Embedded SDK (Software Development Kit)

RSA Library

SDK128/D
Rev. 2, 07/23/2002

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Contents

Chapter 4
Building the RSA Library

Chapter 5
Linking Applications with the RSA Library

Chapter 6
RSA Applications

Chapter 7
License

List of Tables

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

List of Figures

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

List of Examples

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

About This Document

This document describes the Rivest, Shamir and Adleman (RSA) algorithm for use with the Embedded Software Development Kit (SDK).

Audience

This document targets software developers implementing encryption/decryption functions within software applications.

Organization

This manual is arranged in the following sections:

- **Chapter 1, Introduction**—provides a brief overview of this document
- **Chapter 2, Directory Structure**—provides a description of the required core directories
- **Chapter 3, RSA Library Interfaces**—describes all of the RSA Library functions
- **Chapter 4, Building the RSA Library**—tells how to execute the system library project build
- **Chapter 5, Linking Applications with the RSA Library**—describes the organization of the RSA Library
- **Chapter 6, RSA Applications**—describes the use of RSA library through test/demo applications
- **Chapter 7, License**—provides the license required to use this product

Suggested Reading

We recommend that you have a copy of the following references:

- *DSP56800 Family Manual*, DSP56800FM/AD
- *DSP56824 User's Manual*, DSP56824UM/AD
- *Inside CodeWarrior: Core Tools*, Metrowerks Corp.

Conventions

This document uses the following notational conventions:

Typeface, Symbol or Term	Meaning	Examples
`Courier Monospaced Type`	Commands, command parameters, code examples, expressions, datatypes, and directives	...`*Foundational include files`... ...a data structure of type `RSA_sConfigure`...
Italic	Calls, functions, statements, procedures, routines, arguments, file names and applications	...the *pConfig* argument... ...defined in the C header file, *rsa.h*... ...makes a call to the *Callback* procedure...
Bold	Reference sources, paths, emphasis	...refer to the **Targeting DSP56824 Platform** manual.... ... see: **C:\Program Files\Motorola\Embedded SDK\help\tutorials**
Bold/Italic	Directory name, project name	...and contains these core directories: ***applications*** contains applications software.... ...CodeWarrior project, ***rsa.mcp***, is.....
Blue Text	Linkable on-line	...refer to **Chapter 7**, License...
Number	Any number is considered a positive value, unless preceded by a minus symbol to signify a negative value	3V -10
ALL CAPITAL LETTERS	Variables, directives, defined constants, files libraries	INCLUDE_DSPFUNC #define INCLUDE_STACK_CHECK
Brackets [...]	Function keys	...by pressing function key [F7]...
Quotation marks "... "	Returned messages	...the message, "Test Passed" is displayed.... ...if unsuccessful for any reason, it will return "NULL"....

Definitions, Acronyms, and Abbreviations

The following list defines the acronyms and abbreviations used in this document. As this template develops, this list will be generated from the document. As we develop more group resources, these acronyms will be easily defined from a common acronym dictionary. Please note that while the acronyms are in solid caps, terms in the definition should be initial capped ONLY IF they are trademarked names or proper nouns.

DSP Digital Signal Processor or Digital Signal Processing

I/O Input/Output

IDE Integrated Development Environment

LSB Least Significant Bit

MAC	Multiply/Accumulate
MIPS	Million Instructions Per Second
MSB	Most Significant Bit
OnCE™	On-Chip Emulation
OMR	Operating Mode Register
PC	Program Counter
RSA	Rivest, Shamir and Adleman
SDK	Software Development Kit
SP	Stack Pointer
SPI	Serial Peripheral Interface
SR	Status Register
SRC	Source

References

The following sources were used to produce this book:

1. *DSP56800 Family Manua*l, DSP56800FM/AD
2. *DSP56824 User's Manual*, DSP56824UM/AD
3. *Embedded SDK Programmer's Guide*

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Chapter 1
Introduction

Welcome to Motorola's family of Digital Signal Processors, (DSPs). This document describes the RSA Library, which is a part of Motorola's comprehensive Software Development Kit (SDK) for its DSPs. In this document, you will find all the information required to use and maintain the RSA Library interface and algorithms.

Motorola provides these algorithms to you for use with Motorola DSPs to expedite your application development and reduce the time it takes to bring your own products to market.

Motorola's RSA Library is licensed for your use on Motorola processors. Please refer to the standard Software License Agreement in **Chapter 7** for license terms and conditions; please consult with your Motorola representative for premium product licensing.

1.1 Quick Start

Motorola's Embedded SDK is targeted to a variety of hardware platforms. To take full advantage of a particular hardware platform, use **Quick Start** from the **Targeting DSP568xx Platform** documentation.

