Linux on System z

libica Programmer's Reference

Version 2

SC34-2602-00

Linux on System z

libica Programmer's Reference

Version 2

SC34-2602-00

Note

Before using this document, be sure to read the information in “Notices” on page 89.

This edition applies to version 2 of libica (libica Version 2) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this document

This document describes how to install and use Version 2 of the Library for IBM® Cryptographic Architecture (libica Version 2). libica Version 2 is a library of cryptographic functions used to write cryptographic applications on IBM System z®, both with and without cryptographic hardware.

Unless stated otherwise, the tools described in this book are available for the 64-bit architecture and 31-bit architectures with version 2.6 of the Linux kernel.

You can find the latest version of this document on the developerWorks Web site at:
`http://www.ibm.com/developerworks/linux/linux390/development_documentation.html`

How this document is organized

Chapter 1 has general information about libica Version 2.

Chapter 2 contains installation and set up instructions, and coexistence information for libica Version 2.

Chapter 3 describes the libica Version 2 APIs.

Chapter 4 lists the defines, typedefs, structs, and return codes for libica Version 2.

Chapter 5 is a set of programming examples that use the libica Version 2 APIs.

Who should read this document

The information in this document is intended for programmers who want to accelerate their cryptographic operations with IBM hardware.

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 book does not provide information that is specific to a particular Linux distribution. The tools it describes are distribution independent.

Conventions used in this book

This section informs you on the styles, highlighting, and assumptions used throughout the book.

Terminology

In this book, the term *booting* is used for running boot loader code that loads the Linux® operating system. *IPL* is used for issuing an IPL command or to load boot-loader code.

IBM systems mentioned in this book have both long names and short names. They correspond as follows:

Table 1. IBM systems

Long name	Short name
IBM eServer™ zSeries® 990	z990
IBM System z9®	z9
IBM System z10™	z10

Highlighting

This book uses the following highlighting styles:

- Paths and URLs are highlighted in `monospace`.
- Variables are highlighted in *italics*.
- Commands in text are highlighted in **bold**.
- Input and output as normally seen on a computer screen is shown

```
within a screen frame.
Prompts are shown as number signs:
#
```

Finding IBM books

The PDF version of this book contains URL links to much of the referenced literature.

For some of the referenced IBM books, links have been omitted to avoid pointing to a particular edition of a book. You can locate the latest versions of the referenced IBM books through the IBM Publications Center at:

`http://www.ibm.com/shop/publications/order`

Chapter 1. General information about libica

The libica library provides hardware support for cryptographic functions. The cryptographic adapters are used for asymmetric encryption and decryption. The CPACF instructions are used for symmetric encryption and decryption, pseudo random number generation, and Secure Hashing. For all of these functions, if the hardware is not available or has failed, libica will use OpenSSL's low-level cryptographic functions.

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

The libica library works only on IBM System z hardware.

The libica library is not intended as general customer API, and IBM reserves the right to change or modify this API at any time.

The **icastats** command, described in *Linux on System z: Device Driver, Features, and Commands*, is used to obtain statistics about cryptographic processes. The **icastats** command shows whether libica is using cryptographic hardware or software fallback for each specific libica function.

This document describes the libica Version 2 library, which was derived from the libica Version 1 library. These are the significant differences:

- libica Version 1 has support for hardware other than IBM System z hardware. libica Version 2 supports only IBM System z hardware.
- libica Version 1 was written by many people over several years, and as a result the programming interfaces, names, styles, and so forth were not uniform. libica Version 2 has addressed these concerns.
- libica Version 1 had functions and interfaces that were not used or redundant. libica Version 2 has removed these functions.

Deleted functions

These libica Version 1 functions have been deleted:

- **icaRsaKeyReGenerateModExpo**
- **icaRsaKeyReGenerateCrt**

libica examples

There is a list of sample programs in the libica source for each API, as well as instructions about how to use some of the new functions. There are also example program segments in the source for libica itself. From the libica Version 1 implementation in `icalinux.c`, the new functions located in `ica_api.c` are called. You can find the Open Source version of libica at:

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

There is a direct correspondence between libica Version 1 and libica Version 2 APIs.

Sample programs area also in Chapter 5, "Examples," on page 35.

Chapter 2. Installing and using libica Version 2

Installing libica Version 2

You can obtain the libica Version 2 library from the SourceForge Web site at:
`http://sourceforge.net/projects/opencryptoki`

Follow the installation instructions on this Web site to download the libica Version 2 package. This package has a file named INSTALL that contain installation instructions.

Using libica Version 2

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 programs.

libica Version 1 and Version 2 coexistence

Some of the libica Version 1 APIs are available in libica Version 2. Some of them, such as those that work with an environment other than Linux on IBM System z, have been removed and are not present in libica Version 2 If your application program has calls to libica Version 1 APIs, check to see if these APIs appear in libica Version 1. If they do, these API calls should still work. However, we suggest that you convert your application to use the equivalent libica Version 2 functions. See Chapter 3, “libica Application Programming Interfaces (APIs),” on page 5.

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 an unsigned long integer.

Chapter 3. libica Application Programming Interfaces (APIs)

Table 2 is a table of APIs for libica Version 2, and the equivalent libica Version 1 API name.

Table 2. libica APIs

Description	libica Version 2 API	libica Version 1 API
Open and close adapter functions		
Open an adapter.	“ica_open_adapter” on page 7	icaOpenAdapter
Close an adapter.	“ica_close_adapter” on page 8	icaCloseAdapter
Pseudo random number generation		
Generate a pseudo random number.	“ica_random_number_generate” on page 10	icaRandomNumberGenerate
Secure hash operations		
Perform secure hash operation using the SHA-1 algorithm.	“ica_sha1” on page 12	icaSha1
Perform secure hash operation using the SHA-224 algorithm.	“ica_sha224” on page 13	icaSha224
Perform secure hash operation using the SHA-256 algorithm.	“ica_sha256” on page 14	icaSha256
Perform secure hash operation using the SHA-384 algorithm.	“ica_sha384” on page 15	icaSha384
Perform secure hash operation using the SHA-512 algorithm.	“ica_sha512” on page 16	icaSha512
DES, TDES/3DES functions		
Encrypt a DES key.	“ica_des_encrypt” on page 18	icaDesEncrypt
Decrypt a DES key.	“ica_des_decrypt” on page 19	icaDesDecrypt
Encrypt a triple DES key.	“ica_3des_encrypt” on page 20	icaTDesEncrypt
Decrypt a triple DES key.	“ica_3des_decrypt” on page 21	icaTDesDecrypt
AES functions		
Encrypt an AES key.	“ica_aes_encrypt” on page 23	icaAesEncrypt
Decrypt an AES key.	“ica_aes_decrypt” on page 24	icaAesDecrypt
RSA key generation functions		
Generate an RSA public/private key pair of type *ica_rsa_key_mod_expo_t*.	“ica_rsa_key_generate_mod_expo” on page 26	icaRsaKeyGenerateModExpo
Generate an RSA public/private key pair of type *ica_rsa_key_crt_t*.	“ica_rsa_key_generate_crt” on page 27	icaRsaKeyGenerateCrt
RSA mod expo functions		
Perform a modulus/exponent operation with an RSA key of type *ica_rsa_key_mod_expo_t*.	“ica_rsa_mod_expo” on page 29	icaRsaModExpo
Perform a mod/expo operation with an RSA key of type *ica_rsa_key_crt_t*.	“ica_rsa_crt” on page 30	icaRsaCrt

Open and close adapter functions

Introduction

These are functions to open and close an adapter. 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.

Return codes

See “Return codes” on page 32.

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

See “Return codes” on page 32.

Pseudo random number generation function

Introduction

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 will be filled into 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 zcrypt/z90crypt crypto driver. If that fails, the initialization mechanism uses device `/dev/urandom`. Within the initialization, a byte counter *s390_byte_count* is set to 0. After 4096 bytes of the pseudo random number are generated, the random number generator will be seeded again.

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);
```

Parameters

unsigned int output_length
The length in bytes of the *output_data* buffer, and the desired length of the generated pseudo random number

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

Return codes

See “Return codes” on page 32.

Secure hash operations

Introduction

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 has already been performed. Also, it contains the part of the hash that has already been produced. For the user it is only interesting in cases where the message is not hashed at once, for then 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);
```

Parameters

unsigned int message_part
The message chaining state. This 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
The byte length of the input data to be SHA-1 hashed. This value must be greater than zero.

unsigned char *input_data
Pointer to the input data to be hashed.

sha_context_t *sha_context
Pointer to the SHA-1 context structure used to store the intermediate values when chaining is used. The application must not modify the contents of this structure when chaining is used.

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

Return codes

See “Return codes” on page 32.

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);
```

Parameters

unsigned int message_part
The message chaining state. This 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
The byte length of the input data to be SHA-224 hashed. This value must be greater than zero.

unsigned char *input_data
Pointer to the input data to be hashed.

sha256_context_t *sha256_context
Pointer to the SHA-256 context structure used to store the intermediate values when chaining is used. The application must not modify the contents of this structure when chaining is used.

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 will have a length of **SHA224_HASH_LENGTH**. Make sure that the buffer has is at least this size.

Return codes

See “Return codes” on page 32.

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);
```

Parameters

unsigned int message_part
The message chaining state. This 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
The byte length of the input data to be SHA-256 hashed. This value must be greater than zero.

unsigned char *input_data
Pointer to the input data to be hashed.

sha256_context_t *sha256_context
Pointer to the SHA-256 context structure used to store the intermediate values when chaining is used. The application must not modify the contents of this structure when chaining is used.

