb, key, key_length, tmp_iv,
   lcfb, 1);
else
rc = ica_aes_cfb(input_data, encrypt, data_length, key, key_length,
   tmp_iv, lcfb, 1);
if (rc) {
 printf("ica_aes_cfb encrypt failed with rc = %i\n", rc);
 dump_cfb_data(iv, iv_length, key, key_length, input_data,
        data_length, encrypt);
}
if (!silent && !rc) {
 printf("Encrypt:\n");
 dump_cfb_data(iv, iv_length, key, key_length, input_data,
        data_length, encrypt);
}

if (memcmp(result, encrypt, data_length)) {
 printf("Encryption Result does not match the known ciphertext!\n");
 printf("Expected data:\n");
 dump_array(result, data_length);
 printf("Encryption Result:\n");
 dump_array(encrypt, data_length);
 rc++;
}

if (memcmp(expected_iv, tmp_iv, iv_length)) {
 printf("Update of IV does not match the expected IV!\n");
 printf("Expected IV:\n");
 dump_array(expected_iv, iv_length);
 printf("Updated IV:\n");
 dump_array(tmp_iv, iv_length);
 printf("Original IV:\n");
 dump_array(iv, iv_length);
 rc++;
}
if (rc) {
 printf("AES OFB test exited after encryption\n");
 return rc;
}

memcpy(tmp_iv, iv, iv_length);
if (iteration == 3)
rc = ica_aes_cfb(encrypt, decrypt, lcfb, key, key_length, tmp_iv,
   lcfb, 0);
else
rc = ica_aes_cfb(encrypt, decrypt, data_length, key, key_length,
   tmp_iv, lcfb, 0);
if (rc) {
 printf("ica_aes_cfb decrypt failed with rc = %i\n", rc);
 dump_cfb_data(iv, iv_length, key, key_length, encrypt,
        data_length, decrypt);
 return rc;
}


if (!silent && !rc) {
 printf("Decrypt:\n");
 dump_cfb_data(iv, iv_length, key, key_length, encrypt,
```

```
        data_length, decrypt);
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
 }
 return rc;
}

int load_random_test_data(unsigned char *data, unsigned int data_length,
          unsigned char *iv, unsigned int iv_length,
          unsigned char *key, unsigned int key_length)
{
 int rc;
 rc = ica_random_number_generate(data_length, data);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
        rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(iv_length, iv);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
        rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(key_length, key);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
        rc, errno);
  return rc;
 }
 return rc;
}

int random_aes_cfb(int iteration, int silent, unsigned int data_length,
     unsigned int lcfb)
{
 unsigned int iv_length = sizeof(ica_aes_vector_t);
 unsigned int key_length = AES_KEY_LEN128;

 unsigned char iv[iv_length];
 unsigned char tmp_iv[iv_length];
 unsigned char key[key_length];
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];

 int rc = 0;
 for (key_length = AES_KEY_LEN128; key_length <= AES_KEY_LEN256; key_length += 8) {
 memset(encrypt, 0x00, data_length);
 memset(decrypt, 0x00, data_length);

 load_random_test_data(input_data, data_length, iv, iv_length, key,
        key_length);
 memcpy(tmp_iv, iv, iv_length);

 printf("Test Parameters for iteration = %i\n", iteration);
 printf("key length = %i, data length = %i, iv length = %i,"
       " lcfb = %i\n", key_length, data_length, iv_length, lcfb);

 rc = ica_aes_cfb(input_data, encrypt, data_length, key, key_length,
```

```
    tmp_iv, lcfb, 1);
 if (rc) {
  printf("ica_aes_cfb encrypt failed with rc = %i\n", rc);
  dump_cfb_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }
 if (!silent && !rc) {
  printf("Encrypt:\n");
  dump_cfb_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }

 if (rc) {
  printf("AES OFB test exited after encryption\n");
  return rc;
 }

 memcpy(tmp_iv, iv, iv_length);

 rc = ica_aes_cfb(encrypt, decrypt, data_length, key, key_length,
    tmp_iv, lcfb, 0);
 if (rc) {
  printf("ica_aes_cfb decrypt failed with rc = %i\n", rc);
  dump_cfb_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
  return rc;
 }


 if (!silent && !rc) {
  printf("Decrypt:\n");
  dump_cfb_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
 }
 }
 return rc;
}

int main(int argc, char **argv)
{
 unsigned int silent = 0;
 unsigned int endless = 0;
 if (argc > 1) {
  if (strstr(argv[1], "silent"))
   silent = 1;
  if (strstr(argv[1], "endless"))
   endless = 1;
 }
 int rc = 0;
 int error_count = 0;
 int iteration;
 for(iteration = 1; iteration <= NR_TESTS; iteration++) {
  rc = kat_aes_cfb(iteration, silent);
  if (rc) {
   printf("kat_aes_cfb failed with rc = %i\n", rc);
   error_count++;
  } else
   printf("kat_aes_cfb finished successfuly\n");
```

```
 }

 unsigned int data_length = 1;
 unsigned int lcfb = 1;
 unsigned int j;
 for(iteration = 1; iteration <= NR_RANDOM_TESTS; iteration++) {
  for (j = 1; j <= 3; j++) {
   int silent = 1;
   if (!(data_length % lcfb)) {
   rc = random_aes_cfb(iteration, silent, data_length, lcfb);
   if (rc) {
    printf("random_aes_cfb failed with rc = %i\n", rc);
    error_count++;
   } else
    printf("random_aes_cfb finished successfuly\n");
   }
   switch (j) {
    case 1:
     lcfb = 1;
     break;
    case 2:
     lcfb = 8;
     break;
    case 3:
     lcfb = 16;
     break;
   }
  }
  if (data_length == 1)
   data_length = 8;
  else
   data_length += 8;
 }
 if (error_count)
  printf("%i testcases failed\n", error_count);
 else
  printf("All testcases finished successfully\n");

 return rc;
}
```

## AES with CTR mode example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2010, 2011 */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

#define NR_TESTS 7

/* CTR data - 1 for AES128 */
unsigned char NIST_KEY_CTR_E1[] = {
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
};
```

```
unsigned char NIST_IV_CTR_E1[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
};

unsigned char NIST_EXPECTED_IV_CTR_E1[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x00,
};

unsigned char NIST_TEST_DATA_CTR_E1[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
};

unsigned char NIST_TEST_RESULT_CTR_E1[] = {
 0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
 0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
};

/* CTR data - 2 for AES128 */
unsigned char NIST_KEY_CTR_E2[] = {
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
};

unsigned char NIST_IV_CTR_E2[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
};

unsigned char NIST_EXPECTED_IV_CTR_E2[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x03,
};

unsigned char NIST_TEST_DATA_CTR_E2[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
 0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
 0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10,
};

unsigned char NIST_TEST_RESULT_CTR_E2[] = {
 0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
 0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
 0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
 0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff,
 0x5a, 0xe4, 0xdf, 0x3e, 0xdb, 0xd5, 0xd3, 0x5e,
 0x5b, 0x4f, 0x09, 0x02, 0x0d, 0xb0, 0x3e, 0xab,
 0x1e, 0x03, 0x1d, 0xda, 0x2f, 0xbe, 0x03, 0xd1,
 0x79, 0x21, 0x70, 0xa0, 0xf3, 0x00, 0x9c, 0xee,
};

/* CTR data - 3 - for AES192 */
unsigned char NIST_KEY_CTR_E3[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CTR_E3[] = {
```

```
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
};

unsigned char NIST_EXPECTED_IV_CTR_E3[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x00,
};

unsigned char NIST_TEST_DATA_CTR_E3[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
};

unsigned char NIST_TEST_RESULT_CTR_E3[] = {
 0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
 0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
};

/* CTR data - 4 - for AES192 */
unsigned char NIST_KEY_CTR_E4[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CTR_E4[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x00,
};

unsigned char NIST_EXPECTED_IV_CTR_E4[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x01,
};

unsigned char NIST_TEST_DATA_CTR_E4[] = {
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
};

unsigned char NIST_TEST_RESULT_CTR_E4[] = {
 0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
 0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5,
};

/* CTR data 5 - for AES 256 */
unsigned char NIST_KEY_CTR_E5[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CTR_E5[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
};

unsigned char NIST_EXPECTED_IV_CTR_E5[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x03,
};

unsigned char NIST_TEST_DATA_CTR_E5[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
```

```
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
 0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
 0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10,
};

unsigned char NIST_TEST_RESULT_CTR_E5[] = {
 0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
 0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
 0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
 0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5,
 0x2b, 0x09, 0x30, 0xda, 0xa2, 0x3d, 0xe9, 0x4c,
 0xe8, 0x70, 0x17, 0xba, 0x2d, 0x84, 0x98, 0x8d,
 0xdf, 0xc9, 0xc5, 0x8d, 0xb6, 0x7a, 0xad, 0xa6,
 0x13, 0xc2, 0xdd, 0x08, 0x45, 0x79, 0x41, 0xa6,
};

/* CTR data 6 - for AES 256.
 * Data is != BLOCK_SIZE */
unsigned char NIST_KEY_CTR_E6[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CTR_E6[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
};

unsigned char NIST_EXPECTED_IV_CTR_E6[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x03,
};

unsigned char NIST_TEST_DATA_CTR_E6[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
 0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
 0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
};

unsigned char NIST_TEST_RESULT_CTR_E6[] = {
 0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
 0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
 0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
 0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5,
 0x2b, 0x09, 0x30, 0xda, 0xa2, 0x3d, 0xe9, 0x4c,
 0xe8, 0x70, 0x17, 0xba, 0x2d, 0x84, 0x98, 0x8d,
 0xdf, 0xc9, 0xc5, 0x8d, 0xb6, 0x7a, 0xad, 0xa6,
};

/* CTR data 7 - for AES 256
 * Counter as big as the data. Therefore the counter
 * should not be updated. Because it is already pre
 * computed. */
unsigned char NIST_KEY_CTR_E7[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
```

```
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_CTR_E7[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x00,
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x01,
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x02,
};

unsigned char NIST_EXPECTED_IV_CTR_E7[] = {
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x00,
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x01,
 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xff, 0x02,
};

unsigned char NIST_TEST_DATA_CTR_E7[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
 0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
 0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10,
};

unsigned char NIST_TEST_RESULT_CTR_E7[] = {
 0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
 0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
 0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
 0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5,
 0x2b, 0x09, 0x30, 0xda, 0xa2, 0x3d, 0xe9, 0x4c,
 0xe8, 0x70, 0x17, 0xba, 0x2d, 0x84, 0x98, 0x8d,
 0xdf, 0xc9, 0xc5, 0x8d, 0xb6, 0x7a, 0xad, 0xa6,
 0x13, 0xc2, 0xdd, 0x08, 0x45, 0x79, 0x41, 0xa6,
};


void dump_array(unsigned char *ptr, unsigned int size)
{
 unsigned char *ptr_end;
 unsigned char *h;
 int i = 1;

 h = ptr;
 ptr_end = ptr + size;
 while (h < (unsigned char *)ptr_end) {
  printf("0x%02x ",(unsigned char ) *h);
  h++;
  if (i == 8) {
   printf("\n");
   i = 1;
  } else {
   ++i;
  }
 }
 printf("\n");
```

```
}

void dump_ctr_data(unsigned char *iv, unsigned int iv_length,
                   unsigned char *key, unsigned int key_length,
                   unsigned char *input_data, unsigned int data_length,
                   unsigned char *output_data)
{
 printf("IV \n");
 dump_array(iv, iv_length);
 printf("Key \n");
 dump_array(key, key_length);
 printf("Input Data\n");
 dump_array(input_data, data_length);
 printf("Output Data\n");
 dump_array(output_data, data_length);
}

void get_sizes(unsigned int *data_length, unsigned int *iv_length,
        unsigned int *key_length, unsigned int iteration)
{
 switch (iteration) {
  case 1:
   *data_length = sizeof(NIST_TEST_DATA_CTR_E1);
   *iv_length = sizeof(NIST_IV_CTR_E1);
   *key_length = sizeof(NIST_KEY_CTR_E1);
   break;
  case 2:
   *data_length = sizeof(NIST_TEST_DATA_CTR_E2);
   *iv_length = sizeof(NIST_IV_CTR_E2);
   *key_length = sizeof(NIST_KEY_CTR_E2);
   break;
  case 3:
   *data_length = sizeof(NIST_TEST_DATA_CTR_E3);
   *iv_length = sizeof(NIST_IV_CTR_E3);
   *key_length = sizeof(NIST_KEY_CTR_E3);
   break;
  case 4:
   *data_length = sizeof(NIST_TEST_DATA_CTR_E4);
   *iv_length = sizeof(NIST_IV_CTR_E4);
   *key_length = sizeof(NIST_KEY_CTR_E4);
   break;
  case 5:
   *data_length = sizeof(NIST_TEST_DATA_CTR_E5);
   *iv_length = sizeof(NIST_IV_CTR_E5);
   *key_length = sizeof(NIST_KEY_CTR_E5);
   break;
  case 6:
   *data_length = sizeof(NIST_TEST_DATA_CTR_E6);
   *iv_length = sizeof(NIST_IV_CTR_E6);
   *key_length = sizeof(NIST_KEY_CTR_E6);
   break;
  case 7:
   *data_length = sizeof(NIST_TEST_DATA_CTR_E7);
   *iv_length = sizeof(NIST_IV_CTR_E7);
   *key_length = sizeof(NIST_KEY_CTR_E7);
   break;
 }

}

void load_test_data(unsigned char *data, unsigned int data_length,
      unsigned char *result,
      unsigned char *iv, unsigned char *expected_iv,
      unsigned int iv_length,
      unsigned char *key, unsigned int key_length,
      unsigned int iteration)
{
```

```
 switch (iteration) {
  case 1:
   memcpy(data, NIST_TEST_DATA_CTR_E1, data_length);
   memcpy(result, NIST_TEST_RESULT_CTR_E1, data_length);
   memcpy(iv, NIST_IV_CTR_E1, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CTR_E1, iv_length);
   memcpy(key, NIST_KEY_CTR_E1, key_length);
   break;
  case 2:
   memcpy(data, NIST_TEST_DATA_CTR_E2, data_length);
   memcpy(result, NIST_TEST_RESULT_CTR_E2, data_length);
   memcpy(iv, NIST_IV_CTR_E2, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CTR_E2, iv_length);
   memcpy(key, NIST_KEY_CTR_E2, key_length);
   break;
  case 3:
   memcpy(data, NIST_TEST_DATA_CTR_E3, data_length);
   memcpy(result, NIST_TEST_RESULT_CTR_E3, data_length);
   memcpy(iv, NIST_IV_CTR_E3, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CTR_E3, iv_length);
   memcpy(key, NIST_KEY_CTR_E3, key_length);
   break;
  case 4:
   memcpy(data, NIST_TEST_DATA_CTR_E4, data_length);
   memcpy(result, NIST_TEST_RESULT_CTR_E4, data_length);
   memcpy(iv, NIST_IV_CTR_E4, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CTR_E4, iv_length);
   memcpy(key, NIST_KEY_CTR_E4, key_length);
   break;
  case 5:
   memcpy(data, NIST_TEST_DATA_CTR_E5, data_length);
   memcpy(result, NIST_TEST_RESULT_CTR_E5, data_length);
   memcpy(iv, NIST_IV_CTR_E5, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CTR_E5, iv_length);
   memcpy(key, NIST_KEY_CTR_E5, key_length);
   break;
  case 6:
   memcpy(data, NIST_TEST_DATA_CTR_E6, data_length);
   memcpy(result, NIST_TEST_RESULT_CTR_E6, data_length);
   memcpy(iv, NIST_IV_CTR_E6, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CTR_E6, iv_length);
   memcpy(key, NIST_KEY_CTR_E6, key_length);
   break;
  case 7:
   memcpy(data, NIST_TEST_DATA_CTR_E7, data_length);
   memcpy(result, NIST_TEST_RESULT_CTR_E7, data_length);
   memcpy(iv, NIST_IV_CTR_E7, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_CTR_E7, iv_length);
   memcpy(key, NIST_KEY_CTR_E7, key_length);
   break;
 }

}

int random_aes_ctr(int iteration, int silent, unsigned int data_length, unsigned int iv_length)
{
 unsigned int key_length = AES_KEY_LEN256;
 if (data_length % sizeof(ica_aes_vector_t))
  iv_length = sizeof(ica_aes_vector_t);

 printf("Test Parameters for iteration = %i\n", iteration);
 printf("key length = %i, data length = %i, iv length = %i\n",
        key_length, data_length, iv_length);

 unsigned char iv[iv_length];
 unsigned char tmp_iv[iv_length];
 unsigned char key[key_length];
```

```
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];

 int rc = 0;
 rc = ica_random_number_generate(data_length, input_data);
 if (rc) {
  printf("random number generate returned rc = %i, errno = %i\n", rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(iv_length, iv);
 if (rc) {
  printf("random number generate returned rc = %i, errno = %i\n", rc, errno);
  return rc;
 }

 rc = ica_random_number_generate(key_length, key);
 if (rc) {
  printf("random number generate returned rc = %i, errno = %i\n", rc, errno);
  return rc;
 }
 memcpy(tmp_iv, iv, iv_length);

 rc = ica_aes_ctr(input_data, encrypt, data_length, key, key_length,
    tmp_iv, 32, 1);
 if (rc) {
  printf("ica_aes_ctr encrypt failed with rc = %i\n", rc);
  dump_ctr_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
  return rc;
 }
 if (!silent && !rc) {
  printf("Encrypt:\n");
  dump_ctr_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }

 memcpy(tmp_iv, iv, iv_length);
 rc = ica_aes_ctr(encrypt, decrypt, data_length, key, key_length,
    tmp_iv, 32, 0);
 if (rc) {
  printf("ica_aes_ctr decrypt failed with rc = %i\n", rc);
  dump_ctr_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
  return rc;
 }


 if (!silent && !rc) {
  printf("Decrypt:\n");
  dump_ctr_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
 }
 return rc;
}

int kat_aes_ctr(int iteration, int silent)
{
```

```
unsigned int data_length;
unsigned int iv_length;
unsigned int key_length;

get_sizes(&data_length, &iv_length, &key_length, iteration);

printf("Test Parameters for iteration = %i\n", iteration);
printf("key length = %i, data length = %i, iv length = %i\n",
       key_length, data_length, iv_length);

unsigned char iv[iv_length];
unsigned char tmp_iv[iv_length];
unsigned char expected_iv[iv_length];
unsigned char key[key_length];
unsigned char input_data[data_length];
unsigned char encrypt[data_length];
unsigned char decrypt[data_length];
unsigned char result[data_length];

int rc = 0;

load_test_data(input_data, data_length, result, iv, expected_iv,
        iv_length, key, key_length, iteration);
memcpy(tmp_iv, iv, iv_length);

if (iv_length == 16)
 rc = ica_aes_ctr(input_data, encrypt, data_length, key, key_length,
    tmp_iv, 32, 1);
else
 rc = ica_aes_ctrlist(input_data, encrypt, data_length, key, key_length,
    tmp_iv, 1);
if (rc) {
 printf("ica_aes_ctr encrypt failed with rc = %i\n", rc);
 dump_ctr_data(iv, iv_length, key, key_length, input_data,
        data_length, encrypt);
}
if (!silent && !rc) {
 printf("Encrypt:\n");
 dump_ctr_data(iv, iv_length, key, key_length, input_data,
        data_length, encrypt);
}

if (memcmp(result, encrypt, data_length)) {
 printf("Encryption Result does not match the known ciphertext!\n");
 printf("Expected data:\n");
 dump_array(result, data_length);
 printf("Encryption Result:\n");
 dump_array(encrypt, data_length);
 rc++;
}

if (memcmp(expected_iv, tmp_iv, iv_length)) {
 printf("Update of IV does not match the expected IV!\n");
 printf("Expected IV:\n");
 dump_array(expected_iv, iv_length);
 printf("Updated IV:\n");
 dump_array(tmp_iv, iv_length);
 printf("Original IV:\n");
 dump_array(iv, iv_length);
 rc++;
}
if (rc) {
 printf("AES CTR test exited after encryption\n");
 return rc;
}

memcpy(tmp_iv, iv, iv_length);
```

```
 rc = ica_aes_ctr(encrypt, decrypt, data_length, key, key_length,
    tmp_iv, 32,0);
 if (rc) {
  printf("ica_aes_ctr decrypt failed with rc = %i\n", rc);
  dump_ctr_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
  return rc;
 }


 if (!silent && !rc) {
  printf("Decrypt:\n");
  dump_ctr_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
 }
 return rc;
}

int main(int argc, char **argv)
{
 // Default mode is 0. ECB,CBC and CFQ tests will be performed.
 unsigned int silent = 0;
 unsigned int endless = 0;
 if (argc > 1) {
  if (strstr(argv[1], "silent"))
   silent = 1;
  if (strstr(argv[1], "endless"))
   endless = 1;
 }
 int rc = 0;
 int error_count = 0;
 int iteration;
 if (!endless)
 for(iteration = 1; iteration <= NR_TESTS; iteration++) {
  rc = kat_aes_ctr(iteration, silent);
  if (rc) {
   printf("kat_aes_ctr failed with rc = %i\n", rc);
   error_count++;
  } else
   printf("kat_aes_ctr finished successfuly\n");

 }
 int i = 0;
 if (endless)
 while (1) {
  printf("i = %i\n",i);
  silent = 1;
  rc = random_aes_ctr(i, silent, 320, 320);
  if (rc) {
   printf("kat_aes_ctr failed with rc = %i\n", rc);
   return rc;
  } else
   printf("kat_aes_ctr finished successfuly\n");
  i++;
 }

 if (error_count)
  printf("%i testcases failed\n", error_count);
```

```
 else
  printf("All testcases finished successfully\n");

 return rc;
}
```