For example, the **Targeting DSP56824 Platform** manual provides more specific information and examples about this hardware architecture. If you are developing an application for the DSP56824EVM board or any other DSP56824 development system, refer to the **Targeting DSP56824 Platform** manual for **Quick Start** or other DSP56824-specific information.

1.2 Background and Overview of RSA

Discrete exponentiation has been employed in a different way by Rivest, Shamir and Adleman (RSA) to produce a public key cryptosystem. They make use of the fact that finding large prime numbers is computationally easy, but that factoring the product of two such numbers appears to be computationally infeasible.

A user, A, selects two large prime numbers at random, p and q, and multiplies them together to obtain a number, n. The number n is made public, but the factors p and q are kept secret. Using p and q, User A can compute the Euler's Totient function:

$\Phi(n)$ (the number of integers less then n and relatively prime to n as)

$\Phi(n) = (p-1)(q-1)$

User A then chooses another number, e, at random from the interval 2 through $\Phi(n)$, such that e and $\Phi(n)$ are relatively prime. This number is also made public. Enciphering is carried out on each block as m, using the public information e and n as:

$$c = m^e \pmod{n}$$

In this example, c represents the ciphertext. Using the secret number $\Phi(n)$, User A can easily calculate a number, d, such that:

$$(e.d) \bmod \Phi(n) = 1$$

If e has a common factor with $\Phi(n)$ then d does not exist. User B receives c and computes $c^d \bmod (n)$ which is equivalent to

$$m.\exp(k\Phi(n)+1) \pmod{n}$$

for some k. From Euler's theorem, $x.\exp(\Phi(n)) \pmod{n} = 1$, if x and n are relatively prime. Also, $x.\exp(k\Phi(n)+1) \pmod{n} = x$, for any integer x lying between 0 and n-1, where n=pq. Hence, User B gets the original message, m, by exponentiating c with d and finding modulo with respect to n.

Similarly, we can explain the signaturing process with the following equations:

$$m^d \pmod{n} = s$$

$$m = s^e \pmod{n}$$

1.3 Features and Performance

The RSA library is multichannel and re-entrant.

For details on Memory and MIPS for a particular DSP, refer to the **Libraries** chapter of the appropriate Targeting manual.

Chapter 2
Directory Structure

2.1 Required Core Directories

Figure 2-1 details required platform directories:

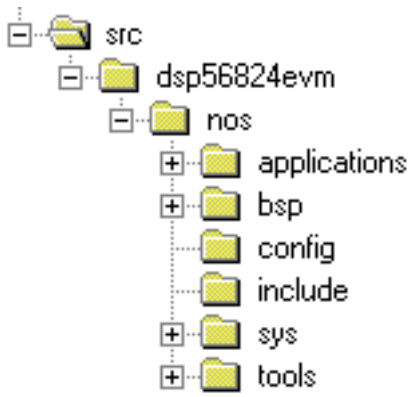

Figure 2-1. Core Directories

As shown in Figure 2-1, DSP56824EVM has no operating system (nos) support, and contains these core directories:

- ***applications*** contains applications software that can be exercised on this platform
- ***bsp*** contains the board support package specific for this platform
- ***config*** contains default hardware/software configurations for this platform
- ***include*** contains SDK header files which define the Application Programming Interface
- ***sys*** contains required system components
- ***tools*** contains utilities used by system components

There are also optional directories that include domain-specific libraries.

2.2 Optional (Domain-Specific) Directories

As shown in Figure 2-2, the ***Security*** directory contains algorithms specific to RSA.

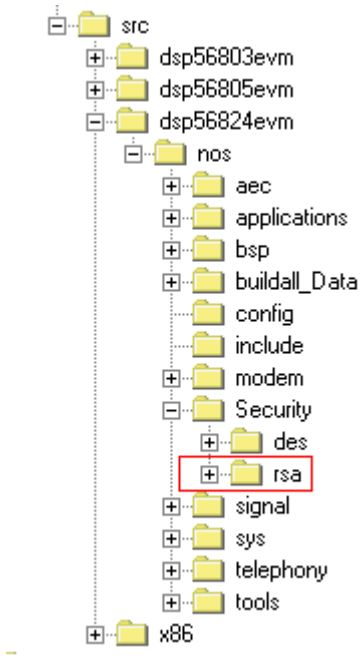

Figure 2-2. *Security* Directory

Figure 2-3 shows details of the ***rsa*** directory structure.

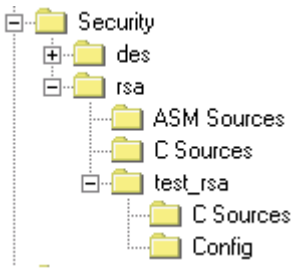

Figure 2-3. *rsa* Directory Structure

The ***rsa*** directory includes the following sub-directories:

- ***ASM Source*** contains all asm sources required for RSA
- ***C Sources*** includes APIs for RSA
- ***test_rsa*** includes C source files and configuration necessary for testing RSA library modules
  - — ***C Sources*** contains an example test code for RSA
  - — ***Config*** contains the configuration files ***appconfig.c***, ***appconfig.h*** and ***linker.cmd*** specific to RSA.