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

Return codes

See “Return codes” on page 32.

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);
```

Parameters

unsigned int message_part
The message chaining state. This 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
The byte length of the input data to be SHA-384 hashed. This value must be greater than zero.

unsigned char *input_data
Pointer to the input data to be hashed.

sha512_context_t *sha512_context
Pointer to the SHA-512 context structure used to store the intermediate values when chaining is used. The application must not modify the contents of this structure when chaining is used.

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 will have a length of **SHA384_HASH_LENGTH**. Make sure that the buffer is at least this size.

Return codes

See “Return codes” on page 32.

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);
```

Parameters

unsigned int message_part
The message chaining state. This 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
The byte length of the input data to be SHA-512 hashed. This value must be greater than zero.

unsigned char *input_data
Pointer to the input data to be hashed.

sha512_context_t *sha512_context
Pointer to the SHA-512 context structure used to store the intermediate values when chaining is used. The application must not modify the contents of this structure when chaining is used.

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

Return codes

See “Return codes” on page 32.

DES, TDES/3DES functions

Introduction

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

These functions perform encryption and decryption using a single DES or triple DES key. In CBC mode, the initialization vector will be overwritten by the function. Therefore, the initialization vector can be used again as the initialization vector in a subsequent function call to encrypt the next block.

ica_des_encrypt

Purpose

Encrypts data using a single-length DES key.

Format

```
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);
```

Parameters

unsigned int mode
The operational mode, which must be one of these values:

MODE_ECB Electronic Code Book mode

MODE_CBC Cipher Block Chaining mode

unsigned int data_length
The length in bytes of the input data. This value must be a multiple of the cipher block, which has a size of 8 bytes.

unsigned char *input_data
Pointer to the input data to be encrypted. This value must be a multiple of the cipher block, in order to use cryptographic hardware acceleration.

ica_des_vector_t *iv
Pointer to a valid 8 byte initialization vector, when using CBC mode.

ica_des_key_single_t *des_key
Pointer to a single-length DES key.

unsigned char *output_data
Pointer to the buffer for the resulting encrypted data. The size must be a multiple of the cipher block size, and at least as large as the buffer for *input_data*.

Return codes

See “Return codes” on page 32.

ica_des_decrypt

Purpose

Decrypts data using a single-length DES key.

Format

```
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);
```

Parameters

unsigned int mode
The operational mode, which must be one of these values:

MODE_ECB Electronic Code Book mode

MODE_CBC Cipher Block Chaining mode

unsigned int data_length
The length in bytes of the input data. This value must be a multiple of the cipher block, which has a size of 8 bytes.

unsigned char *input_data
Pointer to the input data to be decrypted. This value must be a multiple of the cipher block, in order to use cryptographic hardware acceleration.

ica_des_vector_t *iv
Pointer to a valid 8 byte initialization vector, when using CBC mode.

ica_des_key_single_t *des_key
Pointer to a single-length DES key.

unsigned char *output_data
Pointer to the buffer for the resulting decrypted data. The size must be a multiple of the cipher block size, and at least as large as the buffer for *input_data*.

Return codes

See “Return codes” on page 32.

ica_3des_encrypt

Purpose

Encrypts data using a triple-length DES key.

Format

```
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);
```

Parameters

unsigned int mode
The operational mode, which must be one of these values:

MODE_ECB Electronic Code Book mode

MODE_CBC Cipher Block Chaining mode

unsigned int data_length
The length in bytes of the input data. This value must be a multiple of the cipher block, which has a size of 8 bytes.

unsigned char *input_data
Pointer to the input data to be encrypted. This value must be a multiple of the cipher block, in order to use cryptographic hardware acceleration.

ica_des_vector_t *iv
Pointer to a valid 8 byte initialization vector, when using CBC mode.

ica_des_key_single_t *des_key
Pointer to a triple-length DES key.

unsigned char *output_data
Pointer to the buffer for the resulting encrypted data. The size must be a multiple of the cipher block size, and at least as large as the buffer for *input_data*.

Return codes

See “Return codes” on page 32.

ica_3des_decrypt

Purpose

Decrypts data using a triple-length DES key.

Format

```
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);
```

Parameters

unsigned int mode
The operational mode, which must be one of these values:

MODE_ECB Electronic Code Book mode

MODE_CBC Cipher Block Chaining mode

unsigned int data_length
The length in bytes of the input data. This value must be a multiple of the cipher block, which has a size of 8 bytes.

unsigned char *input_data
Pointer to the input data to be decrypted. This value must be a multiple of the cipher block, in order to use cryptographic hardware acceleration.

ica_des_vector_t *iv
Pointer to a valid 8 byte initialization vector, when using CBC mode.

ica_des_key_single_t *des_key
Pointer to a triple-length DES key.

unsigned char *output_data
Pointer to the buffer for the resulting decrypted data. The size must be a multiple of the cipher block size, and at least as large as the buffer for *input_data*.

Return codes

See “Return codes” on page 32.

AES functions

Introduction

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

These functions perform encryption and decryption using an AES key. In CBC mode, the initialization vector will be overwritten by the function. Therefore, the initialization vector can be used again as the initialization vector in a subsequent function call to encrypt the next block.

ica_aes_encrypt

Purpose

Encrypts data using AES keys with a *key_length* of 16, 24, or 32 bytes.

Format

```
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);
```

Parameters

unsigned int mode
The operational mode, which must be one of these values:

MODE_ECB Electronic Code Book mode

MODE_CBC Cipher Block Chaining mode

unsigned int data_length
The length in bytes of the input data. This value must be a multiple of the AES block length, which is 16 bytes.

unsigned char *input_data
Pointer to the input data to be encrypted. This value must be a multiple of the cipher block, in order to use cryptographic hardware acceleration.

ica_aes_vector_t *iv
Pointer to a valid 16 byte initialization vector, when using CBC mode.

unsigned int key_length
Length of the AES key being used.

unsigned char *aes_key
Pointer to a AES key.

unsigned char *output_data
Pointer to the buffer for the resulting encrypted data. The size must be a multiple of the cipher block size, and at least as large as the buffer for *input_data*.

Return codes

See “Return codes” on page 32.

ica_aes_decrypt

Purpose

Decrypts data using AES keys with a *key_length* of 16, 24, or 32 bytes.

Format

```
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);
```

Parameters

unsigned int mode
The operational mode, which must be one of these values:

MODE_ECB Electronic Code Book mode

MODE_CBC Cipher Block Chaining mode

unsigned int data_length
The length in bytes of the input data. This value must be a multiple of the AES block length, which is 16 bytes.

unsigned char *input_data
Pointer to the input data to be decrypted. This value must be a multiple of the cipher block, in order to use cryptographic hardware acceleration.

ica_aes_vector_t *iv
Pointer to a valid 16 byte initialization vector, when using CBC mode.

unsigned int key_length
Length of the AES key being used.

unsigned char *aes_key
Pointer to a AES key.

unsigned char *output_data
Pointer to the buffer for the resulting decrypted data. The size must be a multiple of the cipher block size, and at least as large as the buffer for *input_data*.

Return codes

See “Return codes” on page 32.

RSA key generation functions

Introduction

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 “Structs” on page 31.

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
The 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 will be 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

See “Return codes” on page 32.

ica_rsa_key_generate_crt

Purpose

Generates RSA keys in CRT format.

Comments

For specific information about some of these parameters, see the considerations in “Structs” on page 31.

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
The length of the modulus part of the key, in bits. 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 will be randomly generated. A poorly chosen *exponent* may 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

See “Return codes” on page 32.

RSA mod expo functions

Introduction

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 justify 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 leas the same size as the size of the modulus.

Return codes

See “Return codes” on page 32.

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 justify 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

See “Return codes” on page 32.

Chapter 4. libica defines, typedefs, structs and return codes

These are the defines, typedefs, structs, and return codes used when programming with the libica Version 2 APIs in Chapter 3, “libica Application Programming Interfaces (APIs),” on page 5.

Defines

These are defines that are new with libica Version 2 or have been changed from libica Version 1. Use these defines instead of the equivalent libica Version 1 defines. 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
```

Typedefs

These are typedefs that are new with libica Version 2. These new typedefs are introduced so that the libica Version 1 upper case names will still work. There is no difference.

```
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];
```

Structs

These are structs that are new with libica Version 2. Use these structs for RSA operations. These are simplified versions of the structs used with libica Version 1. There are pointers to the particular parts of the keys.