## AES with OFB mode example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2010, 2011 */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

#define NR_TESTS 6
#define NR_RANDOM_TESTS 10000

/* OFB data - 1 for AES128 */
unsigned char NIST_KEY_OFB_E1[] = {
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
};

unsigned char NIST_IV_OFB_E1[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_OFB_E1[] = {
 0x50, 0xfe, 0x67, 0xcc, 0x99, 0x6d, 0x32, 0xb6,
 0xda, 0x09, 0x37, 0xe9, 0x9b, 0xaf, 0xec, 0x60,
};

unsigned char NIST_TEST_DATA_OFB_E1[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
};

unsigned char NIST_TEST_RESULT_OFB_E1[] = {
 0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
 0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
};

/* OFB data - 2 for AES128 */
unsigned char NIST_KEY_OFB_E2[] = {
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
};

unsigned char NIST_IV_OFB_E2[] = {
 0x50, 0xfe, 0x67, 0xcc, 0x99, 0x6d, 0x32, 0xb6,
 0xda, 0x09, 0x37, 0xe9, 0x9b, 0xaf, 0xec, 0x60,
};

unsigned char NIST_EXPECTED_IV_OFB_E2[] = {
 0xd9, 0xa4, 0xda, 0xda, 0x08, 0x92, 0x23, 0x9f,
 0x6b, 0x8b, 0x3d, 0x76, 0x80, 0xe1, 0x56, 0x74,
};
```

```
unsigned char NIST_TEST_DATA_OFB_E2[] = {
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
};

unsigned char NIST_TEST_RESULT_OFB_E2[] = {
 0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
 0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25,
};

/* OFB data - 3 - for AES192 */
unsigned char NIST_KEY_OFB_E3[] = {
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b,
};

unsigned char NIST_IV_OFB_E3[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_OFB_E3[] = {
 0xa6, 0x09, 0xb3, 0x8d, 0xf3, 0xb1, 0x13, 0x3d,
 0xdd, 0xff, 0x27, 0x18, 0xba, 0x09, 0x56, 0x5e,
};

unsigned char NIST_TEST_DATA_OFB_E3[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
};

unsigned char NIST_TEST_RESULT_OFB_E3[] = {
 0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
 0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
};

/* OFB data - 4 - for AES192 */
unsigned char NIST_KEY_OFB_E4[] = {
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
 0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b,
};

unsigned char NIST_IV_OFB_E4[] = {
 0xa6, 0x09, 0xb3, 0x8d, 0xf3, 0xb1, 0x13, 0x3d,
 0xdd, 0xff, 0x27, 0x18, 0xba, 0x09, 0x56, 0x5e,
};

unsigned char NIST_EXPECTED_IV_OFB_E4[] = {
 0x52, 0xef, 0x01, 0xda, 0x52, 0x60, 0x2f, 0xe0,
 0x97, 0x5f, 0x78, 0xac, 0x84, 0xbf, 0x8a, 0x50,
};

unsigned char NIST_TEST_DATA_OFB_E4[] = {
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
};

unsigned char NIST_TEST_RESULT_OFB_E4[] = {
 0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
 0x09, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01,
};

/* OFB data 5 - for AES 256 */
unsigned char NIST_KEY_OFB_E5[] = {
```

```
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_OFB_E5[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_EXPECTED_IV_OFB_E5[] = {
 0xb7, 0xbf, 0x3a, 0x5d, 0xf4, 0x39, 0x89, 0xdd,
 0x97, 0xf0, 0xfa, 0x97, 0xeb, 0xce, 0x2f, 0x4a,
};

unsigned char NIST_TEST_DATA_OFB_E5[] = {
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
};

unsigned char NIST_TEST_RESULT_OFB_E5[] = {
 0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
 0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
};

/* OFB data 6 - for AES 256 */
unsigned char NIST_KEY_OFB_E6[] = {
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4,
};

unsigned char NIST_IV_OFB_E6[] = {
 0xb7, 0xbf, 0x3a, 0x5d, 0xf4, 0x39, 0x89, 0xdd,
 0x97, 0xf0, 0xfa, 0x97, 0xeb, 0xce, 0x2f, 0x4a,
};

unsigned char NIST_EXPECTED_IV_OFB_E6[] = {
 0xe1, 0xc6, 0x56, 0x30, 0x5e, 0xd1, 0xa7, 0xa6,
 0x56, 0x38, 0x05, 0x74, 0x6f, 0xe0, 0x3e, 0xdc,
};

unsigned char NIST_TEST_DATA_OFB_E6[] = {
 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
};

unsigned char NIST_TEST_RESULT_OFB_E6[] = {
 0x4f, 0xeb, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,
 0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d,
};


void dump_array(unsigned char *ptr, unsigned int size)
{
 unsigned char *ptr_end;
 unsigned char *h;
 int i = 1;

 h = ptr;
 ptr_end = ptr + size;
 while (h < (unsigned char *)ptr_end) {
  printf("0x%02x ",(unsigned char ) *h);
  h++;
  if (i == 8) {
```

```
   printf("\n");
   i = 1;
  } else {
   ++i;
  }
 }
 printf("\n");
}

void dump_ofb_data(unsigned char *iv, unsigned int iv_length,
                   unsigned char *key, unsigned int key_length,
                   unsigned char *input_data, unsigned int data_length,
                   unsigned char *output_data)
{
 printf("IV \n");
 dump_array(iv, iv_length);
 printf("Key \n");
 dump_array(key, key_length);
 printf("Input Data\n");
 dump_array(input_data, data_length);
 printf("Output Data\n");
 dump_array(output_data, data_length);
}

void get_sizes(unsigned int *data_length, unsigned int *iv_length,
        unsigned int *key_length, unsigned int iteration)
{
 switch (iteration) {
  case 1:
   *data_length = sizeof(NIST_TEST_DATA_OFB_E1);
   *iv_length = sizeof(NIST_IV_OFB_E1);
   *key_length = sizeof(NIST_KEY_OFB_E1);
   break;
  case 2:
   *data_length = sizeof(NIST_TEST_DATA_OFB_E2);
   *iv_length = sizeof(NIST_IV_OFB_E2);
   *key_length = sizeof(NIST_KEY_OFB_E2);
   break;
  case 3:
   *data_length = sizeof(NIST_TEST_DATA_OFB_E3);
   *iv_length = sizeof(NIST_IV_OFB_E3);
   *key_length = sizeof(NIST_KEY_OFB_E3);
   break;
  case 4:
   *data_length = sizeof(NIST_TEST_DATA_OFB_E4);
   *iv_length = sizeof(NIST_IV_OFB_E4);
   *key_length = sizeof(NIST_KEY_OFB_E4);
   break;
  case 5:
   *data_length = sizeof(NIST_TEST_DATA_OFB_E5);
   *iv_length = sizeof(NIST_IV_OFB_E5);
   *key_length = sizeof(NIST_KEY_OFB_E5);
   break;
  case 6:
   *data_length = sizeof(NIST_TEST_DATA_OFB_E6);
   *iv_length = sizeof(NIST_IV_OFB_E6);
   *key_length = sizeof(NIST_KEY_OFB_E6);
   break;
 }

}

void load_test_data(unsigned char *data, unsigned int data_length,
      unsigned char *result,
      unsigned char *iv, unsigned char *expected_iv,
      unsigned int iv_length,
      unsigned char *key, unsigned int key_length,
```