The ***applications*** directory includes high-level software that exercises the RSA library, including the ***rsa_demo*** application, detailed in Figure 2-4.

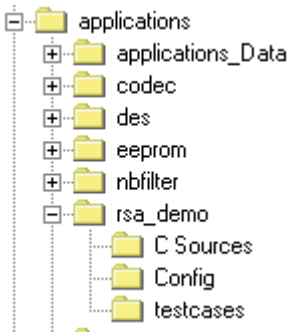

Figure 2-4. *rsa_demo* Application

ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005
Freescale Semiconductor, Inc.

Chapter 3
RSA Library Interfaces

3.1 RSA Services

The RSA library encrypts/decrypts user-supplied data. The data to be supplied must be in words, the format of which is given below (assuming the word size to be 16 bits).

i MSB	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i LSB

In this example, “i” stands for information bit. The user can supply data in bytes by making the remaining bits zero. The format is shown below:

0 MSB	0	0	0	0	0	0	0	i	i	i	i	i	i	i	i LSB

3.2 Interface

The C interface for RSA library services is defined in the C header file *rsa.h,* shown in **Code Example 3-1** as a reference.

Code Example 3-1. C Header File *rsa.h*

```
/* File rsa.h */

#ifndef _RSA_H
#define _RSA_H

/*
   This include file is the master include file for the
   RSA. The applications using RSA should include this file
*/
```

```
/**************************
 Foundational Include Files
**************************/

#include "port.h"


/*********************************************
     Structure for RSA Configuration
*********************************************/

typedef struct
{
    void (*pCallback) (void *pCallbackArg, Word16 *pWords,
                       UWord16 NumberWords);
    void *pCallbackArg;
}RSA_sCallback;

typedef struct
{
    UInt16 RsaModNLen;        /* Length of Modulo buffer N */
    Word16 *RsaN;             /* Pointer to modulo buffer N */
    UInt16 RsaELen;           /* Length of encryptor exponent buffer E */
    Word16 *RsaE;             /* Pointer to encryptor exponent buffer E */
    UInt16 RsaVLen;           /* Length of decryptor exponent buffer V */
    Word16 *RsaV;             /* Pointer to decryptor exponent buffer V */
    RSA_sCallback Callback;

}RSA_sConfigure;

typedef struct
{
    UInt16 RsaModNLen;        /* Length of Modulo buffer N */
    Word16 *RsaN;             /* Pointer to modulo buffer N */
    UInt16 RsaELen;           /* Length of encryptor exponent buffer E */
    Word16 *RsaE;             /* Pointer to encryptor exponent buffer E */
    Word16 *Buffer;           /* Pointer to Encode related buffers */
    Word16 *pOutBuf;          /* Output buffer for encryptor */
    Word16 *ContextBuff;
    UWord16 Length;           /* Context buffer length */
    RSA_sCallback *EncCallback;

    Word16 EN;
    Word16 EK;
    Word16 EKORG;
    Word16 ELEN;
    Word16 mb;
    Word16 s1;
    Word16 ek;
    Word16 tulen;
    Word16 tvlen;
    Word16 tla;
    Word16 tlc;
    Word16 TEMPX;
    Word16 ADD_Y0;
```

```
    Word16 ADD_N1;
    Word16 count1;
    Word16 count2;
    Word16 count3;
    Word16 count4;
    Word16 N0INV;
    Word16 temp_store;
    Word16 ADD_b0;
    Word16 ADD_b1;
    Word16 pm;
    Word16 PMETBL;
    Word16 ps;
    Word16 Persqrs;
    Word16 pssa;
    Word16 N_1;
    Word16 pp;
    Word16 pq;
    Word16 pe;
    Word16 pv;
    Word16 Pzc;
    Word16 pbufu;
    Word16 pbufv;
    Word16 PEXP;
    Word16 RES;
    Word16 IADDR;
    Word16 M;
    Word16 kd;
    Word16 DK;
    Word16 DKORG;
    Word16 DM;
    Word16 DLEN;
    Word16 d1;
    Word16 dmb;
    Word16 dk;
    Word16 Pv;
    Word16 Pd;
    Word16 pdec_msg;
    Word16 Pdrsqrd;
    Word16 Pdsa;
    Word16 ptemp;
    Word16 Pend;
    Word16 temp;
    Word16 ulen;
    Word16 v1;
    Word16 v2;
    Word16 shifts;
    Word16 inv[2];
    Word16 TEMP;
    Word16 EL;

}RSA_sEncHandle;

typedef struct
{
```

```
    UInt16 RsaModDLen;        /* Length of Modulo buffer D */
    Word16 *RsaD;             /* Pointer to modulo buffer D */
    UInt16 RsaVLen;           /* Length of decryptor exponent buffer V */