```
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;
```

Take note of these considerations:

- The buffers must be reserved by the user manually.
- Key parts must always be right justified in their fields.
- All buffers pointed to by parameters *modulus* and *exponent* in struct *ica_rsa_key_mod_expo_t* must be of length *key_length*.
- All buffers pointed to by parameters *p*, *q*, *dp*, *dq*, and *qInverse* in struct *ica_rsa_key_crt_t* must be of size *key_length* / 2 or larger.
- In a CRT key, the buffers of *p*, *dp* and *qInverse* must have an additional 8 bytes of zero padding in front. The *key_length* is not increased by this padding.
- 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 zeroes 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
  ```

These are libica Version 2 structs that have been changed from libica Version 1.

```
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 {
        ica_des_key_single_t key1;
        ica_des_key_single_t key2;
        ica_des_key_single_t key3;
} ica_des_key_triple_t;
```

Return codes

The libica Version 2 functions use these standard Linux return codes:

0 Success

EINVAL	Incorrect parameter
EIO	I/O error
ENODEV	No such device
ENOMEM	Not enough memory
errno	When libica Version 2 calls **open**, **close**, **begin_sigill_section**, or OpenSSL function **RSA_generate_key**, the error codes of these programs are returned.

Chapter 5. Examples

These are sample program segments used to illustrate the libica Version 2 APIs. In most of these programming samples, there is a function with 'old' in the name, which contains the old (libica Version 1) part of the test, and an almost identical version with 'new' in the name, which contains the new (libica Version 2) part of the test.

For example, in the program for Triple DES, the function named test_3des_old_api is an example for using the old (libica Version 1) triple DES API, and the function named test_3des_new_api is an example for the using the new (libica Version 2) triple DES API.

These sample programs are from the libica Version 2 RPM, and they have been enhanced to use the libica Version 2 APIs, in addition to the old (libica Version 1) APIs.

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

Pseudo random number generation example

This is an example of the old (libica Version 1) API. Examples for using the new (libica Version 2) API for random number generation are located in other examples, such as the AES 128 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.
 */

/* (C) COPYRIGHT International Business Machines Corp. 2001, 2009          */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include "ica_api.h"

unsigned char R[512];

extern int errno;

int main(int ac, char **av)
{
   int rc;
   ICA_ADAPTER_HANDLE adapter_handle;

   void dump_array(caddr_t ptr, int size);

   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;
}

void dump_array(char *ptr, int size)
{
   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");
}
```

SHA-1 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.
 */

/* (C) COPYRIGHT International Business Machines Corp. 2001, 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_SHA_HASH] =
{
  // Hash for test 0: "abc"
  {
0xA9,0x99,0x3E,0x36,0x47,0x06,0x81,0x6A,0xBA,0x3E,0x25,0x71,0x78,0x50,0xC2,0x6C,
0x9C,0xD0,0xD8,0x9D,
  },
  // Hash for test 1: "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
  {
0x84,0x98,0x3E,0x44,0x1C,0x3B,0xD2,0x6E,0xBA,0xAE,0x4A,0xA1,0xF9,0x51,0x29,0xE5,
0xE5,0x46,0x70,0xF1,
  },
  // Hash for test 2: 1,000,000 'a'
  {
0x34,0xAA,0x97,0x3C,0xD4,0xC4,0xDA,0xA4,0xF6,0x1E,0xEB,0x2B,0xDB,0xAD,0x27,0x31,
0x65,0x34,0x01,0x6F,
  },
};

void dump_array(char *, int);

int old_api_sha_test(void)
{
 printf("Test of old sha api\n");
  ICA_ADAPTER_HANDLE adapter_handle;
  SHA_CONTEXT ShaContext;
  int rc = 0, i = 0;
  unsigned char input_data[1000000];
  unsigned int  output_hash_length = LENGTH_SHA_HASH;
  unsigned char output_hash[LENGTH_SHA_HASH];

  rc = icaOpenAdapter(0, &adapter_handle);
  if (rc != 0) {
    printf("icaOpenAdapter failed and returned %d (0x%x).\n", rc, rc);
    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 = icaSha1(adapter_handle,
                 SHA_MSG_PART_ONLY,
                 FIPS_TEST_DATA_SIZE[i],
                 input_data,
```

```
                LENGTH_SHA_CONTEXT,
                &ShaContext,
                &output_hash_length,
                output_hash);

    if (rc != 0) {
      printf("icaSha1 failed with errno %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
      return 2;
    }

    if (output_hash_length != LENGTH_SHA_HASH) {
      printf("icaSha1 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_SHA_HASH) != 0) {
       printf("This does NOT match the known result.\n");
    } else {
       printf("Yep, it's what it should be.\n");
    }
  }

  // This test is the same as test 2, except that we use the SHA_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 = icaSha1(adapter_handle,
                shaMessagePart,
                (i < 1024) ? i : 1024,
                input_data,
                LENGTH_SHA_CONTEXT,
                &ShaContext,
                &output_hash_length,
                output_hash);

    if (rc != 0) {
      printf("icaSha1 failed with errno %d (0x%x) on iteration %d.\n", rc, rc, i);
      return 2;
    }

    i -= 1024;
  }

  if (output_hash_length != LENGTH_SHA_HASH) {
    printf("icaSha1 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_SHA_HASH) != 0) {
     printf("This does NOT match the known result.\n");
  } else {
     printf("Yep, it's what it should be.\n");
  }

  // This test is the same as test 2, except that we use the SHA_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 = icaSha1(adapter_handle,
                 shaMessagePart,
                 (i < 64) ? i : 64,
                 input_data,
                 LENGTH_SHA_CONTEXT,
                 &ShaContext,
                 &output_hash_length,
                 output_hash);

    if (rc != 0) {
      printf("icaSha1 failed with errno %d (0x%x) on iteration %d.\n", rc, rc, i);
      return 2;
    }

    i -= 64;
  }

  if (output_hash_length != LENGTH_SHA_HASH) {
    printf("icaSha1 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_SHA_HASH) != 0) {
     printf("This does NOT match the known result.\n");
  } else {
     printf("Yep, it's what it should be.\n");
  }

  printf("\nAll SHA1 tests completed successfully\n");

  icaCloseAdapter(adapter_handle);


}

int new_api_sha_test(void)
{
 printf("Test of new sha api\n");
 sha_context_t sha_context;
 int rc = 0, i = 0;
 unsigned char input_data[1000000];
```

```
unsigned int  output_hash_length = LENGTH_SHA_HASH;
unsigned char output_hash[LENGTH_SHA_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_sha1(SHA_MSG_PART_ONLY, FIPS_TEST_DATA_SIZE[i], input_data,
       &sha_context, output_hash);

if (rc != 0) {
 printf("icaSha1 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_SHA_HASH) != 0)
 printf("This does NOT match the known result.\n");
else
 printf("Yep, it's what it should be.\n");
}

// This test is the same as test 2, except that we use the SHA_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_sha1(sha_message_part, (i < 1024) ? i : 1024,
        input_data, &sha_context, output_hash);

 if (rc != 0) {
  printf("ica_sha1 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_SHA_HASH) != 0)
 printf("This does NOT match the known result.\n");
else
 printf("Yep, it's what it should be.\n");

// This test is the same as test 2, except that we use the SHA_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_sha1(sha_message_part, (i < 64) ? i : 64, input_data,
         &sha_context, output_hash);

  if (rc != 0) {
   printf("ica_sha1 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_SHA_HASH) != 0)
  printf("This does NOT match the known result.\n");
 else
 printf("Yep, it's what it should be.\n");

 printf("\nAll SHA1 tests completed successfully\n");

 return 0;
}

int main(int argc, char **argv)
{
 int rc = 0;

 rc = old_api_sha_test();
 if (rc) {
  printf("old_api_sha_test failed with rc = %i\n", rc);
  return rc;
 }
 rc = new_api_sha_test();
 if (rc) {
  printf("new_api_sha_test failed with rc = %i\n", rc);
  return rc;
 }

 return 0;
}

void
dump_array(char *ptr, int size)
{
  char *ptr_end;
  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);
}
```

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.
 */

/* (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(char *, int);

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("Yep, 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("Yep, 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("Yep, 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("Yep, 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("Yep, 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("Yep, 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;
}