```
      unsigned int iteration)
{
 switch (iteration) {
  case 1:
   memcpy(data, NIST_TEST_DATA_OFB_E1, data_length);
   memcpy(result, NIST_TEST_RESULT_OFB_E1, data_length);
   memcpy(iv, NIST_IV_OFB_E1, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_OFB_E1, iv_length);
   memcpy(key, NIST_KEY_OFB_E1, key_length);
   break;
  case 2:
   memcpy(data, NIST_TEST_DATA_OFB_E2, data_length);
   memcpy(result, NIST_TEST_RESULT_OFB_E2, data_length);
   memcpy(iv, NIST_IV_OFB_E2, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_OFB_E2, iv_length);
   memcpy(key, NIST_KEY_OFB_E2, key_length);
   break;
  case 3:
   memcpy(data, NIST_TEST_DATA_OFB_E3, data_length);
   memcpy(result, NIST_TEST_RESULT_OFB_E3, data_length);
   memcpy(iv, NIST_IV_OFB_E3, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_OFB_E3, iv_length);
   memcpy(key, NIST_KEY_OFB_E3, key_length);
   break;
  case 4:
   memcpy(data, NIST_TEST_DATA_OFB_E4, data_length);
   memcpy(result, NIST_TEST_RESULT_OFB_E4, data_length);
   memcpy(iv, NIST_IV_OFB_E4, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_OFB_E4, iv_length);
   memcpy(key, NIST_KEY_OFB_E4, key_length);
   break;
  case 5:
   memcpy(data, NIST_TEST_DATA_OFB_E5, data_length);
   memcpy(result, NIST_TEST_RESULT_OFB_E5, data_length);
   memcpy(iv, NIST_IV_OFB_E5, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_OFB_E5, iv_length);
   memcpy(key, NIST_KEY_OFB_E5, key_length);
   break;
  case 6:
   memcpy(data, NIST_TEST_DATA_OFB_E6, data_length);
   memcpy(result, NIST_TEST_RESULT_OFB_E6, data_length);
   memcpy(iv, NIST_IV_OFB_E6, iv_length);
   memcpy(expected_iv, NIST_EXPECTED_IV_OFB_E6, iv_length);
   memcpy(key, NIST_KEY_OFB_E6, key_length);
   break;
 }

}

int load_random_test_data(unsigned char *data, unsigned int data_length,
          unsigned char *iv, unsigned int iv_length,
          unsigned char *key, unsigned int key_length)
{
 int rc;
 rc = ica_random_number_generate(data_length, data);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
        rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(iv_length, iv);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
        rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(key_length, key);
```

```
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
         rc, errno);
  return rc;
 }
 return rc;
}

int random_aes_ofb(int iteration, int silent, unsigned int data_length)
{
 int i;
 int rc = 0;
 unsigned int iv_length = sizeof(ica_aes_vector_t);
 unsigned int key_length = AES_KEY_LEN128;
 unsigned char iv[iv_length];
 unsigned char tmp_iv[iv_length];
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];
for (i = 0; i <= 2; i++) {

 unsigned char key[key_length];

 memset(encrypt, 0x00, data_length);
 memset(decrypt, 0x00, data_length);

 load_random_test_data(input_data, data_length, iv, iv_length, key,
         key_length);
 memcpy(tmp_iv, iv, iv_length);
 printf("Test Parameters for iteration = %i\n", iteration);
 printf("key length = %i, data length = %i, iv length = %i\n",
        key_length, data_length, iv_length);

 rc = ica_aes_ofb(input_data, encrypt, data_length, key, key_length,
    tmp_iv, 1);
 if (rc) {
  printf("ica_aes_ofb encrypt failed with rc = %i\n", rc);
  dump_ofb_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }
 if (!silent && !rc) {
  printf("Encrypt:\n");
  dump_ofb_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }

 if (rc) {
  printf("AES OFB test exited after encryption\n");
  return rc;
 }

 memcpy(tmp_iv, iv, iv_length);

 rc = ica_aes_ofb(encrypt, decrypt, data_length, key, key_length,
    tmp_iv, 0);
 if (rc) {
  printf("ica_aes_ofb decrypt failed with rc = %i\n", rc);
  dump_ofb_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
  return rc;
 }


 if (!silent && !rc) {
  printf("Decrypt:\n");
  dump_ofb_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
```

```
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
  return rc;
 }
 key_length += 8;
}

 return rc;
}

int kat_aes_ofb(int iteration, int silent)
{
 unsigned int data_length;
 unsigned int iv_length;
 unsigned int key_length;

 get_sizes(&data_length, &iv_length, &key_length, iteration);

 printf("Test Parameters for iteration = %i\n", iteration);
 printf("key length = %i, data length = %i, iv length = %i\n",
        key_length, data_length, iv_length);

 unsigned char iv[iv_length];
 unsigned char tmp_iv[iv_length];
 unsigned char expected_iv[iv_length];
 unsigned char key[key_length];
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];
 unsigned char result[data_length];

 int rc = 0;

 load_test_data(input_data, data_length, result, iv, expected_iv,
         iv_length, key, key_length, iteration);
 memcpy(tmp_iv, iv, iv_length);

 rc = ica_aes_ofb(input_data, encrypt, data_length, key, key_length,
    tmp_iv, 1);
 if (rc) {
  printf("ica_aes_ofb encrypt failed with rc = %i\n", rc);
  dump_ofb_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }
 if (!silent && !rc) {
  printf("Encrypt:\n");
  dump_ofb_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }

 if (memcmp(result, encrypt, data_length)) {
  printf("Encryption Result does not match the known ciphertext!\n");
  printf("Expected data:\n");
  dump_array(result, data_length);
  printf("Encryption Result:\n");
  dump_array(encrypt, data_length);
  rc++;
 }

 if (memcmp(expected_iv, tmp_iv, iv_length)) {
```

```
  printf("Update of IV does not match the expected IV!\n");
  printf("Expected IV:\n");
  dump_array(expected_iv, iv_length);
  printf("Updated IV:\n");
  dump_array(tmp_iv, iv_length);
  printf("Original IV:\n");
  dump_array(iv, iv_length);
  rc++;
 }
 if (rc) {
  printf("AES OFB test exited after encryption\n");
  return rc;
 }

 memcpy(tmp_iv, iv, iv_length);
 rc = ica_aes_ofb(encrypt, decrypt, data_length, key, key_length,
    tmp_iv, 0);
 if (rc) {
  printf("ica_aes_ofb decrypt failed with rc = %i\n", rc);
  dump_ofb_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
  return rc;
 }


 if (!silent && !rc) {
  printf("Decrypt:\n");
  dump_ofb_data(iv, iv_length, key, key_length, encrypt,
         data_length, decrypt);
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
 }
 return rc;
}

int main(int argc, char **argv)
{
 unsigned int silent = 0;
 if (argc > 1) {
  if (strstr(argv[1], "silent"))
   silent = 1;
 }
 int rc = 0;
 int error_count = 0;
 int iteration;
 unsigned int data_length = sizeof(ica_aes_vector_t);
 for(iteration = 1; iteration <= NR_TESTS; iteration++) {
  rc = kat_aes_ofb(iteration, silent);
  if (rc) {
   printf("kat_aes_ofb failed with rc = %i\n", rc);
   error_count++;
  } else
   printf("kat_aes_ofb finished successfuly\n");

 }
 for(iteration = 1; iteration <= NR_RANDOM_TESTS; iteration++) {
  int silent = 1;
  rc = random_aes_ofb(iteration, silent, data_length);
  if (rc) {
   printf("random_aes_ofb failed with rc = %i\n", rc);
```

```
   error_count++;
   goto out;
  } else
   printf("random_aes_ofb finished successfuly\n");
  data_length += sizeof(ica_aes_vector_t);
 }

out:
 if (error_count)
  printf("%i testcases failed\n", error_count);
 else
  printf("All testcases finished successfully\n");