    Word16 *RsaV;             /* Pointer to decryptor exponent buffer V */
    Word16 *Buffer;           /* Pointer to decode related buffers */
    Word16 *pOutBuf;          /* Output buffer for decryptor */
    Word16 *ContextBuff;
    UWord16 Length;           /* Context buffer length */
    RSA_sCallback *DecCallback;

    Word16 EN;
    Word16 EK;
    Word16 EKORG;
    Word16 ELEN;
    Word16 mb;
    Word16 s1;
    Word16 ek;
    Word16 tulen;
    Word16 tvlen;
    Word16 tla;
    Word16 tlc;
    Word16 TEMPX;
    Word16 ADD_Y0;
    Word16 ADD_N1;
    Word16 count1;
    Word16 count2;
    Word16 count3;
    Word16 count4;
    Word16 N0INV;
    Word16 temp_store;
    Word16 ADD_b0;
    Word16 ADD_b1;
    Word16 pm;
    Word16 PMETBL;
    Word16 ps;
    Word16 Persqrs;
    Word16 pssa;
    Word16 N_1;
    Word16 pp;
    Word16 pq;
    Word16 pe;
    Word16 pv;
    Word16 Pzc;
    Word16 pbufu;
    Word16 pbufv;
    Word16 PEXP;
    Word16 RES;
    Word16 IADDR;
    Word16 M;
    Word16 kd;
    Word16 DK;
    Word16 DKORG;
    Word16 DM;
    Word16 DLEN;
    Word16 d1;
```

```
    Word16 dmb;
    Word16 dk;
    Word16 Pv;
    Word16 Pd;
    Word16 pdec_msg;
    Word16 Pdrsqrd;
    Word16 Pdsa;
    Word16 ptemp;
    Word16 Pend;
    Word16 temp;
    Word16 ulen;
    Word16 v1;
    Word16 v2;
    Word16 shifts;
    Word16 inv[2];
    Word16 TEMP;
    Word16 EL;

}RSA_sDecHandle;

/*--------------------------------------
 * Commands for RSA Control
 *--------------------------------------*/

#define RSA_DEACTIVATE  2

/**************************
 Function Prototypes
**************************/

EXPORT RSA_sEncHandle *rsaEncCreate (RSA_sConfigure *pConfig);

EXPORT RSA_sDecHandle *rsaDecCreate (RSA_sConfigure *pConfig);


EXPORT Result rsaEncInit (RSA_sEncHandle *pRsaEnc, RSA_sConfigure *pConfig);

EXPORT Result rsaDecInit (RSA_sDecHandle *pRsaDec, RSA_sConfigure *pConfig);

EXPORT Result rsaEncrypt (RSA_sEncHandle *pRsaEnc, Word16 *pInWords,
                          UWord16 NumberWords);

EXPORT Result rsaDecrypt (RSA_sDecHandle *pRsaDec, Word16 *pInWords,
                                           UWord16 NumberWords);

EXPORT void rsaEncDestroy (RSA_sEncHandle *pRsaEnc);

EXPORT void rsaDecDestroy (RSA_sDecHandle *pRsaDec);

EXPORT Result rsaEncControl (RSA_sEncHandle *pRsaEnc, UWord16 Command);

EXPORT Result rsaDecControl (RSA_sDecHandle *pRsaDec, UWord16 Command);

#endif
```

3.3 Specifications

The following pages characterize the RSA library functions.

Function arguments for each routine are described as *in*, *out*, or *inout*. An *in* argument means that the parameter value is an input only to the function. An *out* argument means that the parameter value is an output only from the function. An *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, *inout* parameters are input pointer variables in which the caller passes the address of a preallocated data structure to a function. The function stores its results within that data structure. The actual value of the *inout* pointer parameter is not changed.

3.3.1 *rsaEncCreate*

Call(s):

```
RSA_sEncHandle *rsaEncCreate (RSA_sConfigure *pConfig);
```

Required Header: "rsa.h"

Arguments:

Table 3-1. *rsaEncCreate* Arguments

pConfig	in	Points to the configuration data for RSA

Description: The *rsaEncCreate* function creates an instance for RSA encryption. The *pConfig* argument points to the *RSA_sConfigure* structure used to configure RSA operation. For initialization of the *RSA_sConfigure* structure, refer to *rsaEncInit*, **Section 3.3.2**. During the *rsaEncCreate* call, any dynamic resources required by the RSA encryption algorithm are allocated. The memory allocated is (153 + mod_len*26) external data memory words; mod_len is the length of modulo N buffer in words. The library allocates memory dynamically using the *mem* library shown in **Code Example 3-2**. The RSA library is **multichannel** and **re-entrant**.