void
dump_array(char *ptr, int size)
{
  char *ptr_end;
  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);
}
```

DES 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.
 */

/* (C) COPYRIGHT International Business Machines Corp. 2001, 2009          */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

const int cipher_buf_length = 8;
```

```
unsigned char NIST_KEY1[] =
                  { 0x7c, 0xa1, 0x10, 0x45, 0x4a, 0x1a, 0x6e, 0x57 };

unsigned char NIST_TEST_DATA[] =
                  { 0x01, 0xa1, 0xd6, 0xd0, 0x39, 0x77, 0x67, 0x42 };

unsigned char NIST_TEST_RESULT[] =
                  { 0x69, 0x0f, 0x5b, 0x0d, 0x9a, 0x26, 0x93, 0x9b };

int silent = 1;

void dump_array(char *ptr, int size)
{
 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 test_des_old_api(int mode)
{
 ica_adapter_handle_t adapter_handle;
 ICA_DES_VECTOR iv;
 ICA_KEY_DES_SINGLE key;
 int rc = 0;
 unsigned char dec_text[sizeof NIST_TEST_DATA],
               enc_text[sizeof NIST_TEST_DATA];
 unsigned int i;

 bzero(dec_text, sizeof dec_text);
 bzero(enc_text, sizeof enc_text);
 bzero(iv, sizeof iv);
 bcopy(NIST_KEY1, key, sizeof NIST_KEY1);

 i = sizeof enc_text;
 rc = icaDesEncrypt(adapter_handle, mode, sizeof NIST_TEST_DATA,
                    NIST_TEST_DATA, &iv, &key, &i, enc_text);
 if (rc) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof NIST_TEST_DATA);
  printf("icaDesEncrypt failed with errno %d (0x%x).\n", rc, rc);
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof enc_text);
  return rc;
 }
 if (i != sizeof enc_text) {
  printf("icaDesEncrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
  return -1;
 }
 if (memcmp(enc_text, NIST_TEST_RESULT, sizeof NIST_TEST_RESULT) != 0) {
  printf("This does NOT match the known result.\n");
  return -1;
 } else {
  printf("Yep, it's what it should be.\n");
```

```
 }

 i = sizeof dec_text;
 bzero(iv, sizeof iv);
 rc = icaDesDecrypt(adapter_handle, mode, sizeof enc_text,
                    enc_text, &iv, &key, &i, dec_text);
 if (rc) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof NIST_TEST_DATA);
  printf("icaDesEncrypt failed with errno %d (0x%x).\n", rc, rc);
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof enc_text);
  printf("\nDecrypted data:\n");
  dump_array((char *) dec_text, sizeof dec_text);
  printf("icaDesDecrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }
 if (i != sizeof dec_text) {
  printf("icaDesDecrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
 }

 if (memcmp(dec_text, NIST_TEST_DATA, sizeof NIST_TEST_DATA) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof NIST_TEST_DATA);
  printf("icaDesEncrypt failed with errno %d (0x%x).\n", rc, rc);
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof enc_text);
  printf("\nDecrypted data:\n");
  dump_array((char *) dec_text, sizeof dec_text);
  printf("This does NOT match the original data.\n");
  return -1;
 } else {
  printf("Successful!\n");
 }

 return 0;
}

int test_des_new_api(int mode)
{
 ica_des_vector_t iv;
 ica_des_key_single_t key;
 int rc = 0;
 unsigned char dec_text[sizeof NIST_TEST_DATA],
               enc_text[sizeof NIST_TEST_DATA];

 bzero(dec_text, sizeof dec_text);
 bzero(enc_text, sizeof enc_text);
 bzero(iv, sizeof iv);
 bcopy(NIST_KEY1, key, sizeof NIST_KEY1);

 rc = ica_des_encrypt(mode, sizeof NIST_TEST_DATA, NIST_TEST_DATA, &iv,
        &key, enc_text);
 if (rc) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof NIST_TEST_DATA);
  printf("ica_des_encrypt failed with errno %d (0x%x).\n", rc, rc);
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof enc_text);
  return rc;
 }

 if (memcmp(enc_text, NIST_TEST_RESULT, sizeof NIST_TEST_RESULT) != 0) {
  printf("This does NOT match the known result.\n");
  return -1;
 } else {
  printf("Yep, it's what it should be.\n");
```

```
 }

 bzero(iv, sizeof iv);
 rc = ica_des_decrypt(mode, sizeof enc_text, enc_text, &iv, &key,
        dec_text);
 if (rc) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof NIST_TEST_DATA);
  printf("ica_des_encrypt failed with errno %d (0x%x).\n", rc, rc);
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof enc_text);
  printf("\nDecrypted data:\n");
  dump_array((char *) dec_text, sizeof dec_text);
  printf("ica_des_decrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }

 if (memcmp(dec_text, NIST_TEST_DATA, sizeof NIST_TEST_DATA) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof NIST_TEST_DATA);
  printf("ica_des_encrypt failed with errno %d (0x%x).\n", rc, rc);
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof enc_text);
  printf("\nDecrypted data:\n");
  dump_array((char *) dec_text, sizeof dec_text);
  printf("This does NOT match the original data.\n");
  return -1;
 } else {
  printf("Successful!\n");
 }

 return 0;
}

int main(int argc, char **argv)
{
 // Default mode is 0. ECB and CBC tests will be performed.
 unsigned int mode = 0;
 if (argc > 1) {
  if (strstr(argv[1], "ecb"))
   mode = MODE_ECB;
  if (strstr(argv[1], "cbc"))
   mode = MODE_CBC;
  printf("mode = %i \n", mode);
  }
 if (mode != 0 && mode != MODE_ECB && mode != MODE_CBC) {
  printf("Usage: %s [ ecb | cbc ]\n", argv[0]);
  return -1;
 }
 int rc = 0;
 int error_count = 0;
 if (!mode) {
 /* This is the standard loop that will perform all testcases */
  mode = 2;
  while (mode) {
   rc = test_des_old_api(mode);
   if (rc) {
    error_count++;
    printf ("test_des_old_api mode = %i failed \n", mode);
   }
   else
    printf ("test_des_old_api mode = %i finished successfully \n", mode);

   rc = test_des_new_api(mode);
   if (rc) {
    error_count++;
    printf ("test_des_new_api mode = %i failed \n", mode);
```

```
   }
   else
    printf ("test_des_new_api mode = %i finished successfully \n", mode);

   mode--;
  }
  if (error_count)
   printf("%i testcases failed\n", error_count);
  else
   printf("All testcases finished successfully\n");
 } else {
 /* Perform only the old test either ein ECB or CBC mode */
  silent = 0;
  rc = test_des_old_api(mode);
  if (rc)
   printf("test_des_old_api mode = %i failed \n", mode);
  else
   printf("test_des_old_api mode = %i finished successfully \n", mode);

  rc = test_des_new_api(mode);
  if (rc)
   printf ("test_des_new_api mode = %i failed \n", mode);
  else
   printf ("test_des_new_api mode = %i finished successfully \n", mode);
 }
 return rc;
}
```

Triple DES 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.
 */

/* (C) COPYRIGHT International Business Machines Corp. 2001, 2009          */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include "ica_api.h"


unsigned char NIST_KEY1[] =
                  { 0x7c, 0xa1, 0x10, 0x45, 0x4a, 0x1a, 0x6e, 0x57 };

unsigned char NIST_KEY2[] =
                  { 0x7c, 0xa1, 0x10, 0x45, 0x4a, 0x1a, 0x6e, 0x57 };

unsigned char NIST_KEY3[] =
                  { 0x7c, 0xa1, 0x10, 0x45, 0x4a, 0x1a, 0x6e, 0x57 };

unsigned char NIST_TEST_DATA[] =
                  { 0x01, 0xa1, 0xd6, 0xd0, 0x39, 0x77, 0x67, 0x42 };

unsigned char NIST_TEST_RESULT[] =
                  { 0x69, 0x0f, 0x5b, 0x0d, 0x9a, 0x26, 0x93, 0x9b };

int silent = 1;

void dump_array(caddr_t, int);

int test_3des_old_api(int mode)
{
 ica_adapter_handle_t adapter_handle;
```

```
 ica_des_vector_t iv;
 ica_des_key_triple_t key;
 int rc = 0;
 unsigned char dec_text[sizeof(NIST_TEST_DATA)],
        enc_text[sizeof(NIST_TEST_DATA)];
 unsigned int i;

 bzero(dec_text, sizeof(dec_text));
 bzero(enc_text, sizeof(enc_text));
 bzero(iv, sizeof(iv));
 bcopy(NIST_KEY1, key.key1, sizeof(NIST_KEY1));
 bcopy(NIST_KEY2, key.key2, sizeof(NIST_KEY2));
 bcopy(NIST_KEY3, key.key3, sizeof(NIST_KEY3));

 printf("\nOriginal data:\n");
 dump_array(NIST_TEST_DATA, sizeof(NIST_TEST_DATA));

 i = sizeof(enc_text);
 rc = icaTDesEncrypt(adapter_handle, mode, sizeof(NIST_TEST_DATA),
       NIST_TEST_DATA, &iv, &key, &i, enc_text);
 if (rc != 0) {
  printf("icaTDesEncrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }
 if (i != sizeof(enc_text)) {
  printf("icaTDesEncrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
  return -1;
 }

 printf("\nEncrypted data:\n");
 dump_array(enc_text, sizeof(enc_text));
 if (memcmp(enc_text, NIST_TEST_RESULT, sizeof NIST_TEST_RESULT) != 0) {
  printf("This does NOT match the known result.\n");
  return -1;
 } else {
  printf("Yep, it's what it should be.\n");
 }

 i = sizeof dec_text;
 bzero(iv, sizeof(iv));
 rc = icaTDesDecrypt(adapter_handle, mode, sizeof(enc_text), enc_text,
       &iv, &key, &i, dec_text);
 if (rc != 0) {
  printf("icaTDesDecrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }
 if (i != sizeof(dec_text)) {
  printf("icaTDesDecrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
  return rc;
 }

 printf("\nDecrypted data:\n");
 dump_array(dec_text, sizeof(dec_text));
 if (memcmp(dec_text, NIST_TEST_DATA, sizeof(NIST_TEST_DATA)) != 0) {
  printf("This does NOT match the original data.\n");
  return -1;
 } else {
  printf("Successful!\n");
 }

 return 0;
}

int test_3des_new_api(int mode)
{
 ica_des_vector_t iv;
 ica_des_key_triple_t key;
```