 return rc;
}
```

## AES with XTS mode example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2010, 2011 */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

#define NR_TESTS 5
#define NR_RANDOM_TESTS 20000

/* XTS data -1- AES128 */
unsigned char NIST_KEY_XTS_E1[] = {
 0x46, 0xe6, 0xed, 0x9e, 0xf4, 0x2d, 0xcd, 0xb3,
 0xc8, 0x93, 0x09, 0x3c, 0x28, 0xe1, 0xfc, 0x0f,
 0x91, 0xf5, 0xca, 0xa3, 0xb6, 0xe0, 0xbc, 0x5a,
 0x14, 0xe7, 0x83, 0x21, 0x5c, 0x1d, 0x5b, 0x61,
};

unsigned char NIST_TWEAK_XTS_E1[] = {
 0x72, 0xf3, 0xb0, 0x54, 0xcb, 0xdc, 0x2f, 0x9e,
 0x3c, 0x5b, 0xc5, 0x51, 0xd4, 0x4d, 0xdb, 0xa0,
};

/* TWEAK should not be updated, so the exptected tweak is the same as the
 * original TWEAK.
 */
unsigned char NIST_EXPECTED_TWEAK_XTS_E1[] = {
 0x72, 0xf3, 0xb0, 0x54, 0xcb, 0xdc, 0x2f, 0x9e,
 0x3c, 0x5b, 0xc5, 0x51, 0xd4, 0x4d, 0xdb, 0xa0,
};

unsigned char NIST_TEST_DATA_XTS_E1[] = {
 0xe3, 0x77, 0x8d, 0x68, 0xe7, 0x30, 0xef, 0x94,
 0x5b, 0x4a, 0xe3, 0xbc, 0x5b, 0x93, 0x6b, 0xdd,
};

unsigned char NIST_TEST_RESULT_XTS_E1[] = {
 0x97, 0x40, 0x9f, 0x1f, 0x71, 0xae, 0x45, 0x21,
 0xcb, 0x49, 0xa3, 0x29, 0x73, 0xde, 0x4d, 0x05,
};
```

```
/* XTS data -2- AES128 */
unsigned char NIST_KEY_XTS_E2[] = {
 0x93, 0x56, 0xcd, 0xad, 0x25, 0x1a, 0xb6, 0x11,
 0x14, 0xce, 0xc2, 0xc4, 0x4a, 0x60, 0x92, 0xdd,
 0xe9, 0xf7, 0x46, 0xcc, 0x65, 0xae, 0x3b, 0xd4,
 0x96, 0x68, 0x64, 0xaa, 0x36, 0x26, 0xd1, 0x88,
};

unsigned char NIST_TWEAK_XTS_E2[] = {
 0x68, 0x88, 0x27, 0x83, 0x65, 0x24, 0x36, 0xc4,
 0x85, 0x7a, 0x88, 0xc0, 0xc3, 0x73, 0x41, 0x7e,
};

unsigned char NIST_EXPECTED_TWEAK_XTS_E2[] = {
 0x68, 0x88, 0x27, 0x83, 0x65, 0x24, 0x36, 0xc4,
 0x85, 0x7a, 0x88, 0xc0, 0xc3, 0x73, 0x41, 0x7e,
};

unsigned char NIST_TEST_DATA_XTS_E2[] = {
 0xce, 0x17, 0x6b, 0xdd, 0xe3, 0x39, 0x50, 0x5b,
 0xa1, 0x5d, 0xea, 0x36, 0xd2, 0x8c, 0xe8, 0x7d,
};

unsigned char NIST_TEST_RESULT_XTS_E2[] = {
 0x22, 0xf5, 0xf9, 0x37, 0xdf, 0xb3, 0x9e, 0x5b,
 0x74, 0x25, 0xed, 0x86, 0x3d, 0x31, 0x0b, 0xe1,
};

/* XTS data -3- AES128 */
unsigned char NIST_KEY_XTS_E3[] = {
 0x63, 0xf3, 0x6e, 0x9c, 0x39, 0x7c, 0x65, 0x23,
 0xc9, 0x9f, 0x16, 0x44, 0xec, 0xb1, 0xa5, 0xd9,
 0xbc, 0x0f, 0x2f, 0x55, 0xfb, 0xe3, 0x24, 0x44,
 0x4c, 0x39, 0x0f, 0xae, 0x75, 0x2a, 0xd4, 0xd7,
};

unsigned char NIST_TWEAK_XTS_E3[] = {
 0xcd, 0xb1, 0xbd, 0x34, 0x86, 0xf3, 0x53, 0xcc,
 0x16, 0x0a, 0x84, 0x0b, 0xea, 0xdf, 0x03, 0x29,
};

unsigned char NIST_EXPECTED_TWEAK_XTS_E3[] = {
 0xcd, 0xb1, 0xbd, 0x34, 0x86, 0xf3, 0x53, 0xcc,
 0x16, 0x0a, 0x84, 0x0b, 0xea, 0xdf, 0x03, 0x29,
};

unsigned char NIST_TEST_DATA_XTS_E3[] = {
 0x9a, 0x01, 0x49, 0x88, 0x8b, 0xf7, 0x61, 0x60,
 0xa8, 0x14, 0x28, 0xbc, 0x91, 0x40, 0xec, 0xcd,
 0x26, 0xed, 0x18, 0x36, 0x8e, 0x24, 0xd4, 0x9b,
 0x9c, 0xc5, 0x12, 0x92, 0x9a, 0x88, 0xad, 0x1e,
 0x66, 0xc7, 0x63, 0xf4, 0xf5, 0x6b, 0x63, 0xbb,
 0x9d, 0xd9, 0x50, 0x8c, 0x5d, 0x4d, 0xf4, 0x65,
 0xad, 0x98, 0x82, 0x14, 0x82, 0xfc, 0x71, 0x94,
 0xee, 0x23, 0x54, 0xa3, 0xfa, 0xdc, 0xe9, 0x23,
 0x18, 0x54, 0x8e, 0x8c, 0xe9, 0x45, 0x20, 0x81,
 0x60, 0x49, 0x7b, 0x93, 0x05, 0xd9, 0xab, 0x10,
 0x91, 0xab, 0x41, 0xd1, 0xf0, 0x9a, 0x0c, 0x7b,
 0xfa, 0xf9, 0xf9, 0x4f, 0xe7, 0xc8, 0xf1, 0xea,
 0x96, 0x8f, 0x8f, 0x9a, 0x71, 0x3a, 0xca, 0xde,
 0x18, 0xb6, 0x82, 0x32, 0x10, 0x6f, 0xfd, 0x6d,
 0x42, 0x81, 0xe9, 0x9e, 0x11, 0xd6, 0xa4, 0x28,
 0xb5, 0x16, 0x53, 0xc0, 0xc7, 0xdd, 0xe5, 0xa0,
 0xf2, 0x73, 0xe7, 0x4f, 0xf0, 0x15, 0xce, 0x80,
 0x27, 0x7d, 0x74, 0x30, 0xf5, 0xda, 0xea, 0x8f,
 0x73, 0x40, 0x64, 0x5e, 0x0b, 0xec, 0x25, 0xf4,
 0x04, 0x0f, 0xa1, 0x3c, 0x0b, 0x33, 0x06, 0x93,
```

```
 0xb1, 0x00, 0x83, 0xa8, 0xb9, 0xbc, 0x10, 0x8f,
 0xe6, 0x4f, 0x3a, 0x5b, 0x61, 0x3c, 0xbb, 0x56,
 0x5a, 0xee, 0x2f, 0x09, 0xf5, 0xb2, 0x04, 0xae,
 0xe1, 0x72, 0x28, 0xfe, 0x65, 0x31, 0xc7, 0x0c,
 0x0e, 0xc9, 0x47, 0xd2, 0xa5, 0x14, 0x7b, 0x45,
 0xc5, 0x1a, 0xc7, 0xdc, 0x8e, 0x85, 0x87, 0x03,
 0x87, 0xeb, 0x8d, 0xb6, 0x25, 0x13, 0x68, 0x36,
 0x8b, 0xf5, 0xf2, 0x46, 0xb2, 0x95, 0x7d, 0xaf,
 0xf7, 0x02, 0xe3, 0x79, 0x02, 0x2e, 0x99, 0x16,
 0x17, 0x49, 0xe6, 0xbe, 0x8e, 0xb7, 0x9d, 0x51,
 0x97, 0x99, 0xaa, 0xe0, 0x7c, 0x18, 0x31, 0xbd,
 0x0e, 0xe7, 0x25, 0x50, 0xb8, 0x53, 0x33, 0xab,
 0x9e, 0x96, 0xa5, 0x33, 0xe2, 0x97, 0x25, 0xd7,
 0x02, 0x3d, 0x82, 0x1a, 0xbe, 0x1c, 0xe3, 0xa7,
 0x44, 0xbe, 0x02, 0xe0, 0x52, 0x56, 0x8f, 0x84,
 0xe6, 0xe3, 0xf7, 0x44, 0x42, 0xbb, 0xa5, 0x0d,
 0x02, 0xad, 0x2d, 0x6c, 0xa5, 0x8a, 0x69, 0x1f,
 0xd2, 0x43, 0x9a, 0xa3, 0xaf, 0x0c, 0x03, 0x3a,
 0x68, 0xc4, 0x38, 0xb2, 0xd9, 0xa0, 0xa0, 0x1d,
 0x78, 0xc4, 0xf8, 0x7c, 0x50, 0x9f, 0xea, 0x0a,
 0x43, 0x5b, 0xe7, 0x1b, 0xa2, 0x37, 0x06, 0xd6,
 0x08, 0x2d, 0xcb, 0xa6, 0x26, 0x25, 0x99, 0x9e,
 0xce, 0x09, 0xdf, 0xb3, 0xfc, 0xbe, 0x08, 0xeb,
 0xb6, 0xf2, 0x15, 0x1e, 0x2f, 0x12, 0xeb, 0xe8,
 0xa5, 0xbf, 0x11, 0x62, 0xc2, 0x59, 0xf2, 0x02,
 0xc1, 0xba, 0x47, 0x8b, 0x5f, 0x46, 0x8a, 0x28,
 0x69, 0xf1, 0xe7, 0x6c, 0xf5, 0xed, 0x38, 0xde,
 0x53, 0x86, 0x9a, 0xdc, 0x83, 0x70, 0x9e, 0x21,
 0xb3, 0xf8, 0xdc, 0x13, 0xba, 0x3d, 0x6a, 0xa7,
 0xf6, 0xb0, 0xcf, 0xb3, 0xe5, 0xa4, 0x3c, 0x23,
 0x72, 0xe0, 0xee, 0x60, 0x99, 0x1c, 0xe1, 0xca,
 0xd1, 0x22, 0xa3, 0x1d, 0x93, 0x97, 0xe3, 0x0b,
 0x92, 0x1f, 0xd2, 0xf6, 0xee, 0x69, 0x6e, 0x68,
 0x49, 0xae, 0xee, 0x29, 0xe2, 0xb4, 0x45, 0xc0,
 0xfd, 0x9a, 0xde, 0x65, 0x56, 0xc3, 0xc0, 0x69,
 0xc5, 0xd6, 0x05, 0x95, 0xab, 0xbd, 0xf5, 0xba,
 0xe2, 0xcc, 0xc7, 0x9a, 0x49, 0x6e, 0x83, 0xcc,
 0xab, 0x95, 0x74, 0x0e, 0xb8, 0xe4, 0xf2, 0x92,
 0x5d, 0xbf, 0x72, 0x97, 0xa8, 0xc9, 0x92, 0x75,
 0x6e, 0x62, 0x87, 0x0e, 0xdc, 0xe9, 0x8f, 0x6c,
 0xba, 0x1a, 0xa0, 0xd5, 0xb8, 0x6f, 0x09, 0x21,
 0x43, 0xb1, 0x6d, 0xa1, 0x44, 0x15, 0x47, 0xd1,
 0xd4, 0x2b, 0x80, 0x06, 0xfa, 0xce, 0x69, 0x5b,
 0x03, 0xfd, 0xfa, 0xe6, 0x45, 0xf9, 0x5b, 0xd6,
};

unsigned char NIST_TEST_RESULT_XTS_E3[] = {
 0x0e, 0xee, 0xf2, 0x8c, 0xa1, 0x59, 0xb8, 0x05,
 0xf5, 0xc2, 0x15, 0x61, 0x05, 0x51, 0x67, 0x8a,
 0xb7, 0x72, 0xf2, 0x79, 0x37, 0x4f, 0xb1, 0x40,
 0xab, 0x55, 0x07, 0x68, 0xdb, 0x42, 0xcf, 0x6c,
 0xb7, 0x36, 0x37, 0x64, 0x19, 0x34, 0x19, 0x5f,
 0xfc, 0x08, 0xcf, 0x5a, 0x91, 0x88, 0xb8, 0x2b,
 0x84, 0x0a, 0x00, 0x7d, 0x52, 0x72, 0x39, 0xea,
 0x3f, 0x0d, 0x7d, 0xd1, 0xf2, 0x51, 0x86, 0xec,
 0xae, 0x30, 0x87, 0x7d, 0xad, 0xa7, 0x7f, 0x24,
 0x3c, 0xdd, 0xb2, 0xc8, 0x8e, 0x99, 0x04, 0x82,
 0x7d, 0x3e, 0x09, 0x82, 0xda, 0x0d, 0x13, 0x91,
 0x1d, 0x0e, 0x2d, 0xbb, 0xbb, 0x2d, 0x01, 0x6c,
 0xbe, 0x4d, 0x06, 0x76, 0xb1, 0x45, 0x9d, 0xa8,
 0xc5, 0x3a, 0x91, 0x45, 0xe8, 0x3c, 0xf4, 0x2f,
 0x30, 0x11, 0x2c, 0xa6, 0x5d, 0x77, 0xc8, 0x93,
 0x4a, 0x26, 0xee, 0x00, 0x1f, 0x39, 0x0f, 0xfc,
 0xc1, 0x87, 0x03, 0x66, 0x2a, 0x8f, 0x71, 0xf9,
 0xda, 0x0e, 0x7b, 0x68, 0xb1, 0x04, 0x3c, 0x1c,
 0xb5, 0x26, 0x08, 0xcf, 0x0e, 0x69, 0x51, 0x0d,
 0x38, 0xc8, 0x0f, 0xa0, 0x0d, 0xe4, 0x3d, 0xef,
```

```
 0x98, 0x4d, 0xff, 0x2f, 0x32, 0x4e, 0xcf, 0x39,
 0x89, 0x44, 0x53, 0xd3, 0xe0, 0x1b, 0x3d, 0x7b,
 0x3b, 0xc0, 0x57, 0x04, 0x9d, 0x19, 0x5c, 0x8e,
 0xb9, 0x3f, 0xe4, 0xd9, 0x5a, 0x83, 0x00, 0xa5,
 0xe6, 0x0a, 0x7c, 0x89, 0xe4, 0x0c, 0x69, 0x16,
 0x79, 0xfb, 0xca, 0xfa, 0xd8, 0xeb, 0x41, 0x8f,
 0x8d, 0x1f, 0xf7, 0xb9, 0x11, 0x75, 0xf8, 0xeb,
 0x3c, 0x6f, 0xf2, 0x87, 0x2d, 0x32, 0xee, 0x4c,
 0x57, 0x36, 0x9e, 0x61, 0xb6, 0x6d, 0x16, 0x6f,
 0xd0, 0xa4, 0x34, 0x57, 0x47, 0x82, 0x75, 0xfe,
 0x14, 0xbf, 0x34, 0x63, 0x8a, 0x9e, 0x4e, 0x1d,
 0x25, 0xcc, 0x5a, 0x5f, 0x9e, 0x25, 0x7e, 0x61,
 0x7a, 0xdc, 0xdd, 0xe6, 0x5e, 0x25, 0x57, 0x40,
 0x53, 0x62, 0xc8, 0x91, 0xe6, 0x54, 0x6a, 0x6d,
 0xee, 0xaa, 0x8f, 0xc0, 0x3b, 0x12, 0x2a, 0x55,
 0x87, 0x4d, 0x33, 0xe0, 0xa7, 0x73, 0x52, 0x34,
 0x68, 0x32, 0x5e, 0xc2, 0x4d, 0x4f, 0xaf, 0xfb,
 0x63, 0xc0, 0x52, 0xc8, 0x11, 0xa1, 0xc0, 0x22,
 0xba, 0xfc, 0xcb, 0x97, 0x98, 0x8b, 0x7e, 0x45,
 0x67, 0xb2, 0x47, 0xd4, 0x04, 0x4b, 0x05, 0x2f,
 0xf7, 0x3f, 0x4c, 0x67, 0x1d, 0x27, 0xe0, 0x52,
 0xe2, 0xeb, 0xc7, 0x2d, 0x00, 0x57, 0xcb, 0x21,
 0x7c, 0x52, 0x59, 0xb6, 0x09, 0x50, 0xe3, 0xc8,
 0xb3, 0xd9, 0xe3, 0xe7, 0x63, 0x0f, 0x9e, 0xcb,
 0xe5, 0x48, 0xb9, 0xe3, 0x62, 0x20, 0xf3, 0x3c,
 0x2b, 0x45, 0x68, 0x30, 0x7c, 0xd0, 0x37, 0x5b,
 0xba, 0x13, 0x35, 0xe5, 0x8b, 0xfb, 0xcd, 0xe8,
 0x5c, 0xc8, 0x4c, 0x9c, 0x9c, 0x1c, 0xe7, 0x4f,
 0x44, 0xb2, 0x8e, 0xa1, 0xb6, 0x97, 0x30, 0x5b,
 0xb6, 0xba, 0x3b, 0x46, 0x4e, 0x5a, 0xb7, 0x45,
 0x01, 0x29, 0x3e, 0xf9, 0x15, 0x2c, 0x0f, 0x5d,
 0x33, 0x07, 0xd2, 0x6a, 0x1f, 0x07, 0x41, 0xc5,
 0xe5, 0x72, 0x1a, 0x71, 0x3d, 0x1b, 0x86, 0xc1,
 0x80, 0x82, 0x11, 0xf5, 0x7a, 0xad, 0x09, 0xa9,
 0x50, 0xb6, 0x86, 0x30, 0xaf, 0xce, 0x4f, 0x0a,
 0xd9, 0xf3, 0x2e, 0x67, 0x69, 0xb5, 0xfe, 0x31,
 0x92, 0x9c, 0x44, 0x6f, 0x7a, 0x33, 0x55, 0xf4,
 0x58, 0x84, 0xc7, 0x48, 0xc9, 0x05, 0x54, 0x15,
 0xe6, 0x37, 0xd9, 0xad, 0x87, 0xd9, 0x4c, 0x46,
 0x57, 0xb1, 0xad, 0x03, 0x4c, 0xb1, 0x4d, 0x9a,
 0x72, 0xea, 0x74, 0x5f, 0xe5, 0x2d, 0x7a, 0x71,
 0x1b, 0xa4, 0x1c, 0xa0, 0x35, 0x85, 0x6a, 0x5a,
 0x44, 0x89, 0xa4, 0x27, 0x0b, 0xb3, 0x0d, 0x5b,
 0x63, 0xf4, 0x9c, 0x05, 0x12, 0xfe, 0xd4, 0xb4
};

/* XTS data -4- AES256 */
unsigned char NIST_KEY_XTS_E4[] = {
 0x97, 0x09, 0x8b, 0x46, 0x5a, 0x44, 0xca, 0x75,
 0xe7, 0xa1, 0xc2, 0xdb, 0xfc, 0x40, 0xb7, 0xa6,
 0x1a, 0x20, 0xe3, 0x2c, 0x6d, 0x9d, 0xbf, 0xda,
 0x80, 0x72, 0x6f, 0xee, 0x10, 0x54, 0x1b, 0xab,
 0x47, 0x54, 0x63, 0xca, 0x07, 0xc1, 0xc1, 0xe4,
 0x49, 0x61, 0x73, 0x32, 0x14, 0x68, 0xd1, 0xab,
 0x3f, 0xad, 0x8a, 0xd9, 0x1f, 0xcd, 0xc6, 0x2a,
 0xbe, 0x07, 0xbf, 0xf8, 0xef, 0x96, 0x1b, 0x6b,
};

unsigned char NIST_TWEAK_XTS_E4[] = {
 0x15, 0x60, 0x1e, 0x2e, 0x35, 0x85, 0x10, 0xa0,
 0x9d, 0xdc, 0xa4, 0xea, 0x17, 0x51, 0xf4, 0x3c,
};

unsigned char NIST_EXPECTED_TWEAK_XTS_E4[] = {
 0x15, 0x60, 0x1e, 0x2e, 0x35, 0x85, 0x10, 0xa0,
 0x9d, 0xdc, 0xa4, 0xea, 0x17, 0x51, 0xf4, 0x3c,
};
```

```
unsigned char NIST_TEST_DATA_XTS_E4[] = {
 0xd1, 0x9c, 0xfb, 0x38, 0x3b, 0xaf, 0x87, 0x2e,
 0x6f, 0x12, 0x16, 0x87, 0x45, 0x1d, 0xe1, 0x5c,
};

unsigned char NIST_TEST_RESULT_XTS_E4[] = {
 0xeb, 0x22, 0x26, 0x9b, 0x14, 0x90, 0x50, 0x27,
 0xdc, 0x73, 0xc4, 0xa4, 0x0f, 0x93, 0x80, 0x69,
};


/* XTS data -5- AES256 */
unsigned char NIST_KEY_XTS_E5[] = {
 0xfb, 0xf0, 0x77, 0x6e, 0x7d, 0xbe, 0x49, 0x10,
 0xfb, 0x0c, 0x12, 0x0f, 0x41, 0x85, 0x71, 0x21,
 0x92, 0x6c, 0x05, 0x2f, 0xd6, 0x5a, 0x27, 0x8c,
 0xd2, 0xf0, 0xd9, 0x8d, 0xa5, 0x4e, 0xdf, 0xd5,
 0x08, 0x03, 0xa4, 0x2f, 0xbe, 0x6f, 0xd1, 0x33,
 0x58, 0x49, 0x00, 0xe8, 0xdc, 0x7a, 0x11, 0x52,
 0x39, 0x1f, 0x82, 0x2d, 0x76, 0xa7, 0x56, 0x68,
 0xcf, 0xce, 0x7f, 0x8d, 0xde, 0x20, 0x3e, 0xc8,
};

unsigned char NIST_TWEAK_XTS_E5[] = {
 0x39, 0x5b, 0x6a, 0xcf, 0x9a, 0xdc, 0xd2, 0x91,
 0xc2, 0xc9, 0x48, 0x86, 0x36, 0x33, 0xaf, 0xf8,
};

unsigned char NIST_EXPECTED_TWEAK_XTS_E5[] = {
 0x39, 0x5b, 0x6a, 0xcf, 0x9a, 0xdc, 0xd2, 0x91,
 0xc2, 0xc9, 0x48, 0x86, 0x36, 0x33, 0xaf, 0xf8,
};

unsigned char NIST_TEST_DATA_XTS_E5[] = {
 0x3e, 0x2e, 0x26, 0x9d, 0x78, 0x3a, 0x2b, 0x29,
 0xe8, 0x73, 0xd6, 0x73, 0x47, 0x9f, 0x51, 0x16,
 0x73, 0x4f, 0xe0, 0x3e, 0xe3, 0x29, 0x65, 0xed,
 0xc4, 0x79, 0x35, 0xc0, 0xea, 0x99, 0xa0, 0x64,
 0xbd, 0x44, 0x4b, 0xec, 0x12, 0x5b, 0x2c, 0x78,
 0x9d, 0xb9, 0xde, 0x6d, 0x18, 0x35, 0x92, 0x05,
 0x3b, 0x48, 0xa8, 0x77, 0xa9, 0x5a, 0xc2, 0x55,
 0x9c, 0x3d, 0xdf, 0xc7, 0xb4, 0xdb, 0x99, 0x07,
};

unsigned char NIST_TEST_RESULT_XTS_E5[] = {
 0x4c, 0x70, 0xbd, 0xbb, 0x77, 0x30, 0x2b, 0x7f,
 0x1f, 0xdd, 0xca, 0x50, 0xdc, 0x70, 0x73, 0x1e,
 0x00, 0x8a, 0x26, 0x55, 0xd2, 0x2a, 0xd0, 0x20,
 0x0c, 0x11, 0x1f, 0xd3, 0x2a, 0x67, 0x5a, 0x7e,
 0x09, 0x97, 0x11, 0x43, 0x6f, 0x98, 0xd2, 0x1c,
 0x72, 0x77, 0x2e, 0x0d, 0xd7, 0x67, 0x2f, 0xf5,
 0xfd, 0x00, 0xdd, 0xcb, 0xe1, 0x1e, 0xb9, 0x7e,
 0x69, 0x87, 0x83, 0xbf, 0xa4, 0x05, 0x46, 0xe3,
};

void dump_array(unsigned char *ptr, unsigned int size)
{
 unsigned char *ptr_end;
 unsigned char *h;
 int i = 1;

 h = ptr;
 ptr_end = ptr + size;
 while (h < (unsigned char *)ptr_end) {
  printf("0x%02x ",(unsigned char ) *h);
  h++;
```

```
  if (i == 8) {
   printf("\n");
   i = 1;
  } else {
   ++i;
  }
 }
 printf("\n");
}

void dump_xts_data(unsigned char *tweak, unsigned int tweak_length,
                   unsigned char *key, unsigned int key_length,
                   unsigned char *input_data, unsigned int data_length,
                   unsigned char *output_data)
{
 printf("TWEAK \n");
 dump_array(tweak, tweak_length);
 printf("Key \n");
 dump_array(key, key_length);
 printf("Input Data\n");
 dump_array(input_data, data_length);
 printf("Output Data\n");
 dump_array(output_data, data_length);
}

void get_sizes(unsigned int *data_length, unsigned int *tweak_length,
        unsigned int *key_length, unsigned int iteration)
{
 switch (iteration) {
  case 1:
   *data_length = sizeof(NIST_TEST_DATA_XTS_E1);
   *tweak_length = sizeof(NIST_TWEAK_XTS_E1);
   *key_length = sizeof(NIST_KEY_XTS_E1);
   break;
  case 2:
   *data_length = sizeof(NIST_TEST_DATA_XTS_E2);
   *tweak_length = sizeof(NIST_TWEAK_XTS_E2);
   *key_length = sizeof(NIST_KEY_XTS_E2);
   break;
  case 3:
   *data_length = sizeof(NIST_TEST_DATA_XTS_E3);
   *tweak_length = sizeof(NIST_TWEAK_XTS_E3);
   *key_length = sizeof(NIST_KEY_XTS_E3);
   break;
  case 4:
   *data_length = sizeof(NIST_TEST_DATA_XTS_E4);
   *tweak_length = sizeof(NIST_TWEAK_XTS_E4);
   *key_length = sizeof(NIST_KEY_XTS_E4);
   break;
  case 5:
   *data_length = sizeof(NIST_TEST_DATA_XTS_E5);
   *tweak_length = sizeof(NIST_TWEAK_XTS_E5);
   *key_length = sizeof(NIST_KEY_XTS_E5);
   break;
 }

}

void load_test_data(unsigned char *data, unsigned int data_length,
      unsigned char *result,
      unsigned char *tweak, unsigned char *expected_tweak,
      unsigned int tweak_length,
      unsigned char *key, unsigned int key_length,
      unsigned int iteration)
{
 switch (iteration) {
  case 1:
```

```
      memcpy(data, NIST_TEST_DATA_XTS_E1, data_length);
      memcpy(result, NIST_TEST_RESULT_XTS_E1, data_length);
      memcpy(tweak, NIST_TWEAK_XTS_E1, tweak_length);
      memcpy(expected_tweak, NIST_EXPECTED_TWEAK_XTS_E1,
             tweak_length);
      memcpy(key, NIST_KEY_XTS_E1, key_length);
      break;
    case 2:
      memcpy(data, NIST_TEST_DATA_XTS_E2, data_length);
      memcpy(result, NIST_TEST_RESULT_XTS_E2, data_length);
      memcpy(tweak, NIST_TWEAK_XTS_E2, tweak_length);
      memcpy(expected_tweak, NIST_EXPECTED_TWEAK_XTS_E2,
             tweak_length);
      memcpy(key, NIST_KEY_XTS_E2, key_length);
      break;
    case 3:
      memcpy(data, NIST_TEST_DATA_XTS_E3, data_length);
      memcpy(result, NIST_TEST_RESULT_XTS_E3, data_length);
      memcpy(tweak, NIST_TWEAK_XTS_E3, tweak_length);
      memcpy(expected_tweak, NIST_EXPECTED_TWEAK_XTS_E3,
             tweak_length);
      memcpy(key, NIST_KEY_XTS_E3, key_length);
      break;
    case 4:
      memcpy(data, NIST_TEST_DATA_XTS_E4, data_length);
      memcpy(result, NIST_TEST_RESULT_XTS_E4, data_length);
      memcpy(tweak, NIST_TWEAK_XTS_E4, tweak_length);
      memcpy(expected_tweak, NIST_EXPECTED_TWEAK_XTS_E4,
             tweak_length);
      memcpy(key, NIST_KEY_XTS_E4, key_length);
      break;
    case 5:
      memcpy(data, NIST_TEST_DATA_XTS_E5, data_length);
      memcpy(result, NIST_TEST_RESULT_XTS_E5, data_length);
      memcpy(tweak, NIST_TWEAK_XTS_E5, tweak_length);
      memcpy(expected_tweak, NIST_EXPECTED_TWEAK_XTS_E5,
             tweak_length);
      memcpy(key, NIST_KEY_XTS_E5, key_length);
      break;
  }

}

int kat_aes_xts(int iteration, int silent)
{
 unsigned int data_length;
 unsigned int tweak_length;
 unsigned int key_length;

 get_sizes(&data_length, &tweak_length, &key_length, iteration);

 unsigned char tweak[tweak_length];
 unsigned char tmp_tweak[tweak_length];
 unsigned char expected_tweak[tweak_length];
 unsigned char key[key_length];
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];
 unsigned char result[data_length];

 int rc = 0;
 memset(encrypt, 0x00, data_length);
 memset(decrypt, 0x00, data_length);

 load_test_data(input_data, data_length, result, tweak, expected_tweak,
         tweak_length, key, key_length, iteration);
 memcpy(tmp_tweak, tweak, tweak_length);
```

```
printf("Test Parameters for iteration = %i\n", iteration);
printf("key length = %i, data length = %i, tweak length = %i,",
        key_length, data_length, tweak_length);

rc = ica_aes_xts(input_data, encrypt, data_length,
   key, key+(key_length/2), (key_length/2),
   tmp_tweak, 1);
if (rc) {
 printf("ica_aes_xts encrypt failed with rc = %i\n", rc);
 dump_xts_data(tweak, tweak_length, key, key_length, input_data,
        data_length, encrypt);
}
if (!silent && !rc) {
 printf("Encrypt:\n");
 dump_xts_data(tweak, tweak_length, key, key_length, input_data,
        data_length, encrypt);
}

if (memcmp(result, encrypt, data_length)) {
 printf("Encryption Result does not match the known ciphertext!\n");
 printf("Expected data:\n");
 dump_array(result, data_length);
 printf("Encryption Result:\n");
 dump_array(encrypt, data_length);
 rc++;
}

if (memcmp(expected_tweak, tmp_tweak, tweak_length)) {
 printf("Update of TWEAK does not match the expected TWEAK!\n");
 printf("Expected TWEAK:\n");
 dump_array(expected_tweak, tweak_length);
 printf("Updated TWEAK:\n");
 dump_array(tmp_tweak, tweak_length);
 printf("Original TWEAK:\n");
 dump_array(tweak, tweak_length);
 rc++;
}
if (rc) {
 printf("AES XTS test exited after encryption\n");
 return rc;
}

memcpy(tmp_tweak, tweak, tweak_length);
rc = ica_aes_xts(encrypt, decrypt, data_length,
   key, key+(key_length/2), (key_length/2),
   tmp_tweak, 0);
if (rc) {
 printf("ica_aes_xts decrypt failed with rc = %i\n", rc);
 dump_xts_data(tweak, tweak_length, key, key_length, encrypt,
        data_length, decrypt);
 return rc;
}


if (!silent && !rc) {
 printf("Decrypt:\n");
 dump_xts_data(tweak, tweak_length, key, key_length, encrypt,
        data_length, decrypt);
}

if (memcmp(decrypt, input_data, data_length)) {
 printf("Decryption Result does not match the original data!\n");
 printf("Original data:\n");
 dump_array(input_data, data_length);
 printf("Decryption Result:\n");
 dump_array(decrypt, data_length);
```

```
  rc++;
 }
 return rc;
}

int load_random_test_data(unsigned char *data, unsigned int data_length,
          unsigned char *iv, unsigned int iv_length,
          unsigned char *key, unsigned int key_length)
{
 int rc;
 rc = ica_random_number_generate(data_length, data);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
         rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(iv_length, iv);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
         rc, errno);
  return rc;
 }
 rc = ica_random_number_generate(key_length, key);
 if (rc) {
  printf("ica_random_number_generate with rc = %i errnor = %i\n",
         rc, errno);
  return rc;
 }
 return rc;
}

int random_aes_xts(int iteration, int silent, unsigned int data_length)
{
 int i;
 int rc = 0;
 unsigned int iv_length = sizeof(ica_aes_vector_t);
 unsigned int key_length = AES_KEY_LEN128 * 2;
 unsigned char iv[iv_length];
 unsigned char tmp_iv[iv_length];
 unsigned char input_data[data_length];
 unsigned char encrypt[data_length];
 unsigned char decrypt[data_length];
for (i = 1; i <= 2; i++) {

 unsigned char key[key_length];

 memset(encrypt, 0x00, data_length);
 memset(decrypt, 0x00, data_length);

 load_random_test_data(input_data, data_length, iv, iv_length, key,
         key_length);
 memcpy(tmp_iv, iv, iv_length);
 printf("Test Parameters for iteration = %i\n", iteration);
 printf("key length = %i, data length = %i, iv length = %i\n",
        key_length, data_length, iv_length);

 rc = ica_aes_xts(input_data, encrypt, data_length,
    key, key+(key_length/2), (key_length/2),
    tmp_iv, 1);
 if (rc) {
  printf("ica_aes_xts encrypt failed with rc = %i\n", rc);
  dump_xts_data(iv, iv_length, key, key_length, input_data,
         data_length, encrypt);
 }
 if (!silent && !rc) {
  printf("Encrypt:\n");
  dump_xts_data(iv, iv_length, key, key_length, input_data,
```

```
        data_length, encrypt);
 }

 if (rc) {
  printf("AES XTS test exited after encryption\n");
  return rc;
 }

 memcpy(tmp_iv, iv, iv_length);

 rc = ica_aes_xts(encrypt, decrypt, data_length,
    key, key+(key_length/2), (key_length/2),
    tmp_iv, 0);
 if (rc) {
  printf("ica_aes_xts decrypt failed with rc = %i\n", rc);
  dump_xts_data(iv, iv_length, key, key_length, encrypt,
        data_length, decrypt);
  return rc;
 }


 if (!silent && !rc) {
  printf("Decrypt:\n");
  dump_xts_data(iv, iv_length, key, key_length, encrypt,
        data_length, decrypt);
 }

 if (memcmp(decrypt, input_data, data_length)) {
  printf("Decryption Result does not match the original data!\n");
  printf("Original data:\n");
  dump_array(input_data, data_length);
  printf("Decryption Result:\n");
  dump_array(decrypt, data_length);
  rc++;
  return rc;
 }
 key_length = AES_KEY_LEN256 * 2;
}

 return rc;
}

int main(int argc, char **argv)
{
 unsigned int silent = 0;
 if (argc > 1) {
  if (strstr(argv[1], "silent"))
   silent = 1;
 }
 int rc = 0;
 int error_count = 0;
 int iteration;
 unsigned int data_length = sizeof(ica_aes_vector_t);
 for(iteration = 1; iteration <= NR_TESTS; iteration++) {
  rc = kat_aes_xts(iteration, silent);
  if (rc) {
   printf("kat_aes_xts failed with rc = %i\n", rc);
   error_count++;
  } else
   printf("kat_aes_xts finished successfuly\n");

 }
 for(iteration = 1; iteration <= NR_RANDOM_TESTS; iteration++) {
  int silent = 1;
  rc = random_aes_xts(iteration, silent, data_length);
  if (rc) {
   printf("random_aes_xts failed with rc = %i\n", rc);
```

```
   error_count++;
   goto out;
  } else
   printf("random_aes_xts finished successfuly\n");
  data_length += sizeof(ica_aes_vector_t) / 2;
 }

out:
 if (error_count)
  printf("%i testcases failed\n", error_count);
 else
  printf("All testcases finished successfully\n");

 return rc;
}
```