Code Example 3-2. *mem* Library

```
#include "rsa.h"
#include "mem.h"

RSA_sEncHandle *rsaEncCreate (RSA_sConfigure *pConfig)
{
    RSA_sEncHandle *pRsaEnc;
    Word16 EKORG_3, psize;

    /* Memory allocation for RSA Handle */
    pRsaEnc = (RSA_sEncHandle *) memMallocEM (sizeof (RSA_sEncHandle));
    if (pRsaEnc == NULL) return (NULL);

    EKORG_3 = (pConfig->RsaModNLen+15)>>4;
    EKORG_3 = EKORG_3 + 3;

    /* pOutBuf */
    pRsaEnc->pOutBuf = (Word16 *) memMallocEM (EKORG_3 * sizeof (Word16));

    /* ContextBuff */
    pRsaEnc->ContextBuff = (Word16 *) memMallocEM (EKORG_3 * sizeof (Word16));

    EKORG_3 = (EKORG_3 * 24) + 1;

    /* Enc_buffer */
    pRsaEnc->Buffer = (Word16 *) memMallocEM (EKORG_3 * sizeof (Word16));

    /* Callback structure */
    pRsaEnc->EncCallback = (RSA_sCallback *) memMallocEM (sizeof (RSA_sCallback));
```

```
    if (!(pRsaEnc->pOutBuf && pRsaEnc->ContextBuff &&
          pRsaEnc->Buffer && pRsaEnc->EncCallback))
    {
        rsaEncDestroy (pRsaEnc);
        return (NULL);
    }

    /* Call RSA initialization */
    rsaEncInit (pRsaEnc, pConfig);

    return (pRsaEnc);
}
```

For details on the *RSA_sEncHandle* structure, please refer to **Code Example 3-1**.

If an *rsaEncCreate* function is called to create an instance, then *rsaEncDestroy*, **Section 3.3.10**, should be used to destroy the instance.

Alternatively, the user can allocate memory statically, which requires duplicating all statements in the *rsaEncCreate* function. In this case, the user can call the *rsaEncInit* function directly, bypassing the *rsaEncCreate* function. If the user dynamically allocates memory without calling *rsaEncCreate*, then the user himself must destroy the memory allocated.

Returns: Upon successful completion, the *rsaEncCreate* function will return a pointer to the specific instance of RSA encryptor created. If *rsaEncCreate* is unsuccessful for any reason, it will return “NULL”.

Special Considerations:

- The RSA application is multichannel and re-entrant.
- If *rsaEncCreate* is called, then the user need not call *rsaEncInit* function, as it is called internally in the *rsaEncCreate* function.
- The user must specify the length of modulo buffers N and E in bits during initialization of configuration structure

In **Code Example 3-3**, the application creates an instance of RSA encryptor.

Code Example 3-3. Use of *rsaEncCreate* Interface

```
#include “rsa.h”
#include "mem.h"

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 e[] = { 0xcda7,0xda5d,0xdd3f,0x3faa,0x3227,0x4cdc,0x149e,
               0xdbf1,0x25a9,0x2d0f,0xcc3f,0x72d7,0x5f95,0x16a8,
```

```
                0x6a3f,0x81c9,0xdc92,0xc9d0,0x51f4,0x5af6,0x5f6c,
                0x49dd,0x783a,0xc9f0,0xaaaa,0xaaaa,0xaaaa,0xaaaa,
                0xaaaa,0xaaaa,0xaaaa,0x2aaa
              };

Frac16 message[] = {
                      0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                      0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                      0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                      0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                      0x5522,0x8833,0x9944,0xc455
                    };
/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);
/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sEncHandle *pRsaEnc;

void testRsa ()
{
              Result res;

/* Allocate memory for RSA_sConfigure structure */
pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
if (pConfig == NULL)
              assert (!"Memory error");
....
/* Initialize the pConfig structure */
pConfig->RsaModNLen = 513; /* in bits = 33 words */
pConfig->RsaN = n;
pConfig->RsaELen = 512;   /* in bits = 32 words */
pConfig->RsaE = e;
pConfig->Callback.pCallback = Callback;
....
pRsaEnc = rsaEncCreate (pConfig); /* Create and initialize the RSA instance. */

....
}
```

3.3.2 *rsaEncInit*

Call(s):

```
Result rsaEncInit (RSA_sEncHandle *pRsaEnc, RSA_sConfigure *pConfig);
```

Required Header: "rsa.h"

Arguments:

Table 3-2. *rsaEncInit* Arguments

pRsaEnc	in	Handle to an instance of RSA encryptor generated by a call to *rsaEncCreate*
pConfig	in	A pointer to a data structure containing data for initializing the RSA algorithm

Description: The *rsaEncInit* function will initialize the RSA encryptor algorithm. During the initialization, all resources will be set to their initial values in preparation for RSA encryption operation. Before calling *rsaEncInit* function, an RSA encryptor instance must be created. The RSA encryptor instance, *pRsaEnc*, can be created by either calling the *rsaEncCreate* function, shown in **Section 3.3.1**, or by statically allocating memory, which does not require calling the *rsaEncCreate* function.