```
 int rc = 0;
 unsigned char dec_text[sizeof(NIST_TEST_DATA)],
        enc_text[sizeof(NIST_TEST_DATA)];

 bzero(dec_text, sizeof(dec_text));
 bzero(enc_text, sizeof(enc_text));
 bzero(iv, sizeof(iv));
 bcopy(NIST_KEY1, key.key1, sizeof(NIST_KEY1));
 bcopy(NIST_KEY2, key.key2, sizeof(NIST_KEY2));
 bcopy(NIST_KEY3, key.key3, sizeof(NIST_KEY3));

 printf("\nOriginal data:\n");
 dump_array(NIST_TEST_DATA, sizeof(NIST_TEST_DATA));

 rc = ica_3des_encrypt(mode, sizeof(NIST_TEST_DATA), NIST_TEST_DATA,
         &iv, &key, enc_text);
 if (rc != 0) {
  printf("ica_3des_encrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }

 printf("\nEncrypted data:\n");
 dump_array(enc_text, sizeof(enc_text));
 if (memcmp(enc_text, NIST_TEST_RESULT, sizeof NIST_TEST_RESULT) != 0) {
  printf("This does NOT match the known result.\n");
  return -1;
 } else {
  printf("Yep, it's what it should be.\n");
 }

 bzero(iv, sizeof(iv));
 rc = ica_3des_decrypt(mode, sizeof(enc_text), enc_text,
       &iv, &key, dec_text);
 if (rc != 0) {
  printf("ica_3des_decrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }

 printf("\nDecrypted data:\n");
 dump_array(dec_text, sizeof(dec_text));
 if (memcmp(dec_text, NIST_TEST_DATA, sizeof(NIST_TEST_DATA)) != 0) {
  printf("This does NOT match the original data.\n");
  return -1;
 } else {
  printf("Successful!\n");
 }

 return 0;
}

int main(int argc, char **argv)
{
 // Default mode is 0. ECB and CBC tests will be performed.
 unsigned int mode = 0;
 if (argc > 1) {
  if (strstr(argv[1], "ecb"))
   mode = MODE_ECB;
  if (strstr(argv[1], "cbc"))
   mode = MODE_CBC;
  printf("mode = %i \n", mode);
 }
 if (mode != 0 && mode != MODE_ECB && mode != MODE_CBC) {
  printf("Usage: %s [ ecb | cbc ]\n", argv[0]);
  return -1;
 }

 int rc = 0;
```

```
 int error_count = 0;
 if (!mode) {
 /* This is the standard loop that will perform all testcases */
  mode = 2;
  while (mode) {
   rc = test_3des_old_api(mode);
   if (rc) {
    error_count++;
    printf ("test_des_old_api mode = %i failed \n", mode);
   }
   else
    printf ("test_des_old_api mode = %i finished successfully \n", mode);

   rc = test_3des_new_api(mode);
   if (rc) {
    error_count++;
    printf ("test_des_new_api mode = %i failed \n", mode);
   }
   else
    printf ("test_des_new_api mode = %i finished successfully \n", mode);

   mode--;
  }
  if (error_count)
   printf("%i testcases failed\n", error_count);
  else
   printf("All testcases finished successfully\n");
 } else {
 /* Perform only the old test either ein ECB or CBC mode */
  silent = 0;
  rc = test_3des_old_api(mode);
  if (rc)
   printf("test_des_old_api mode = %i failed \n", mode);
  else
   printf("test_des_old_api mode = %i finished successfully \n", mode);

  rc = test_3des_new_api(mode);
  if (rc)
   printf ("test_des_new_api mode = %i failed \n", mode);
  else
   printf ("test_des_new_api mode = %i finished successfully \n", mode);
 }

 return rc;
}

void dump_array(char *ptr, int size)
{
   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");
}
```

AES 128 example

This is an example using the AES encrypt and decrypt APIs with a key length of 128.

```
/* 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. 2005, 2009          */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include "ica_api.h"
#include <stdlib.h>
#include <openssl/aes.h>

int silent = 0;

unsigned char NIST_KEY1[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
};

unsigned char NIST_TEST_DATA[] = {
 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff,
};

unsigned char NIST_TEST_RESULT[] = {
 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30,
 0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a,
};

void dump_array(char *ptr, int size)
{
 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 test_aes128_old_api(int mode)
{
 ICA_ADAPTER_HANDLE adapter_handle;
 ICA_AES_VECTOR iv;
 ICA_KEY_AES_LEN128 key;
 int rc = 0;
```

```
unsigned char dec_text[sizeof(NIST_TEST_DATA)],
              enc_text[sizeof(NIST_TEST_DATA)];
unsigned int i;

bzero(dec_text, sizeof(dec_text));
bzero(enc_text, sizeof(enc_text));
bzero(iv, sizeof(iv));
bcopy(NIST_KEY1, key, sizeof(NIST_KEY1));
i = sizeof(enc_text);
rc = icaAesEncrypt(adapter_handle, mode, sizeof(NIST_TEST_DATA),
  NIST_TEST_DATA, &iv, AES_KEY_LEN128,
  (ICA_KEY_AES_SINGLE *)&key, &i, enc_text);
if (rc) {
 printf("key \n");
 dump_array((char *) &key, sizeof(NIST_KEY1));
 printf("\nOriginal data:\n");
 dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
 printf("test iv\n");
 dump_array((char *) &iv, sizeof(ica_aes_vector_t));
 printf("key\n");
 dump_array((char *) &key, AES_KEY_LEN128);
 printf("icaAesEncrypt failed with errno %d (0x%x).\n", rc, rc);
 return 1;
}
if (i != sizeof(enc_text)) {
 printf("icaAesEncrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
 return -1;
}

if (memcmp(enc_text, NIST_TEST_RESULT, sizeof(NIST_TEST_RESULT)) != 0) {
 printf("key \n");
 dump_array((char *) &key, sizeof(NIST_KEY1));
 printf("\nOriginal data:\n");
 dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
 printf("test iv\n");
 dump_array((char *) &iv, sizeof(ica_aes_vector_t));
 printf("key\n");
 dump_array((char *) &key, AES_KEY_LEN128);
 printf("\nEncrypted data:\n");
 dump_array((char *) enc_text, sizeof(enc_text));
 printf("This does NOT match the known result.\n");
 return 1;
} else {
 printf("Yep, it's what it should be.\n");
}

bzero(iv, sizeof(iv));
i = sizeof(dec_text);
rc = icaAesDecrypt(adapter_handle, mode, sizeof(enc_text),
     enc_text, &iv, AES_KEY_LEN128,
     (ICA_KEY_AES_SINGLE *)&key, &i, dec_text);
if (rc) {
 printf("key \n");
 dump_array((char *) &key, sizeof(NIST_KEY1));
 printf("\nOriginal data:\n");
 dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
 printf("test iv\n");
 dump_array((char *) &iv, sizeof(ica_aes_vector_t));
 printf("key\n");
 dump_array((char *) &key, AES_KEY_LEN128);
 printf("\nEncrypted data:\n");
 dump_array((char *) enc_text, sizeof(enc_text));
 printf("\nDecrypted data:\n");
 dump_array((char *) dec_text, sizeof(dec_text));
 printf("icaAesDecrypt failed with errno %d (0x%x).\n", rc, rc);
 return 1;
}
```

```
 if (i != sizeof(dec_text)) {
  printf("icaAesDecrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
  return 1;
 }

 if (memcmp(dec_text, NIST_TEST_DATA, sizeof(NIST_TEST_DATA)) != 0) {
  printf("This does NOT match the original data.\n");
  return 1;
 } else {
  printf("Successful!\n");
  if (!silent) {
   printf("key \n");
   dump_array((char *) &key, sizeof(NIST_KEY1));
   printf("\nOriginal data:\n");
   dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
   printf("test iv\n");
   dump_array((char *) &iv, sizeof(ica_aes_vector_t));
   printf("key\n");
   dump_array((char *) &key, AES_KEY_LEN128);
   printf("\nEncrypted data:\n");
   dump_array((char *) enc_text, sizeof(enc_text));
   printf("\nDecrypted data:\n");
   dump_array((char *) dec_text, sizeof(dec_text));
  }
 }

// Test 2

 rc = 0;
 bzero(dec_text, sizeof(dec_text));
 bzero(enc_text, sizeof(enc_text));
 bzero(iv, sizeof(iv));
 bzero(key, sizeof(key));

 unsigned int length = 64;
 unsigned char *decrypt = malloc(length);
 unsigned char *encrypt = malloc(length);
 unsigned char *original = malloc(length);

 rc = ica_random_number_generate(length, original);
 if (rc) {
  printf("ica_random_number_generate returned rc = %i\n", rc);
  return rc;
 }

 rc = ica_random_number_generate(AES_KEY_LEN128, (unsigned char *) key);
 if (rc) {
  printf("ica_random_number_generate returned rc = %i\n", rc);
  return rc;
 }

 i = length;
 rc = icaAesEncrypt(adapter_handle, mode, length,
      original, &iv, AES_KEY_LEN128,
      (unsigned char *)key, &i, (unsigned char *)encrypt);
 if (rc) {
  printf("\nOriginal data:\n");
  dump_array((char *) original, length);
  printf("KEY: \n");
  dump_array((char *) key, AES_KEY_LEN128);
  printf("icaAesEncrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }

 if (memcmp(encrypt, original, length) == 0) {
  printf("Encrypt and original are the same.\n");
  return 1;
```