## CMAC example

```
/* This program is released under the Common Public License V1.0
 *
 * You should have received a copy of Common Public License V1.0 along with
 * with this program.
 */

/* Copyright IBM Corp. 2010, 2011 */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ica_api.h"

#define BYTE 8

#define NUM_TESTS 12

unsigned int key_length[12] = {16, 16, 16, 16, 24, 24, 24, 24, 32, 32, 32,
    32};
unsigned char key[12][32] = {{
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15,
 0x88, 0x09, 0xcf, 0x4f, 0x3c},{
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15,
 0x88, 0x09, 0xcf, 0x4f, 0x3c},{
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15,
 0x88, 0x09, 0xcf, 0x4f, 0x3c},{
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15,
 0x88, 0x09, 0xcf, 0x4f, 0x3c},{
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10 ,0xf3,
 0x2b, 0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c,
 0x6b, 0x7b},{
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10 ,0xf3,
 0x2b, 0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c,
 0x6b, 0x7b},{
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10 ,0xf3,
 0x2b, 0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c,
 0x6b, 0x7b},{
 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8, 0x10 ,0xf3,
 0x2b, 0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c,
 0x6b, 0x7b},{
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae,
 0xf0, 0x85, 0x7d, 0x77, 0x81, 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61,
 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4},{
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae,
 0xf0, 0x85, 0x7d, 0x77, 0x81, 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61,
 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4},{
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae,
 0xf0, 0x85, 0x7d, 0x77, 0x81, 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61,
```

```
 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4},{
 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b, 0x73, 0xae,
 0xf0, 0x85, 0x7d, 0x77, 0x81, 0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61,
 0x08, 0xd7, 0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4}
};

unsigned char last_block[3][16] = {{
 0x7d, 0xf7, 0x6b, 0x0c, 0x1a, 0xb8, 0x99, 0xb3, 0x3e, 0x42, 0xf0,
 0x47, 0xb9, 0x1b, 0x54, 0x6f},{
 0x22, 0x45, 0x2d, 0x8e, 0x49, 0xa8, 0xa5, 0x93, 0x9f, 0x73, 0x21,
 0xce, 0xea, 0x6d, 0x51, 0x4b},{
 0xe5, 0x68, 0xf6, 0x81, 0x94, 0xcf, 0x76, 0xd6, 0x17, 0x4d, 0x4c,
 0xc0, 0x43, 0x10, 0xa8, 0x54}
};

unsigned long mlen[12] = { 0, 16, 40, 64, 0,16, 40, 64, 0, 16, 40, 64};
unsigned char message[12][512] = {{
 0x00},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a, 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03,
 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf ,0x8e, 0x51, 0x30,
 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a, 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03,
 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51, 0x30,
 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb, 0xc1, 0x19,
 0x1a, 0x0a, 0x52, 0xef, 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b,
 0x17, 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10},{
 0x00},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a, 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03,
 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf ,0x8e, 0x51, 0x30,
 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a, 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03,
 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51, 0x30,
 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb, 0xc1, 0x19,
 0x1a, 0x0a, 0x52, 0xef, 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b,
 0x17, 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10},{
 0x00},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a, 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03,
 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf ,0x8e, 0x51, 0x30,
 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11},{
 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e,
 0x11, 0x73, 0x93, 0x17, 0x2a, 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03,
 0xac, 0x9c, 0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51, 0x30,
 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb, 0xc1, 0x19,
 0x1a, 0x0a, 0x52, 0xef, 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b,
 0x17, 0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10}
};

unsigned char expected_cmac[12][16] = {{
 0xbb, 0x1d, 0x69, 0x29, 0xe9, 0x59, 0x37, 0x28, 0x7f, 0xa3, 0x7d,
 0x12, 0x9b, 0x75, 0x67, 0x46},{
 0x07, 0x0a, 0x16, 0xb4, 0x6b, 0x4d, 0x41, 0x44, 0xf7, 0x9b, 0xdd,
 0x9d, 0xd0, 0x4a, 0x28, 0x7c},{
 0xdf, 0xa6, 0x67, 0x47, 0xde, 0x9a, 0xe6, 0x30, 0x30, 0xca, 0x32,
 0x61, 0x14, 0x97, 0xc8, 0x27},{
 0x51, 0xf0, 0xbe, 0xbf, 0x7e, 0x3b, 0x9d, 0x92, 0xfc, 0x49, 0x74,
 0x17, 0x79, 0x36, 0x3c, 0xfe},{
```

```
 0xd1, 0x7d, 0xdf, 0x46, 0xad, 0xaa, 0xcd, 0xe5, 0x31, 0xca, 0xc4,
 0x83, 0xde, 0x7a, 0x93, 0x67},{
 0x9e, 0x99, 0xa7, 0xbf, 0x31, 0xe7, 0x10, 0x90, 0x06, 0x62, 0xf6,
 0x5e, 0x61, 0x7c, 0x51, 0x84},{
 0x8a, 0x1d, 0xe5, 0xbe, 0x2e, 0xb3, 0x1a, 0xad, 0x08, 0x9a, 0x82,
 0xe6, 0xee, 0x90, 0x8b, 0x0e},{
 0xa1, 0xd5, 0xdf, 0x0e, 0xed, 0x79, 0x0f, 0x79, 0x4d, 0x77, 0x58,
 0x96, 0x59, 0xf3, 0x9a, 0x11},{
 0x02, 0x89, 0x62, 0xf6, 0x1b, 0x7b, 0xf8, 0x9e, 0xfc, 0x6b, 0x55,
 0x1f, 0x46, 0x67, 0xd9, 0x83},{
 0x28, 0xa7, 0x02, 0x3f, 0x45, 0x2e, 0x8f, 0x82, 0xbd, 0x4b, 0xf2,
 0x8d, 0x8c, 0x37, 0xc3, 0x5c},{
 0xaa, 0xf3, 0xd8, 0xf1, 0xde, 0x56, 0x40, 0xc2, 0x32, 0xf5, 0xb1,
 0x69, 0xb9, 0xc9, 0x11, 0xe6},{
 0xe1, 0x99, 0x21, 0x90, 0x54, 0x9f, 0x6e, 0xd5, 0x69, 0x6a, 0x2c,
 0x05, 0x6c, 0x31, 0x54, 0x10}
};

unsigned int i = 0;

void dump_array(unsigned char *ptr, unsigned int size)
{
 unsigned char *ptr_end;
 unsigned char *h;
 int i = 1, trunc = 0;
 int maxsize = 2000;

 puts("Dump:");

 if (size > maxsize) {
  trunc = size - maxsize;
  size = maxsize;
 }
 h = ptr;
 ptr_end = ptr + size;
 while (h < ptr_end) {
  printf("0x%02x ", *h);
  h++;
  if (i == 16) {
   if (h != ptr_end)
   printf("\n");
   i = 1;
  } else {
   ++i;
  }
 }
 printf("\n");
 if (trunc > 0)
 printf("... %d bytes not printed\n", trunc);
}
unsigned char *cmac;
unsigned int cmac_length = 16;

int api_cmac_test(void)
{
 printf("Test of CMAC api\n");
 int rc = 0;
 for (i = 0 ; i < NUM_TESTS; i++) {
  if (!(cmac = malloc(cmac_length)))
   return EINVAL;
  memset(cmac, 0, cmac_length);
  rc = (ica_aes_cmac(message[i], mlen[i],
       cmac, cmac_length,
       key[i], key_length[i],
       ICA_ENCRYPT));
  if (rc) {
   printf("ica_aes_cmac generate failed with errno %d (0x%x)."
```

```
    "\n",rc,rc);
   return rc;
  }
  if (memcmp(cmac, expected_cmac[i], cmac_length) != 0) {
   printf("This does NOT match the known result. "
    "Testcase %i failed\n",i);
   printf("\nOutput MAC for test %d:\n", i);
   dump_array((unsigned char *)cmac, cmac_length);
   printf("\nExpected MAC for test %d:\n", i);
   dump_array((unsigned char *)expected_cmac[i], 16);
   free(cmac);
   return 1;
  }
  printf("Expected MAC has been generated.\n");
  rc = (ica_aes_cmac(message[i], mlen[i],
       cmac, cmac_length,
       key[i], key_length[i],
       ICA_DECRYPT));
  if (rc) {
   printf("ica_aes_cmac verify failed with errno %d (0x%x).\n",
    rc, rc);
   free(cmac);
   return rc;
  }
  free(cmac);
  if (! rc )
   printf("MAC was successful verified. testcase %i "
    "succeeded\n",i);
  else {
   printf("MAC verification failed for testcase %i "
    "with RC=%i\n",i,rc);
   return rc;
  }
 }
 return 0;
}

int main(int argc, char **argv)
{
 int rc = 0;

 rc = api_cmac_test();
 if (rc) {
  printf("api_cmac_test failed with rc = %i\n", rc);
  return rc;
 }
 printf("api_cmac_test was succesful\n");
 return 0;
}
```