The parameter *pConfig* points to a data structure of type `RSA_sConfigure`; its fields initialize RSA encryptor operation in the following manner:

RsaModNLen - length of modulo N buffer in bits

RsaN - pointer to a *modulo N buffer* (see *rsa.h* file for details)

RsaELen - length of encryption buffer E in bits

RsaE - pointer to a buffer used for encryption

Callback - a structure of type *RSA_sCallback*; it describes the procedure which RSA will call once the data bytes are processed (encrypted) by the algorithm. The callback procedure has the following declaration:

```
void (*pCallback) (void *pCallbackArg, Word16 *pWords, UWord16
                   NumberWords);
```

The callback procedure parameter, ***pCallbackArg***, is supplied by the user in the *RSA_sCallback* structure; this value is passed back to the user during the call to the Callback procedure. Typically, *pCallbackArg* points to context information used by the callback procedure, which theuser must write; see **Code Example 3-4**.

pWords - pointer to input data words

NumberWords - Number of words in the input data buffer pointed to by *pWords*

Code Example 3-4. Sample Callback Procedure

```
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords)
{
    Int16 i;

    for (i = 0; i < NumberWords; i++)
    {
       Data_Out[i]  = pWords[i];
    }
    flag = 1;
    return;
}
```

Since this function must be written by the user, its content depends on how the user wants to process the RSA encryptor output.

Returns: Upon successful completion, a value of “TRUE” will be returned. If pConfig->RsaModNLen = 0, a value of “FALSE” will be returned.

Special Considerations:

- If *rsaEncCreate* is called, then the user need not call *rsaEncInit* function, as it is called internally in the *rsaEncCreate* function.

In **Code Example 3-5**, the application creates an instance of RSA encryptor. The instance is passed to *rsaEncInit* along with RSA configuration structure *pConfig*.

Code Example 3-5. Use of *rsaEncInit* Interface

```
#include “rsa.h”
#include “mem.h”

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 e[] = { 0xcda7,0xda5d,0xdd3f,0x3faa,0x3227,0x4cdc,0x149e,
               0xdbf1,0x25a9,0x2d0f,0xcc3f,0x72d7,0x5f95,0x16a8,
               0x6a3f,0x81c9,0xdc92,0xc9d0,0x51f4,0x5af6,0x5f6c,
               0x49dd,0x783a,0xc9f0,0xaaaa,0xaaaa,0xaaaa,0xaaaa,
               0xaaaa,0xaaaa,0xaaaa,0x2aaa
             };

Frac16 message[] = {
0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                     0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                     0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                     0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
```

```
                    0x5522,0x8833,0x9944,0xc455
                  };

/* output buffers */
Word16 Data_Out[NUMWORDS + 2];
Int16 flag; /* Flag to indicate data is available in o/p vector */

/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);

/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sEncHandle *pRsaEnc;

void testRsa ()
{
Result res;

/* Allocate memory for RSA_sConfigure structure */
             pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
             if (pConfig == NULL)
                   assert (!"Memory error");

             /* Initialise pConfig structure */
             pConfig->RsaModNLen = 0x0201;
             pConfig->RsaN = n;
             pConfig->RsaELen = 0x0200;
             pConfig->RsaE = e;
             pConfig->Callback.pCallback = Callback;

             /* Create and init Encode handle */
             pRsaEnc = rsaEncCreate (pConfig);
             if (pRsaEnc == NULL)
                   assert (!" Encode create failed");

             .......

}
```

3.3.3 *rsaEncrypt*

Call(s):

```
Result rsaEncrypt (RSA_sEncHandle *pRsaEnc, Word16 *pInWords, UWord16
                   NumberWords);
```

Required Header: “rsa.h”

Arguments:

Table 3-3. *rsaEncrypt* Arguments

pRsaEnc	in	Handle to an instance of RSA encryptor generated by a call to *rsaEncCreate*
pInWords	in	Pointer to the user data words to be encrypted by the RSA algorithm
NumberWords	in	The number of data words to be encrypted

Description: The *rsaEncrypt* function will encrypt the data supplied. Once encryption is complete, the result is returned to the user by calling the *Callback* procedure. The user can call the *rsaEncrypt* function any number of times, as long as the user has data.