```
 }

 bzero(iv, sizeof(iv));
 i = length;
 rc = icaAesDecrypt(adapter_handle, mode, length,
      encrypt, &iv, AES_KEY_LEN128,
      (unsigned char *)key, &i, (unsigned char *) decrypt);
 if (rc) {
  printf("\nOriginal data:\n");
  dump_array((char *) original, length);
  printf("KEY: \n");
  dump_array((char *) key, AES_KEY_LEN128);
  printf("\nEncrypted data:\n");
  dump_array((char *) encrypt, length);
  printf("icaAesDecrypt failed with errno %d (0x%x).\n", rc, rc);
  goto free;
 }

 if (memcmp(decrypt, original, length) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) original, length);
  printf("KEY: \n");
  dump_array((char *) key, AES_KEY_LEN128);
  printf("\nEncrypted data:\n");
  dump_array((char *) encrypt, length);
  printf("\nDecrypted data:\n");
  dump_array((char *) decrypt, length);
  printf("This does NOT match the original data.\n");
  rc = -1;
  goto free;
 }

 if(memcmp(decrypt, encrypt, length) == 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) original, length);
  printf("KEY: \n");
  dump_array((char *) key, AES_KEY_LEN128);
  printf("\nEncrypted data:\n");
  dump_array((char *) encrypt, length);
  printf("\nDecrypted data:\n");
  dump_array((char *) decrypt, length);
  printf("decrypt and encrypt are the same\n");
  rc = -1;
  goto free;

    } else {
  printf("Successful!\n");
 }
 free:
  free(original);
  free(encrypt);
  free(decrypt);

   return rc;
}

int test_aes128_new_api(int mode)
{
 ica_aes_vector_t iv;
 unsigned char key[AES_KEY_LEN128];
 int rc = 0;
 unsigned char dec_text[sizeof(NIST_TEST_DATA)],
               enc_text[sizeof(NIST_TEST_DATA)];

 bzero(dec_text, sizeof(dec_text));
 bzero(enc_text, sizeof(enc_text));
 bzero(iv, sizeof(iv));
```

```
bcopy(NIST_KEY1, key, sizeof(NIST_KEY1));

rc = ica_aes_encrypt(mode, sizeof(NIST_TEST_DATA), NIST_TEST_DATA, &iv,
       AES_KEY_LEN128, key, enc_text);
if (rc) {
 printf("key \n");
 dump_array((char *) key, sizeof(NIST_KEY1));
 printf("\nOriginal data:\n");
 dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
 printf("test iv\n");
 dump_array((char *) &iv, sizeof(ica_aes_vector_t));
 printf("key\n");
 dump_array((char *) key, AES_KEY_LEN128);
 printf("ica_aes_encrypt failed with errno %d (0x%x).\n", rc, rc);
 return 1;
}

if (memcmp(enc_text, NIST_TEST_RESULT, sizeof(NIST_TEST_RESULT)) != 0) {
 printf("key \n");
 dump_array((char *) key, sizeof(NIST_KEY1));
 printf("\nOriginal data:\n");
 dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
 printf("test iv\n");
 dump_array((char *) &iv, sizeof(ica_aes_vector_t));
 printf("key\n");
 dump_array((char *) key, AES_KEY_LEN128);
 printf("\nEncrypted data:\n");
 dump_array((char *) enc_text, sizeof(enc_text));
 printf("This does NOT match the known result.\n");
 return 1;
} else {
 printf("Yep, it's what it should be.\n");
}

bzero(iv, sizeof(iv));
rc = ica_aes_decrypt(mode, sizeof(enc_text), enc_text, &iv,
       AES_KEY_LEN128, key, dec_text);
if (rc) {
 printf("key \n");
 dump_array((char *) key, sizeof(NIST_KEY1));
 printf("\nOriginal data:\n");
 dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
 printf("test iv\n");
 dump_array((char *) &iv, sizeof(ica_aes_vector_t));
 printf("key\n");
 dump_array((char *) key, AES_KEY_LEN128);
 printf("\nEncrypted data:\n");
 dump_array((char *) enc_text, sizeof(enc_text));
 printf("\nDecrypted data:\n");
 dump_array((char *) dec_text, sizeof(dec_text));
 printf("ica_aes_decrypt failed with errno %d (0x%x).\n", rc, rc);
 return 1;
}

if (memcmp(dec_text, NIST_TEST_DATA, sizeof(NIST_TEST_DATA)) != 0) {
 printf("This does NOT match the original data.\n");
 return 1;
} else {
 printf("Successful!\n");
 if (!silent) {
  printf("key \n");
  dump_array((char *) key, sizeof(NIST_KEY1));
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("test iv\n");
  dump_array((char *) &iv, sizeof(ica_aes_vector_t));
  printf("key\n");
```

```
    dump_array((char *) key, AES_KEY_LEN128);
    printf("\nEncrypted data:\n");
    dump_array((char *) enc_text, sizeof(enc_text));
    printf("\nDecrypted data:\n");
    dump_array((char *) dec_text, sizeof(dec_text));
   }
  }

 // Test 2

  rc = 0;
  bzero(dec_text, sizeof(dec_text));
  bzero(enc_text, sizeof(enc_text));
  bzero(iv, sizeof(iv));
  bzero(key, sizeof(key));

  unsigned int length = 64;
  unsigned char *decrypt = malloc(length);
  unsigned char *encrypt = malloc(length);
  unsigned char *original = malloc(length);
  ica_aes_key_len_128_t key2;

  rc = ica_random_number_generate(length, original);
  if (rc) {
   printf("ica_random_number_generate returned rc = %i\n", rc);
   return rc;
  }

  rc = ica_random_number_generate(AES_KEY_LEN128, (unsigned char *) &key2);
  if (rc) {
   printf("ica_random_number_generate returned rc = %i\n", rc);
   return rc;
  }

  rc = ica_aes_encrypt(mode, length, original, &iv, AES_KEY_LEN128, (unsigned char *) &key2,
         (unsigned char *) encrypt);
  if (rc) {
   printf("\nOriginal data:\n");
   dump_array((char *) original, length);
   printf("KEY: \n");
   dump_array((char *) &key2, AES_KEY_LEN128);
   printf("ica_aes_encrypt failed with errno %d (0x%x).\n", rc, rc);
   return rc;
  }

  if (memcmp(encrypt, original, length) == 0) {
   printf("Encrypt and original are the same.\n");
   return 1;
  }

  bzero(iv, sizeof(iv));
  rc = ica_aes_decrypt(mode, length, encrypt, &iv, AES_KEY_LEN128,
         (unsigned char *) &key2, decrypt);
  if (rc) {
   printf("\nOriginal data:\n");
   dump_array((char *) original, length);
   printf("KEY: \n");
   dump_array((char *) &key2, AES_KEY_LEN128);
   printf("\nEncrypted data:\n");
   dump_array((char *) encrypt, length);
   printf("ica_aes_decrypt failed with errno %d (0x%x).\n", rc, rc);
   goto free;
  }

  if (memcmp(decrypt, original, length) != 0) {
   printf("\nOriginal data:\n");
   dump_array((char *) original, length);
```

```
  printf("KEY: \n");
  dump_array((char *) &key2, AES_KEY_LEN128);
  printf("\nEncrypted data:\n");
  dump_array((char *) encrypt, length);
  printf("\nDecrypted data:\n");
  dump_array((char *) decrypt, length);
  printf("This does NOT match the original data.\n");
  rc = -1;
  goto free;
 }

 if(memcmp(decrypt, encrypt, length) == 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) original, length);
  printf("KEY: \n");
  dump_array((char *) &key2, AES_KEY_LEN128);
  printf("\nEncrypted data:\n");
  dump_array((char *) encrypt, length);
  printf("\nDecrypted data:\n");
  dump_array((char *) decrypt, length);
  printf("decrypt and encrypt are the same\n");
  rc = -1;
  goto free;

    } else {
  printf("Successful!\n");
 }
 free:
  free(original);
  free(encrypt);
  free(decrypt);

   return rc;
}

int main(int argc, char **argv)
{
 // Default mode is 0. ECB and CBC tests will be performed.
 unsigned int mode = 0;
 if (argc > 1) {
  if (strstr(argv[1], "ecb"))
   mode = MODE_ECB;
  if (strstr(argv[1], "cbc"))
   mode = MODE_CBC;
  printf("mode = %i \n", mode);
  }
 if (mode != 0 && mode != MODE_ECB && mode != MODE_CBC) {
  printf("Usage: %s [ ecb | cbc ]\n", argv[0]);
  return -1;
 }
 int rc = 0;
 int error_count = 0;
 if (!mode) {
  silent = 0;
 /* This is the standard loop that will perform all testcases */
  mode = 2;
  while (mode) {
   rc = test_aes128_old_api(mode);
   if (rc) {
    error_count++;
    printf ("test_aes_old_api mode = %i failed \n", mode);
   }
   else
    printf ("test_aes_old_api mode = %i finished successfully \n", mode);

   rc = test_aes128_new_api(mode);
   if (rc) {
```

```
    error_count++;
    printf ("test_aes_new_api mode = %i failed \n", mode);
   }
   else
    printf ("test_aes_new_api mode = %i finished successfully \n", mode);

   mode--;
  }
  if (error_count)
   printf("%i testcases failed\n", error_count);
  else
   printf("All testcases finished successfully\n");
 } else {
 /* Perform only the old test either ein ECB or CBC mode */
  rc = test_aes128_old_api(mode);
  if (rc)
   printf("test_aes_old_api mode = %i failed \n", mode);
  else
   printf("test_aes_old_api mode = %i finished successfully \n", mode);

  rc = test_aes128_new_api(mode);
  if (rc)
   printf ("test_aes_new_api mode = %i failed \n", mode);
  else
   printf ("test_aes_new_api mode = %i finished successfully \n", mode);
 }
 return rc;
}
```

AES 192 example

This is an example using the AES encrypt and decrypt APIs with a key length of 192.