## Makefile example

```
# Specify include directory. Leave blank for default system location.
INCDIR =

# Specify library directory. Leave blank for default system location.
LIBDIR =

# Specify library.
LIBS = -lica

TARGETS = example_des_ecb

all: $(TARGETS)

%: %.c
```

```
 gcc $(INCDIR) $(LIBDIR) $(LIBS) -o $@ $^

clean:
 rm -f $(TARGETS)
```

## Common Public License - V1.0

```
Common Public License - V1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON
PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF
THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:
 1. in the case of the initial Contributor, the initial code and
 documentation distributed under this Agreement, and

 2. in the case of each subsequent Contributor:
        1. changes to the Program, and
        2. additions to the Program;

 where such changes and/or additions to the Program originate
 from and are distributed by that particular Contributor. A
 Contribution 'originates' from a Contributor if it was added to
 the Program by such Contributor itself or anyone acting on such
 Contributor's behalf. Contributions do not include additions to
 the Program which: (i) are separate modules of software
 distributed in conjunction with the Program under their own
 license agreement, and (ii) are not derivative works of the
 Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor
which are necessarily infringed by the use or sale of its Contribution
alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this
Agreement.

"Recipient" means anyone who receives the Program under this Agreement,
including all Contributors.


2. GRANT OF RIGHTS

 1. Subject to the terms of this Agreement, each Contributor
 hereby grants Recipient a non-exclusive, worldwide,
 royalty-free copyright license to reproduce, prepare derivative
 works of, publicly display, publicly perform, distribute and
 sublicense the Contribution of such Contributor, if any, and
 such derivative works, in source code and object code form.

 2. Subject to the terms of this Agreement, each Contributor
 hereby grants Recipient a non-exclusive, worldwide,
 royalty-free patent license under Licensed Patents to make,
 use, sell, offer to sell, import and otherwise transfer the
 Contribution of such Contributor, if any, in source code and
 object code form. This patent license shall apply to the
 combination of the Contribution and the Program if, at the time
 the Contribution is added by the Contributor, such addition of
 the Contribution causes such combination to be covered by the
 Licensed Patents. The patent license shall not apply to any
 other combinations which include the Contribution. No hardware
 per se is licensed hereunder.
```

3. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

4. Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

1. it complies with the terms and conditions of this Agreement; and

2. its license agreement:

   1. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

   2. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

   3. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

   4. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

1. it must be made available under this Agreement; and
2. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the

```
minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with
respect to a patent applicable to software (including a cross-claim or
counterclaim in a lawsuit), then any patent licenses granted by that
Contributor to such Recipient under this Agreement shall terminate as
of the date such litigation is filed. In addition, if Recipient
institutes patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Program
itself (excluding combinations of the Program with other software or
hardware) infringes such Recipient's patent(s), then such Recipient's
rights granted under Section 2(b) shall terminate as of the date such
litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails
to comply with any of the material terms or conditions of this
Agreement and does not cure such failure in a reasonable period of time
after becoming aware of such noncompliance. If all Recipient's rights
under this Agreement terminate, Recipient agrees to cease use and
distribution of the Program as soon as reasonably practicable. However,
Recipient's obligations under this Agreement and any licenses granted
by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement,
but in order to avoid inconsistency the Agreement is copyrighted and
may only be modified in the following manner. The Agreement Steward
reserves the right to publish new versions (including revisions) of
this Agreement from time to time. No one other than the Agreement
Steward has the right to modify this Agreement. IBM is the initial
Agreement Steward. IBM may assign the responsibility to serve as the
Agreement Steward to a suitable separate entity. Each new version of
the Agreement will be given a distinguishing version number. The
Program (including Contributions) may always be distributed subject to
the version of the Agreement under which it was received. In addition,
after a new version of the Agreement is published, Contributor may
elect to distribute the Program (including its Contributions) under the
new version. Except as expressly stated in Sections 2(a) and 2(b)
above, Recipient receives no rights or licenses to the intellectual
property of any Contributor under this Agreement, whether expressly, by
implication, estoppel or otherwise. All rights in the Program not
expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the
intellectual property laws of the United States of America. No party to
this Agreement will bring a legal action under this Agreement more than
one year after the cause of action arose. Each party waives its rights
to a jury trial in any resulting litigation.
```

## openCryptoki code samples

This section provides the following code samples:
- "Dynamic library call" on page 155
- "Shared linked library" on page 155

### Coding samples (C)

To develop an application that uses openCryptoki, you need to access the library.

There are two ways to access the library:
- Load shared objects using dynamic library calls (dlopen)
- Link the library(statically) to your application during built time

For a list of supported mechanisms for ica-token, refer to “Supported mechanisms for the ica token” on page 70.