Returns: “PASS”

Special Considerations:

- Inplace computation is allowed; *i.e.*, input and output buffers could be the same
- The *rsaEncrypt* function makes a call to the *Callback* procedure only when max_message_len = {(pConfig->RsaModNLen+2) >> 4} of data are encrypted
- Suppose 65 words of data have to be encrypted and max_message_len = 32. The length to be passed for encryption is shown below:

  ```
  rsaEncrypt (pRsaEnc, pInWords, 65); /* for encryption */
  ```

  The *rsaEncrypt* function will make two calls to callback, giving 64 words of encrypted data; the remaining word is put into the context buffer. At the end of encryption, a call to *rsaEncControl* should be made to flush out any remaining data.
- The output buffer must be a multiple of max_message_len = {(pConfig->RsaModNLen+2) >> 4}

Code Example: In **Code Example 3-6**, during encryption, the **total** length of input data is (32 + 65 + 100) = 197 words. It’s not a multiple of 32, so the encryptor will return 192 words of encrypted data and will hold the remaining 5 words. At the end, *rsaEncDestroy* is called, which in turn calls *rsaEncControl* to flush out this data and to destroy the instance of encryptor. The encryptor will generate 32 words of data after a call to *rsaEncDestroy*. The total encrypted data length is 192 + 32 = 224, which exceeds the input data length, since the input data length is not a multiple of 32.

Code Example 3-6. Use of *rsaEncrypt* Interface

```
#include “rsa.h”
#include “mem.h”

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 e[] = { 0xcda7,0xda5d,0xdd3f,0x3faa,0x3227,0x4cdc,0x149e,
               0xdbf1,0x25a9,0x2d0f,0xcc3f,0x72d7,0x5f95,0x16a8,
               0x6a3f,0x81c9,0xdc92,0xc9d0,0x51f4,0x5af6,0x5f6c,
               0x49dd,0x783a,0xc9f0,0xaaaa,0xaaaa,0xaaaa,0xaaaa,
               0xaaaa,0xaaaa,0xaaaa,0x2aaa
             };

Frac16 message[] = {
                     0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                     0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                     0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                     0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                     0x5522,0x8833,0x9944,0xc455
                   };

/* output buffers */
Word16 Data_Out[NUMWORDS + 2];
Int16 flag; /* Flag to indicate data is available in o/p vector */

/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);

/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sEncHandle *pRsaEnc;

void testRsa ()
{
Result res;

/* Allocate memory for RSA_sConfigure structure */
             pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
             if (pConfig == NULL)
                   assert (!"Memory error");

             /* Initialise pConfig structure */
             pConfig->RsaModNLen = 0x0201;
             pConfig->RsaN = n;
             pConfig->RsaELen = 0x0200;
             pConfig->RsaE = e;
             pConfig->Callback.pCallback = Callback;
```

```
                /* Create and init Encode handle */
                pRsaEnc = rsaEncCreate (pConfig);
                if (pRsaEnc == NULL)
                      assert (!" Encode create failed");

.....
res = rsaEncrypt (pRsaEnc, pInWords, 32);

...

res = rsaEncrypt (pRsaEnc, pInWords, 65);

...

res = rsaEncrypt (pRsaEnc, pInWords, 100);

...

rsaEncDestroy (pRsaEnc); /* destroy calls rsaEncControl before destroying the
                                    instance */

...

}
```

3.3.4 *rsaEncControl*

Call(s):

Result rsaEncControl (RSA_sEncHandle *pRsaEnc, UWord16 Command);

Required Header: "rsa.h"

Arguments:

Table 3-4. *rsaEncControl* Arguments

pRsaEnc	in	Handle to an instance of RSA encryptor generated by a call to *rsaEncCreate*
Command	in	The command to be executed by the *rsaEncControl* procedure

Description: The *rsaEncControl* function provides control functions to the RSA encryption algorithm. When in the process of encryption, if RSA encryption must be terminated, the user can do so by calling the *rsaEncControl* function. The *rsaEncControl* function flushes the data to be encrypted, even if it doesn't form a proper block required for RSA Encryption operation, **by appending zeros**, encrypting and then calling the *Callback* procedure.

The parameter *pRsaEnc* must have been generated from a call to *rsaEncCreate*. The parameter *Command* determines which action the *rsaEncControl* algorithm will perform, including:

RSA_DEACTIVATE - Deactivate the RSA Encryption operation.

Returns: Upon successful completion, *rsaEncControl* will return "PASS"; otherwise, "FAIL" is returned.

Special Considerations: Calling the *rsaEncControl* function does not free the memory allocated during the *rsaEncCreate* function. To deallocate buffers, the *rsaEncDestroy* function must be called only if *rsaEncCreate* function was used to create the instance. If user himself bypassed the *rsaEncCreate* function to create the instance, then the user must free the memory.