```
/* 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. 2005, 2009          */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

unsigned char NIST_KEY2[] = {
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
};

unsigned char NIST_TEST_DATA[] = {
 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff,
};

unsigned char NIST_TEST_RESULT[] = {
 0xdd, 0xa9, 0x7c, 0xa4, 0x86, 0x4c, 0xdf, 0xe0,
 0x6e, 0xaf, 0x70, 0xa0, 0xec, 0x0d, 0x71, 0x91,
};

int silent = 1;
```

```
void dump_array(char *ptr, int size)
{
 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 test_aes192_old_api(int mode)
{
 ICA_ADAPTER_HANDLE adapter_handle;
 ICA_AES_VECTOR iv;
 ICA_KEY_AES_LEN192 key;
 int rc = 0;
 unsigned char dec_text[sizeof(NIST_TEST_DATA)],
        enc_text[sizeof(NIST_TEST_DATA)];
 unsigned int i;

 bzero(dec_text, sizeof(dec_text));
 bzero(enc_text, sizeof(enc_text));
 bzero(iv, sizeof(iv));
 bcopy(NIST_KEY2, key, sizeof(NIST_KEY2));

 i = sizeof(enc_text);
 rc = icaAesEncrypt(adapter_handle, mode, sizeof(NIST_TEST_DATA),
      NIST_TEST_DATA, &iv, AES_KEY_LEN192,
      (unsigned char *)&key, &i, enc_text);
 if (rc) {
  printf("\nOriginal data:\n");
  dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("icaAesEncrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }
 if (i != sizeof(enc_text)) {
  printf("icaAesEncrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
  return 1;
 }

 if (memcmp(enc_text, NIST_TEST_RESULT, sizeof(NIST_TEST_RESULT)) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char*)enc_text, sizeof(enc_text));
  printf("This does NOT match the known result.\n");
  return 1;
 } else {
  printf("Yep, it's what it should be.\n");
 }

 bzero(iv, sizeof(iv));
 i = sizeof(dec_text);
 rc = icaAesDecrypt(adapter_handle, mode, sizeof(enc_text),
   enc_text, &iv, AES_KEY_LEN192,
   (unsigned char *)&key, &i, dec_text);
```

```
 if (rc != 0) {
  printf("icaAesDecrypt failed with errno %d (0x%x).\n", rc, rc);
  return 1;
 }
 if (i != sizeof(dec_text)) {
  printf("\nOriginal data:\n");
  dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char*)enc_text, sizeof(enc_text));
  printf("\nDecrypted data:\n");
  dump_array((char*)dec_text, sizeof(dec_text));
  printf("icaAesDecrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
  return 1;
 }

 if (memcmp(dec_text, NIST_TEST_DATA, sizeof(NIST_TEST_DATA)) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char*)enc_text, sizeof(enc_text));
  printf("\nDecrypted data:\n");
  dump_array((char*)dec_text, sizeof(dec_text));
  printf("This does NOT match the original data.\n");
  return 1;
 } else {
  printf("Successful!\n");
  if (!silent) {
   printf("\nOriginal data:\n");
   dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
   printf("\nEncrypted data:\n");
   dump_array((char*)enc_text, sizeof(enc_text));
   printf("\nDecrypted data:\n");
   dump_array((char*)dec_text, sizeof(dec_text));
  }
 }

 return 0;
}

int test_aes192_new_api(int mode)
{
 ica_aes_vector_t iv;
 ica_aes_key_len_192_t key;
 int rc = 0;
 unsigned char dec_text[sizeof(NIST_TEST_DATA)],
        enc_text[sizeof(NIST_TEST_DATA)];

 bzero(dec_text, sizeof(dec_text));
 bzero(enc_text, sizeof(enc_text));
 bzero(iv, sizeof(iv));
 bcopy(NIST_KEY2, key, sizeof(NIST_KEY2));

 rc = ica_aes_encrypt(mode, sizeof(NIST_TEST_DATA), NIST_TEST_DATA, &iv,
        AES_KEY_LEN192, (unsigned char *) &key, enc_text);
 if (rc) {
  printf("\nOriginal data:\n");
  dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("ica_aes_encrypt failed with errno %d (0x%x).\n", rc, rc);
  return rc;
 }

 if (memcmp(enc_text, NIST_TEST_RESULT, sizeof(NIST_TEST_RESULT)) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char*)enc_text, sizeof(enc_text));
  printf("This does NOT match the known result.\n");
```

```
  return 1;
 } else {
  printf("Yep, it's what it should be.\n");
 }

 bzero(iv, sizeof(iv));
 rc = ica_aes_decrypt(mode, sizeof(enc_text), enc_text, &iv,
        AES_KEY_LEN192, (unsigned char *) &key, dec_text);
 if (rc != 0) {
  printf("ica_aes_decrypt failed with errno %d (0x%x).\n", rc, rc);
  return 1;
 }

 if (memcmp(dec_text, NIST_TEST_DATA, sizeof(NIST_TEST_DATA)) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char*)enc_text, sizeof(enc_text));
  printf("\nDecrypted data:\n");
  dump_array((char*)dec_text, sizeof(dec_text));
  printf("This does NOT match the original data.\n");
  return 1;
 } else {
  printf("Successful!\n");
  if (!silent) {
   printf("\nOriginal data:\n");
   dump_array((char*)NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
   printf("\nEncrypted data:\n");
   dump_array((char*)enc_text, sizeof(enc_text));
   printf("\nDecrypted data:\n");
   dump_array((char*)dec_text, sizeof(dec_text));
  }
 }

 return 0;
}

int main(int argc, char **argv)
{
 // Default mode is 0. ECB and CBC tests will be performed.
 unsigned int mode = 0;
 if (argc > 1) {
  if (strstr(argv[1], "ecb"))
   mode = MODE_ECB;
  if (strstr(argv[1], "cbc"))
   mode = MODE_CBC;
  printf("mode = %i \n", mode);
  }
 if (mode != 0 && mode != MODE_ECB && mode != MODE_CBC) {
  printf("Usage: %s [ ecb | cbc ]\n", argv[0]);
  return -1;
 }
 int rc = 0;
 int error_count = 0;
 if (!mode) {
 /* This is the standard loop that will perform all testcases */
  mode = 2;
  while (mode) {
   rc = test_aes192_old_api(mode);
   if (rc) {
    error_count++;
    printf ("test_aes_old_api mode = %i failed \n", mode);
   } else
    printf ("test_aes_old_api mode = %i finished successfully \n", mode);

   rc = test_aes192_new_api(mode);
   if (rc) {
```

```
    error_count++;
    printf ("test_aes_new_api mode = %i failed \n", mode);
   } else
    printf ("test_aes_new_api mode = %i finished successfully \n", mode);

   mode--;
  }
  if (error_count)
   printf("%i testcases failed\n", error_count);
  else
   printf("All testcases finished successfully\n");
 } else {
 /* Perform only the old test either ein ECB or CBC mode */
  silent = 0;
  rc = test_aes192_old_api(mode);
  if (rc)
   printf("test_aes_old_api mode = %i failed \n", mode);
  else
   printf("test_aes_old_api mode = %i finished successfully \n", mode);
  rc = test_aes192_new_api(mode);
  if (rc)
   printf ("test_aes_new_api mode = %i failed \n", mode);
  else
   printf ("test_aes_new_api mode = %i finished successfully \n", mode);
 }
 return rc;
}
```

AES 256 example

This is an example using the AES encrypt and decrypt APIs with a key length of 256.