## Dynamic library call

openCryptoki code samples for a dynamic library call.

```
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <dlfcn.h>
#include <pkcs11types.h>

CK_RV init();
CK_RV cleanup();
CK_RV rc;                                  /* return code */
void *dllPtr, (*symPtr)();                 /* pointer to the ock library */
CK_FUNCTION_LIST_PTR  FunctionPtr = NULL;  /* pointer to function list */

int main(int argc, char *argv[]){
  init("/usr/lib64/opencryptoki/libopencryptoki.so"); /* opencryptoki initialization */
  /* .... other opencryptoki commands.... */
  cleanup();       /* cleanup/close shared library */
  return 0;
}

CK_RV init(char *libPath){

  dllPtr = dlopen(libPath, RTLD_NOW);      /* open the PKCS11 library */
  if (!dllPtr) {
     printf("Error loading PKCS#11 library \n");
     return errno;
  }
  symPtr = (void (*)())dlsym(dllPtr, "C_GetFunctionList"); /* Get ock function list */
  if (!symPtr) {
     printf("Error getting function list \n");
     return errno;
  }
  symPtr(&FunctionPtr);
  rc = FunctionPtr->C_Initialize(NULL);    /* initialize opencryptoki/tokens) */
   if (rc != CKR_OK) {
    printf("Error initializing the opencryptoki library: 0x%X\n", rc);
    cleanup();
    }
    printf("Opencryptoki initialized.\n");
    return CKR_OK;
}

  CK_RV cleanup(void) {
      rc = FunctionPtr->C_Finalize(NULL);
      if (dllPtr)
          dlclose(dllPtr);
      return rc;
}
```

To compile your sample code you need to provide the path of the source/include files. Issue a command of the form:

```
gcc sample_dynamic.c -g -O0 -o sample_dynamic -I <include filepath>
```

The exact location of the include files depends on your Linux distribution.

## Shared linked library

When you use your sample code with a static linked library you can access the APIs directly.

At the compile time you need to specify the openCryptoki library:

```
gcc sample_shared.c -g -O0 -o sample_shared /usr/lib64/opencryptoki/libopencryptoki.so
-I /usr/<include filepath>
```

The exact location of the include files depend on your Linux distribution.

The following code samples that interact with the openCryptoki API are based on the shared linked openCryptoki library.

**Base procedures:**

View some openCryptoki code samples for base procedures, such as main program, initialization, slot and token, mechanism, and finalize information.

The following code sample provides an insight into how to deal with the openCryptoki API's. After describing some basic functions such as initialization, session and login handling, the sample shows how to retrieve data, such as get slot and token information and also detailed mechanism information. It also provides an introduction about how to create key objects and process symmetric encryption/decryption (DES). The last section shows RSA key generation with RSA encrypt and decrypt operations.

### Main program

```
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <dlfcn.h>
#include <pkcs11types.h>
#include <defs.h>

K_SLOT_ID   slotID;
CK_SLOT_ID_PTR         pSlotList = NULL;
CK_ULONG               slotCount, ulCount, rsaLen = 2048, msgLen = 8, cipherLen = 8, c;
CK_FLAGS   rw_sessionFlags = CKF_RW_SESSION | CKF_SERIAL_SESSION;
CK_SESSION_HANDLE  hSession;
CK_MECHANISM_TYPE_PTR  pMechList = NULL;
CK_BYTE keyValue[]  = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef};
CK_BYTE msg[]       = {'T', 'h', 'e', ' ', 'b', 'i', 'r', 'd'};
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;

/* <insert helper functions (provided below) here> */

int main(int argc, char *argv[]) {
  init();
  getSlotList(pSlotList, &slotCount);         // get the number of slots
  pSlotList = malloc(slotCount * sizeof(CK_SLOT_ID));  // allocate memory
  getSlotList(pSlotList, &slotCount);             // retrieve slot list
  slotID = *pSlotList;             // first slot provide ica-token
  getSlotInfo(slotID);
  getTokenInfo(slotID);
  getMechanismList(slotID, pMechList, &ulCount);    // retrieve number of mech's
  pMechList = malloc(ulCount * sizeof(CK_MECHANISM_TYPE)); // allocate memory
  getMechanismList(slotID, pMechList, &ulCount);         // retrieve mechanism list
  getMechanismInfo(slotID, CKM_DES3_ECB);               // get mechanism information
  openSession(slotID, rw_sessionFlags, &hSession);
  loginSession(CKU_USER, "01234567", 8, hSession);
  createKeyObject(hSession, keyValue);
  CK_BYTE_PTR pCipherText = malloc(DES_BLOCK_SIZE*sizeof(CK_BYTE));
  DESencrypt(hSession, (CK_BYTE_PTR)&msg, msgLen, pCipherText, &cipherLen);
  DESdecrypt(hSession, pCipherText, cipherLen, (CK_BYTE_PTR)&msg, &msgLen);
  generateRSAKeyPair(hSession, rsaLen, &hPublicKey, &hPrivateKey);
  CK_BYTE_PTR pEncryptText = malloc(rsaLen*sizeof(CK_BYTE));
  CK_BYTE_PTR pClearText = malloc(rsaLen*sizeof(CK_BYTE));
  RSAencrypt(hSession, hPublicKey, (CK_BYTE_PTR)&msg, msgLen, pEncryptText, &rsaLen);
  RSAdecrypt(hSession, hPrivateKey, pEncryptText, rsaLen, pClearText, &rsaLen);
  logoutSession(hSession);   closeSession(hSession);
  finalize();
  return 0;
 }
```

### C_Initialize:

```
CK_RV init(void){
  CK_RV rc;
  rc = C_Initialize(NULL);
  if (rc != CKR_OK) {
      printf("Error initializing the opencryptoki library: 0x%X\n", rc);
  }
  return CKR_OK;
}
```

### C_GetSlotList:

```
CK_RV getSlotList(CK_SLOT_ID_PTR pSlotList, CK_ULONG_PTR pSlotCount){
   CK_RV rc;
   rc = C_GetSlotList(TRUE, pSlotList, pSlotCount);
   if (rc != CKR_OK) {
      printf("Error getting number of slots: %x \n", rc);
      return rc;
   }
   return CKR_OK;
}
```

### C_GetSlotInfo:

```
CK_RV getSlotInfo(CK_SLOT_ID slotID){
      CK_RV         rc;
      CK_SLOT_INFO   slotInfo;

       rc = C_GetSlotInfo(slotID, &slotInfo);
       if (rc != CKR_OK) {
          printf("Error getting slot information: %x \n", rc);
          return rc;
       }
       printf("Slot %d Information:\n", slotID);
       printf("  Description: %.64s\n", slotInfo.slotDescription);
       printf("  Manufacturer: %.32s\n", slotInfo.manufacturerID);
       printf("  Flags: 0x%X\n", slotInfo.flags);
       if ((slotInfo.flags & CKF_TOKEN_PRESENT) == CKF_TOKEN_PRESENT) {
           printf("Token Present!\n");
       }
       if ((slotInfo.flags & CKF_REMOVABLE_DEVICE) ==
              CKF_REMOVABLE_DEVICE) {
          printf("Removable Device!\n");
       }
       if ((slotInfo.flags & CKF_HW_SLOT) == CKF_HW_SLOT){
          printf("Hardware support!\n");
       }
       else { printf("Software support!\n");}
       printf("  Hardware Version: %d.%d\n",
            slotInfo.hardwareVersion.major,
            slotInfo.hardwareVersion.minor);
       printf("  Firmware Version: %d.%d\n",
            slotInfo.firmwareVersion.major,
            slotInfo.firmwareVersion.minor);
    return CKR_OK;
 }
```

## C_GetTokenInfo:

```
CK_RV getTokenInfo(CK_SLOT_ID slotID){
  CK_RV          rc;
  CK_TOKEN_INFO tokInfo;
  rc = C_GetTokenInfo(slotID, &tokinfo);
  if (rc != CKR_OK) {
      printf("Error getting token info: 0x%X\n", rc); return rc;
  }
  printf("Token #%d Info:\n", slotID);
  printf("  Label: %.32s\n", (&tokinfo)->label);
  printf("  Manufacturer: %.32s\n", (&tokinfo)->manufacturerID);
  printf("  Model: %.16s\n", (&tokinfo)->model);
  printf("  Serial Number: %.16s\n", (&tokinfo)->serialNumber);
  printf("  Flags: 0x%X\n", (&tokinfo)->flags);
  if (((&tokinfo)->flags & CKF_RNG)== CKF_RNG)
     printf("                |_ token has random generator\n");
  if (((&tokinfo)->flags & CKF_WRITE_PROTECTED)== CKF_WRITE_PROTECTED)
     printf("                |_ write protected token\n");
  if (((&tokinfo)->flags & CKF_LOGIN_REQUIRED)== CKF_LOGIN_REQUIRED)
     printf("                |_ Login required\n");
  if (((&tokinfo)->flags & CKF_USER_PIN_INITIALIZED)== CKF_USER_PIN_INITIALIZED)
     printf("                |_ User Pin initialized\n");
  if (((&tokinfo)->flags & CKF_RESTORE_KEY_NOT_NEEDED)== CKF_RESTORE_KEY_NOT_NEEDED)
     printf("               |_ Restore Keys not needed\n");
  if (((&tokinfo)->flags & CKF_CLOCK_ON_TOKEN)== CKF_CLOCK_ON_TOKEN)
     printf("               |_ Token has hardware clock\n");
  if(((&tokinfo)->flags & CKF_PROTECTED_AUTHENTICATION_PATH)==CKF_PROTECTED_AUTHENTICATION_PATH)
     printf("              |_ Token has protected configuration path\n");
  if (((&tokinfo)->flags & CKF_DUAL_CRYPTO_OPERATIONS)== CKF_DUAL_CRYPTO_OPERATIONS)
     printf("              |_ Token supports dual crypto operations\n");
  if (((&tokinfo)->flags & CKF_TOKEN_INITIALIZED) == CKF_TOKEN_INITIALIZED)
     printf("              |_ Token initialized\n");
  if (((&tokinfo)->flags & CKF_SECONDARY_AUTHENTICATION) == CKF_SECONDARY_AUTHENTICATION)
     printf("              |_ Token supports secondary authentication\n");
  if (((&tokinfo)->flags & CKF_USER_PIN_COUNT_LOW) == CKF_USER_PIN_COUNT_LOW)
     printf("            |_ at least one wrong user PIN submitted since last successful authentication\n");
  if (((&tokinfo)->flags & CKF_USER_PIN_FINAL_TRY) == CKF_USER_PIN_FINAL_TRY)
     printf("            |_ one last try before user PIN become locked\n");
  if (((&tokinfo)->flags & CKF_USER_PIN_LOCKED) == CKF_USER_PIN_LOCKED)
     printf("            |_ user PIN locked!!!\n");
  if (((&tokinfo)->flags & CKF_USER_PIN_TO_BE_CHANGED) == CKF_USER_PIN_TO_BE_CHANGED)
     printf("            |_ still default user PIN configured, PIN change recommended.\n");
  if (((&tokinfo)->flags & CKF_SO_PIN_COUNT_LOW) == CKF_SO_PIN_COUNT_LOW)
     printf("           |_ at least one wrong SO PIN submitted since last successful authentication\n");
  if (((&tokinfo)->flags & CKF_SO_PIN_FINAL_TRY) == CKF_SO_PIN_FINAL_TRY)
     printf("           |_ one last try before SO PIN become locked\n");
  if (((&tokinfo)->flags & CKF_SO_PIN_LOCKED) == CKF_SO_PIN_LOCKED)
     printf("           |_ SO PIN locked!!!\n");
  if (((&tokinfo)->flags & CKF_SO_PIN_TO_BE_CHANGED) == CKF_SO_PIN_TO_BE_CHANGED)
     printf("           |_ still default SO PIN configured, PIN change recommended.\n");
     printf("  Sessions: %d/%d\n", (&tokinfo)->ulSessionCount, (&tokinfo)->ulMaxSessionCount);
     printf("  R/W Sessions: %d/%d\n", (&tokinfo)->ulRwSessionCount, (&tokinfo)->ulMaxRwSessionCount);
     printf("  PIN Length: %d-%d\n", (&tokinfo)->ulMinPinLen, (&tokinfo)->ulMaxPinLen);
     printf("  Public Memory: 0x%X/0x%X\n", (&tokinfo)->ulFreePublicMemory, (&tokinfo)->ulTotalPublicMemory);
     printf("  Private Memory: 0x%X/0x%X\n",(&tokinfo)->ulFreePrivateMemory, (&tokinfo)->ulTotalPrivateMemory);
     printf("  Hardware Version: %d.%d\n", (&tokinfo)->hardwareVersion.major, (&tokinfo)->hardwareVersion.minor);
     printf("  Firmware Version: %d.%d\n", (&tokinfo)->firmwareVersion.major, (&tokinfo)->firmwareVersion.minor);
     printf("  Time: %.16s\n", (&tokinfo)->utcTime);
  return CKR_OK;
}
```

## C_GetMechanismList:

```
CK_RV getMechanismList(CK_SLOT_ID slotID, CK_MECHANISM_TYPE_PTR
        pMechList, CK_ULONG_PTR pulCount) {
  CK_RV rc;
   rc = C_GetMechanismList(slotID, pMechList, pulCount);
   if (rc != CKR_OK) {
     printf("Error retrieve mechanism list: %x\n", rc);
     return rc;
    }
  return CKR_OK;
}
```

### C_GetMechanismInfo:

```
CK_RV getMechanismInfo(CK_SLOT_ID slotID, CK_MECHANISM_TYPE type){
   CK_RV rc;
   CK_MECHANISM_INFO mechInfo;

   rc = C_GetMechanismInfo(slotID, type, &mechinfo);
   if (rc != CKR_OK) {
      printf("Error in mechanism info: %x\n", rc);
      return rc;
   }
   printf("MinKeySize: %d\n", (&mechinfo)->ulMinKeySize);
   printf("MaxKeySize: %d\n", (&mechinfo)->ulMaxKeySize);
   printf("Flags: %d\n", (&mechinfo)->flags);
   return CKR_OK;
}
```

### C_Finalize:

```
CK_RV finalize(void) {
   CK_RV rc;
   rc = C_Finalize(NULL);
   if (rc != CKR_OK) {
      printf("Error during finalize: %x\n", rc);
      return rc;
   }
   return CKR_OK;
}
```

### Session and login:

openCryptoki session and login code samples.

### C_OpenSession:

```
CK_RV openSession(CK_SLOT_ID slotID, CK_FLAGS sFlags,
                                     CK_SESSION_HANDLE_PTR phSession) {
 CK_RV rc;
  rc = C_OpenSession(slotID, sFlags, NULL, NULL, phSession);
  if (rc != CKR_OK) {
     printf("Error opening session: %x\n", rc); return rc;
  }
  printf("Open session successful.\n");
  return CKR_OK;
}
```

### C_Login:

```
CK_RV loginSession(CK_USER_TYPE userType, CK_CHAR_PTR pPin, CK_ULONG ulPinLen,
            CK_SESSION_HANDLE hSession) {
  CK_RV rc;
  rc = C_Login(hSession, userType, pPin, ulPinLen);
  if (rc != CKR_OK) {
      printf("Error login session: %x\n", rc);  return rc;
  }
  printf("Login session successful.\n");
  return CKR_OK;
}
```

### C_Logout:

```
CK_RV logoutSession(CK_SESSION_HANDLE hSession) {
CK_RV rc;
   rc = C_Logout(hSession);
   if (rc != CKR_OK) {
     printf("Error logout session: %x\n", rc); return rc;
  }
  printf("Logout session successful.\n");
  return CKR_OK;
}
```

### C_CloseSession:

```
CK_RV closeSession(CK_SESSION_HANDLE hSession) {
CK_RV rc;
   rc = C_CloseSession(hSession);
   if (rc != CKR_OK) {
     printf("Error closing session: 0x%X\n", rc); return rc;
  }
  printf("Close session successful.\n");
  return CKR_OK;
}
```

### Object handling:

openCryptoki object handling code samples.