```
/* 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. 2005, 2009          */
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include "ica_api.h"

unsigned char NIST_KEY3[] = {
 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,
};

unsigned char NIST_TEST_DATA[] = {
 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff,
};

unsigned char NIST_TEST_RESULT[] = {
 0x8e, 0xa2, 0xb7, 0xca, 0x51, 0x67, 0x45, 0xbf,
 0xea, 0xfc, 0x49, 0x90, 0x4b, 0x49, 0x60, 0x89,
};

int silent = 1;
```

```
void dump_array(char *ptr, int size)
{
 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 test_aes256_old_api(int mode)
{
 ICA_ADAPTER_HANDLE adapter_handle;
 ICA_AES_VECTOR iv;
 ICA_KEY_AES_LEN256 key;
 int rc = 0;
 unsigned char dec_text[sizeof(NIST_TEST_DATA)],
        enc_text[sizeof(NIST_TEST_DATA)];
 unsigned int i;

 bzero(dec_text, sizeof(dec_text));
 bzero(enc_text, sizeof(enc_text));
 bzero(iv, sizeof(iv));
 bcopy(NIST_KEY3, key, sizeof(NIST_KEY3));

 i = sizeof(enc_text);
 rc = icaAesEncrypt(adapter_handle, mode, sizeof(NIST_TEST_DATA),
      NIST_TEST_DATA, &iv, AES_KEY_LEN256,
      (ICA_KEY_AES_SINGLE *)&key, &i, enc_text);
 if (rc) {
  printf("icaAesEncrypt failed with errno %d (0x%x).\n", rc, rc);
  return 1;
 }
 if (i != sizeof(enc_text)) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof(enc_text));
  printf("icaAesEncrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
  return 1;
 }

 if (memcmp(enc_text, NIST_TEST_RESULT, sizeof(NIST_TEST_RESULT)) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof(enc_text));
  printf("This does NOT match the known result.\n");
  return 1;
 } else {
  printf("Yep, it's what it should be.\n");
 }

 bzero(iv, sizeof(iv));
 i = sizeof(dec_text);
 rc = icaAesDecrypt(adapter_handle, mode, sizeof(enc_text),
```

```
      enc_text, &iv, AES_KEY_LEN256,
      (ICA_KEY_AES_SINGLE *)&key, &i, dec_text);
 if (rc) {
  printf("icaAesDecrypt failed with errno %d (0x%x).\n", rc, rc);
  return 1;
 }
 if (i != sizeof(dec_text)) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof(enc_text));
  printf("\nDecrypted data:\n");
  dump_array((char *) dec_text, sizeof(dec_text));
  printf("icaAesDecrypt returned an incorrect output data length, %u (0x%x).\n", i, i);
  return 1;
 }

 if (memcmp(dec_text, NIST_TEST_DATA, sizeof(NIST_TEST_DATA)) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof(enc_text));
  printf("\nDecrypted data:\n");
  dump_array((char *) dec_text, sizeof(dec_text));
  printf("This does NOT match the original data.\n");
  return 1;
 } else {
  if (!silent) {
   printf("\nOriginal data:\n");
   dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
   printf("\nEncrypted data:\n");
   dump_array((char *) enc_text, sizeof(enc_text));
   printf("\nDecrypted data:\n");
   dump_array((char *) dec_text, sizeof(dec_text));
  }
  printf("Successful!\n");
 }

 return 0;
}

int test_aes256_new_api(int mode)
{
 ica_aes_vector_t iv;
 unsigned char key[AES_KEY_LEN256];
 int rc = 0;
 unsigned char dec_text[sizeof(NIST_TEST_DATA)],
        enc_text[sizeof(NIST_TEST_DATA)];
 unsigned int i;

 bzero(dec_text, sizeof(dec_text));
 bzero(enc_text, sizeof(enc_text));
 bzero(iv, sizeof(iv));
 bcopy(NIST_KEY3, key, sizeof(NIST_KEY3));

 i = sizeof(enc_text);
 rc = ica_aes_encrypt(mode, sizeof(NIST_TEST_DATA), NIST_TEST_DATA, &iv,
        AES_KEY_LEN256, key, enc_text);
 if (rc) {
  printf("ica_aes_encrypt failed with errno %d (0x%x).\n", rc, rc);
  return 1;
 }

 if (memcmp(enc_text, NIST_TEST_RESULT, sizeof(NIST_TEST_RESULT)) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
```

```
  dump_array((char *) enc_text, sizeof(enc_text));
  printf("This does NOT match the known result.\n");
  return 1;
 } else {
  printf("Yep, it's what it should be.\n");
 }

 bzero(iv, sizeof(iv));
 rc = ica_aes_decrypt(mode, sizeof(enc_text), enc_text, &iv,
        AES_KEY_LEN256, key, dec_text);
 if (rc) {
  printf("ica_aes_decrypt failed with errno %d (0x%x).\n", rc, rc);
  return 1;
 }

 if (memcmp(dec_text, NIST_TEST_DATA, sizeof(NIST_TEST_DATA)) != 0) {
  printf("\nOriginal data:\n");
  dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
  printf("\nEncrypted data:\n");
  dump_array((char *) enc_text, sizeof(enc_text));
  printf("\nDecrypted data:\n");
  dump_array((char *) dec_text, sizeof(dec_text));
  printf("This does NOT match the original data.\n");
  return 1;
 } else {
  if (!silent) {
   printf("\nOriginal data:\n");
   dump_array((char *) NIST_TEST_DATA, sizeof(NIST_TEST_DATA));
   printf("\nEncrypted data:\n");
   dump_array((char *) enc_text, sizeof(enc_text));
   printf("\nDecrypted data:\n");
   dump_array((char *) dec_text, sizeof(dec_text));
  }
  printf("Successful!\n");
 }

 return 0;
}

int main(int argc, char **argv)
{
 // Default mode is 0. ECB and CBC tests will be performed.
 unsigned int mode = 0;
 if (argc > 1) {
  if (strstr(argv[1], "ecb"))
   mode = MODE_ECB;
  if (strstr(argv[1], "cbc"))
   mode = MODE_CBC;
  printf("mode = %i \n", mode);
  }
 if (mode != 0 && mode != MODE_ECB && mode != MODE_CBC) {
  printf("Usage: %s [ ecb | cbc ]\n", argv[0]);
  return -1;
 }
 int rc = 0;
 int error_count = 0;
 if (!mode) {
 /* This is the standard loop that will perform all testcases */
  mode = 2;
  while (mode) {
   rc = test_aes256_old_api(mode);
   if (rc) {
    error_count++;
    printf ("test_aes_old_api mode = %i failed \n", mode);
   }
   else
    printf ("test_aes_old_api mode = %i finished successfully \n", mode);
```

```
   rc = test_aes256_new_api(mode);
   if (rc) {
    error_count++;
    printf ("test_aes_new_api mode = %i failed \n", mode);
   }
   else
    printf ("test_aes_new_api mode = %i finished successfully \n", mode);

   mode--;
  }
  if (error_count)
   printf("%i testcases failed\n", error_count);
  else
   printf("All testcases finished successfully\n");
 } else {
 /* Perform only the old test either ein ECB or CBC mode */
  silent = 0;
  rc = test_aes256_old_api(mode);
  if (rc)
   printf("test_aes_old_api mode = %i failed \n", mode);
  else
   printf("test_aes_old_api mode = %i finished successfully \n", mode);

  rc = test_aes256_new_api(mode);
  if (rc)
   printf("test_aes_new_api mode = %i failed \n", mode);
  else
   printf("test_aes_new_api mode = %i finished successfully \n", mode);
 }

 return rc;
}
```

Key generation example

This is an example using 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 *, int);

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;
}

void dump_array(char *ptr, int size)
{
 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");
}
```

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.
 */

/* (C) COPYRIGHT International Business Machines Corp. 2001, 2009          */

#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(char *ptr, int size);

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 = 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, 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(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 = 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, my_result2)) != 0) {
      printf("icaRsaCrt failed and returned %d (0x%x).\n", i, i);
    }

    printf("Result of decrypt\n");
    dump_array(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;
}

void dump_array(char *ptr, int size)
{
   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");
}
```

Makefile example

```
#OPTS = -O0 -g -Wall -fprofile-arcs -ftest-coverage -fPIC
#OPTS = -O0 -g -Wall -m31 -D_LINUX_S390_
OPTS = -O0 -g -Wall -D_LINUX_S390_
INCLUDE = -I. -I../../include
CC = gcc

TARGETS = libica_des_test libica_3des_test \
          libica_aes128_test libica_aes192_test libica_aes256_test \
          libica_sha1_test \
          libica_sha256_test \
          libica_rsa_test libica_rng_test libica_keygen_test \

all: $(TARGETS)

# Every target is created from a single .c file.
%: %.c
 gcc $(OPTS) $(INCLUDE) -lica -lcrypto -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.

Glossary

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 ciphertext by performing an exclusive-OR operation on each 8-byte block of data with the previously encrypted 8-byte block before it is encrypted.

CPACF instructions. Instruction set for the CPACF hardware.

Crypto Express2 (CEX2) . The two PCI-X adapters on a CEX2 feature can be configured in two ways: Either as cryptographic Coprocessor (CEX2C) for secure key encrypted transactions, or as cryptographic Accelerator (CEX2A) for Secure Sockets Layer (SSL) acceleration. A CEX2A works only in clear key mode. Both adapters can be of the same type, or you can configure one adapter as CEX2A and the other as CEX2C.

electronic code book mode (ECB mode). A method of enciphering and deciphering data in address spaces or data spaces. Each 64-bit block of plaintext is separately enciphered and each block of the ciphertext is separately deciphered.

libica. Library for IBM Cryptographic Architecture.

modulus-exponent (Mod-Expo). A type of exponentiation performed using a modulus.

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.

zcrypt. Linux device driver for cryptographic adapters of IBM System z. The libica Version 2 library interacts directly with the zcrpyt device driver.

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 grant 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.

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

Trademarks

IBM, the IBM logo, ibm.com®, eServer, System z, S/390®, zSeries, z9, and System z10 are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at 'Copyright and trademark information at http://www.ibm.com/legal/copytrade.shtml.

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

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

Index

Numerics

A

C

D

E

G

H

I

K

L

M

N

P

R

S

T

W

Readers' Comments — We'd Like to Hear from You

Linux on System z
libica Programmer's Reference
Version 2

Publication No. SC34-2602-00

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 e-mail 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.	E-mail address

Readers' Comments — We'd Like to Hear from You
SC34-2602-00

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 3248
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape

SC34-2602-00

Cut or Fold
Along Line

SC34-2602-00