### C_CreateObject:

```
CK_RV createKeyObject(CK_SESSION_HANDLE hSession, CK_BYTE keyValue[]) {
CK_RV rc;
CK_OBJECT_HANDLE hKey;
CK_BBOOL true  = TRUE;
CK_BBOOL false = FALSE;

CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES;
CK_ATTRIBUTE keyTempl[] = {
   {CKA_CLASS, &keyClass, sizeof(keyClass)},
   {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
   {CKA_ENCRYPT, &true, sizeof(true)},
   {CKA_DECRYPT, &true, sizeof(true)},
   {CKA_SIGN, &true, sizeof(true)},
   {CKA_VERIFY, &true, sizeof(true)},
   {CKA_TOKEN, &true, sizeof(true)},     // token object
   {CKA_PRIVATE, &false, sizeof(false)},    // public object
   {CKA_VALUE, keyValue, sizeof(keyValue)},
   {CKA_LABEL, "Public_DES_Key", sizeof("Public_DES_Key")}
};
  rc = C_CreateObject(hSession, keyTempl, sizeof (keyTempl)/sizeof (CK_ATTRIBUTE), &hKey);
  if (rc != CKR_OK) {
     printf("Error creating key object: 0x%X\n", rc); return rc;
  }
}
```

### C_FindObjects:

```
CK_RV getKey(CK_CHAR_PTR label, int labelLen, CK_OBJECT_HANDLE_PTR hObject,
CK_SESSION_HANDLE hSession) {
CK_RV rc;
CK_ULONG ulMaxObjectCount = 1;
CK_ULONG ulObjectCount;
CK_ATTRIBUTE objectMask[] = { {CKA_LABEL, label, labelLen}  };

  rc = C_FindObjectsInit(hSession, objectMask, 1);
  if (rc != CKR_OK) {
     printf("Error FindObjectsInit: 0x%X\n", rc); return rc;
  }

  rc = C_FindObjects(hSession, hObject, ulMaxObjectCount, &ulObjectCount);
    if (rc != CKR_OK) {
       printf("Error FindObjects: 0x%X\n", rc); return rc;
     }

  rc = C_FindObjectsFinal(hSession);
    if (rc != CKR_OK) {
       printf("Error FindObjectsFinal: 0x%X\n", rc); return rc;
   }
}
```

### Cryptographic operations:

View some openCryptoki cryptographic operations code samples.

### C_Encrypt (DES):

```
K_RV DESencrypt(CK_SESSION_HANDLE hSession,
       CK_BYTE_PTR pClearData, CK_ULONG ulClearDataLen,
       CK_BYTE_PTR pEncryptedData, CK_ULONG_PTR pulEncryptedDataLen) {
CK_RV rc;
CK_MECHANISM myMechanism = {CKM_DES_ECB, NULL_PTR, 0};
CK_MECHANISM_PTR pMechanism = &myMechanism
CK_OBJECT_HANDLE hKey;

getKey("Public_DES_Key", sizeof("Public_DES_Key"), &hKey, hSession);
rc = C_EncryptInit(hSession, pMechanism, hKey);
  if (rc != CKR_OK) {
     printf("Error initializing encryption: 0x%X\n", rc);
     return rc;
   }

rc = C_Encrypt(hSession, pClearData, ulClearDataLen, pEncryptedData, pulEncryptedDataLen);
  if (rc != CKR_OK) {
      printf("Error during encryption: %x\n", rc);
      return rc;
  }
  CK_BYTE_PTR tmp = pEncryptedData;
  for (c=0; c<*pulEncryptedDataLen;c++, pEncryptedData++) {
    printf("%X", *pEncryptedData);
  }
  printf("\n"); pEncryptedData = tmp;
return CKR_OK;
}
```

**C_Decrypt (DES):**

```
CK_RV DESdecrypt(CK_SESSION_HANDLE hSession,
       CK_BYTE_PTR pEncryptedData, CK_ULONG ulEncryptedDataLen,
       CK_BYTE_PTR pClearData, CK_ULONG_PTR pulClearDataLen) {
CK_RV rc;
CK_MECHANISM myMechanism = {CKM_DES_ECB, NULL_PTR, 0};
CK_MECHANISM_PTR pMechanism = &myMechanism
CK_OBJECT_HANDLE hKey;

getKey("Public_DES_Key", sizeof("Public_DES_Key"), &hKey, hSession);

  rc = C_DecryptInit(hSession, pMechanism, hKey);
  if (rc != CKR_OK) {
     printf("Error initializing decryption: 0x%X\n", rc);
     return rc;
   }

  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen,
         pClearData, pulClearDataLen);
  if (rc != CKR_OK) {
      printf("Error during decryption: %x\n", rc);
      return rc;
  }
  CK_BYTE_PTR tmp = pClearData;
  for (c=0; c<*pulClearDataLen;c++,pClearData++) {
    printf("%c", *pClearData);
  }
  printf("\n"); pClearData = tmp;
return CKR_OK;
}
```

### C_GenerateKeyPair (RSA):

```
CK_RV generateRSAKeyPair(CK_SESSION_HANDLE hSession, CK_ULONG keySize,
       CK_OBJECT_HANDLE_PTR phPublicKey, CK_OBJECT_HANDLE_PTR phPrivateKey ) {
  CK_RV rc;
  CK_BBOOL true  = TRUE;
  CK_BBOOL false = FALSE;

  CK_OBJECT_CLASS keyClassPub = CKO_PUBLIC_KEY;
  CK_OBJECT_CLASS keyClassPriv = CKO_PRIVATE_KEY;
  CK_KEY_TYPE keyTypeRSA = CKK_RSA;
  CK_ULONG modulusBits = keySize;
  CK_BYTE_PTR pModulus = malloc(sizeof(CK_BYTE)*modulusBits/8);
  CK_BYTE publicExponent[] = {1, 0, 1};
  CK_MECHANISM rsaKeyGenMech = {CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0};

  CK_ATTRIBUTE publicKeyTemplate[] = {
     {CKA_CLASS, &keyClassPub, sizeof(keyClassPub)},
     {CKA_KEY_TYPE, &keyTypeRSA, sizeof(keyTypeRSA)},
     {CKA_TOKEN, &true, sizeof(true)},
    {CKA_PRIVATE, &true, sizeof(true)},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_WRAP, &true, sizeof(true)},
    {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
    {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)},
    {CKA_LABEL, "My_Private_Token_RSA1024_PubKey",
     sizeof("My_Private_Token_RSA1024_PubKey")},
    {CKA_MODIFIABLE, &true, sizeof(true)},
  };

  CK_ATTRIBUTE privateKeyTemplate[] = {
    {CKA_CLASS, &keyClassPriv, sizeof(keyClassPriv)},
    {CKA_KEY_TYPE, &keyTypeRSA, sizeof(keyTypeRSA)},
    {CKA_EXTRACTABLE, &true, sizeof(true)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_PRIVATE, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_UNWRAP, &true, sizeof(true)},
    {CKA_LABEL, "My_Private_Token_RSA1024_PrivKey",
    sizeof("My_Private_Token_RSA1024_PrivKey")},
    {CKA_MODIFIABLE, &true, sizeof(true)},
  };

  rc = C_GenerateKeyPair(hSession, &rsaKeyGenMech , &publicKeyTemplate,
 sizeof(publicKeyTemplate)/sizeof (CK_ATTRIBUTE), &privateKeyTemplate,
  sizeof(privateKeyTemplate)/sizeof (CK_ATTRIBUTE), phPublicKey, phPrivateKey);
  if (rc != CKR_OK) {
     printf("Error generating RSA keys: %x\n", rc);
     return rc;
   }
}
```

### C_Encrypt (RSA):

```
CK_RV RSAencrypt(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hKey,
  CK_BYTE_PTR pClearData, CK_ULONG ulClearDataLen,
  CK_BYTE_PTR pEncryptedData, CK_ULONG_PTR pulEncryptedDataLen) {
CK_RV rc;
CK_MECHANISM rsaMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};

  rc = C_EncryptInit(hSession, rsaMechanism, hKey);
  if (rc != CKR_OK) {
     printf("Error initializing RSA encryption: %x\n", rc);
     return rc;
   }
  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
  pEncryptedData, pulEncryptedDataLen);
  if (rc != CKR_OK) {
      printf("Error during RSA encryption: %x\n", rc);
      return rc;
  }
 CK_BYTE_PTR tmp = pEncryptedData;
  for (c=0; c<*pulEncryptedDataLen;c++,pEncryptedData++) {
    printf("%X", *pEncryptedData);
  }
  printf("\n"); pEncryptedData = tmp;
 return CKR_OK;
}
```

### C_Decrypt (RSA):

```
CK_RV RSAdecrypt(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hKey,
   CK_BYTE_PTR pEncryptedData, CK_ULONG ulEncryptedDataLen,
   CK_BYTE_PTR pClearData, CK_ULONG_PTR pulClearDataLen) {
CK_RV rc;
CK_MECHANISM rsaMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};

  rc = C_DecryptInit(hSession, rsaMechanism, hKey);
  if (rc != CKR_OK) {
     printf("Error initializing RSA decryption: %x\n", rc);
     return rc;
   }
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen,
  pClearData, pulClearDataLen);
  if (rc != CKR_OK) {
      printf("Error during RSA decryption: %x\n", rc);
      return rc;
  }
  CK_BYTE_PTR tmp = pClearData;
  for (c=0; c<*pulClearDataLen;c++,pClearData++) {
    printf("%c", *pClearData);
  }
  printf("\n"); pClearData = tmp;
  return CKR_OK;
}
```

For more information, refer to the current PKCS#11standard/specification: `http://www.cryptsoft.com/pkcs11doc/`

# Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

## Documentation accessibility

The Linux on System z publications are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF file and want to request a Web-based format for this publication, use the Readers' Comments form in the back of this publication, send an email to eservdoc@de.ibm.com, or write to:

IBM Deutschland Research & Development GmbH
Information Development
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

In the request, be sure to include the publication number and title.

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

## IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility at

`www.ibm.com/able`

# Notices

This information was developed for products and services offered in the U.S.A. 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
U.S.A.

**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.

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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

## Trademarks

# Glossary

**Advanced Encryption Standard (AES)**
: A data encryption technique that improved upon and officially replaced the Data Encryption Standard (DES). AES is sometimes referred to as Rijndael, which is the algorithm on which the standard is based.

**asymmetric cryptography**
: Synonym for public key cryptography..

**Central Processor Assist for Cryptographic Function (CPACF)**
: Hardware that provides support for symmetric ciphers and secure hash algorithms (SHA) on every central processor. Hence the potential encryption/decryption throughput scales with the number of central processors in the system.

**Chinese-Remainder Theorem (CRT)**
: A mathematical problem described by Sun Tsu Suan-Ching using the remainder from a division operation.

**Cipher Block Chaining (CBC)**
: A method of reducing repetitive patterns in cipher-text by performing an exclusive-OR operation on each 8-byte block of data with the previously encrypted 8-byte block before it is encrypted.

**Cipher block length**
: The length of a block that can be encrypted or decrypted by a symmetric cipher. Each symmetric cipher has a specific cipher block length.

**clear key**
: Any type of encryption key not protected by encryption under another key.

**CPACF instructions**
: Instruction set for the CPACF hardware.

**Crypto Express4S (CEX4S)**
: Successor to the Crypto Express3 feature. The PCIe adapter on a CEX4S feature can be configured in three ways: Either as cryptographic accelerator (CEX4A), or as CCA coprocessor (CEX4C) for secure key encrypted transactions, or in EP11 coprocessor mode (CEX4P) for exploiting Enterprise PKCS #11 functionality.

  A CEX4P only supports secure key mode.

**electronic code book mode (ECB mode)**
: A method of enciphering and deciphering data in address spaces or data spaces. Each 64-bit block of plain-text is separately enciphered and each block of the cipher-text is separately deciphered.

**libica**
: Library for IBM Cryptographic Architecture.

**master key (MK)**
: In computer security, the top-level key in a hierarchy of key-encrypting keys.

**Mode of operation**
: A schema describing how to apply a symmetric cipher to encrypt or decrypt a message that is longer than the cipher block length. The goal of most modes of operation is to keep the security level of the cipher by

 

avoiding the situation where blocks that occur more than once will always be translated to the same value. Some modes of operations allow handling messages of arbitrary lengths.

**modulus-exponent (Mod-Expo)**
A type of exponentiation performed using a modulus.

**public key cryptography**
In computer security, cryptography in which a public key is used for encryption and a private key is used for decryption. Synonymous with asymmetric cryptography.

**Rivest-Shamir-Adleman (RSA)**
An algorithm used in public key cryptography. These are the surnames of the three researchers responsible for creating this asymmetric or public/private key algorithm.

**Secure Hash Algorithm (SHA)**
An encryption method in which data is encrypted in a way that is mathematically impossible to reverse. Different data can possibly produce the same hash value, but there is no way to use the hash value to determine the original data.

**secure key**
A key that is encrypted under a master key. When using a secure key, it is passed to a cryptographic coprocessor where the coprocessor decrypts the key and performs the function. The secure key never appears in the clear outside of the cryptographic coprocessor.

**symmetric cryptogrphy**
An encryption method that uses the same key for encryption and decryption. Keys of symmetric ciphers are private keys.

**zcrypt device driver**
Kernel device driver to access Crypto Express adapters. Formerly, a monolithic module called **z90crypt**. Today, it consists of multiple modules that are implicitly loaded when loading the **ap** main module of the device driver.

# Index

## Numerics



## A



## C



## D



## E

## U



## W



## Z

# Readers' Comments — We'd Like to Hear from You

**Linux on System z**
**libica Programmer's Reference**
**Version 2.4**

**Publication No. SC34-2602-06**

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

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 state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send your comments via email to: eservdoc@de.ibm.com

If you would like a response from IBM, please fill in the following information:

| | |
|---|---|
| Name | Address |
| Company or Organization | |
| Phone No. | Email address |

**Readers' Comments — We'd Like to Hear from You**
SC34-2602-06

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Research & Development GmbH
Information Development
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape



Cut or Fold
Along Line

SC34-2602-06