Code Example 3-7. Use of *rsaEncControl* Interface

```
#include "rsa.h"
#include "mem.h"

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 e[] = { 0xcda7,0xda5d,0xdd3f,0x3faa,0x3227,0x4cdc,0x149e,
               0xdbf1,0x25a9,0x2d0f,0xcc3f,0x72d7,0x5f95,0x16a8,
               0x6a3f,0x81c9,0xdc92,0xc9d0,0x51f4,0x5af6,0x5f6c,
               0x49dd,0x783a,0xc9f0,0xaaaa,0xaaaa,0xaaaa,0xaaaa,
               0xaaaa,0xaaaa,0xaaaa,0x2aaa
             };
```

```
Frac16 message[] = {
                      0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                      0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                      0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                      0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                      0x5522,0x8833,0x9944,0xc455
                    };

/* output buffers */
Word16 Data_Out[NUMWORDS + 2];
Int16 flag; /* Flag to indicate data is available in o/p vector */

/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);

/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sEncHandle *pRsaEnc;

void testRsa ()
{
Result res;
UWord16 Command = RSA_DEACTIVATE;

/* Allocate memory for RSA_sConfigure structure */
             pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
             if (pConfig == NULL)
                   assert (!"Memory error");

             /* Initialise pConfig structure */
             pConfig->RsaModNLen = 0x0201;
             pConfig->RsaN = n;
             pConfig->RsaELen = 0x0200;
             pConfig->RsaE = e;
             pConfig->Callback.pCallback = Callback;

             /* Create and init Encode handle */
             pRsaEnc = rsaEncCreate (pConfig);
             if (pRsaEnc == NULL)
                   assert (!" Encode create failed");


......
res = rsaEncrypt (pRsaEnc, pInWords, 32);

...

res = rsaEncrypt (pRsaEnc, pInWords, 65);

...

res = rsaEncrypt (pRsaEnc, pInWords, 100);

...
```

```
res = rsaEncControl (pRsaEnc, Command);
....

}
```

3.3.5 *rsaEncDestroy*

Call(s):

void rsaEncDestroy (RSA_sEncHandle *pRsaEnc);

Required Header: "rsa.h"

Arguments:

Table 3-5. *rsaEncDestroy* Arguments

pRsaEnc	in	Handle to an instance of RSA encryptor generated by a call to *rsaEncCreate*

Description: The *rsaEncDestroy* function destroys the instance of the RSA encryptor originally created by a call to *rsaEncCreate*. It also internally calls *rsaEncControl* to complete the operation on any residual non-complete blocks by appending zeros.

Returns: None

Special Considerations: The function *rsaEncDestroy* calls *rsaEncControl*, then frees the memory allocated during the *rsaEncCreate* function. Hence, *rsaEncDestroy* deactivates RSA and frees the memory allocated during the *rsaEncCreate* function.

Code Example 3-8. Use of *rsaEncDestroy* Interface

```
#include "rsa.h"
#include "mem.h"

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 e[] = { 0xcda7,0xda5d,0xdd3f,0x3faa,0x3227,0x4cdc,0x149e,
               0xdbf1,0x25a9,0x2d0f,0xcc3f,0x72d7,0x5f95,0x16a8,
               0x6a3f,0x81c9,0xdc92,0xc9d0,0x51f4,0x5af6,0x5f6c,
               0x49dd,0x783a,0xc9f0,0xaaaa,0xaaaa,0xaaaa,0xaaaa,
               0xaaaa,0xaaaa,0xaaaa,0x2aaa
             };

Frac16 message[] = {
                     0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                     0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                     0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                     0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                     0x5522,0x8833,0x9944,0xc455
                   };

/* output buffers */
Word16 Data_Out[NUMWORDS + 2];
Int16 flag; /* Flag to indicate data is available in o/p vector */
```

```
/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);

/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sEncHandle *pRsaEnc;

void testRsa ()
{
Result res;

/* Allocate memory for RSA_sConfigure structure */
             pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
             if (pConfig == NULL)
                   assert (!"Memory error");

             /* Initialise pConfig structure */
             pConfig->RsaModNLen = 0x0201;
             pConfig->RsaN = n;
             pConfig->RsaELen = 0x0200;
             pConfig->RsaE = e;
             pConfig->Callback.pCallback = Callback;

             /* Create and init Encode handle */
             pRsaEnc = rsaEncCreate (pConfig);
             if (pRsaEnc == NULL)
                   assert (!" Encode create failed");

......
res = rsaEncrypt (pRsaEnc, pInWords, 32);

...

res = rsaEncrypt (pRsaEnc, pInWords, 65);

...

res = rsaEncrypt (pRsaEnc, pInWords, 100);

...

rsaEncDestroy (pRsaEnc);

....
}
```

3.3.6 *rsaDecCreate*

Call(s):

```
RSA_sDecHandle *rsaDecCreate (RSA_sConfigure *pConfig);
```

Required Header: "rsa.h"

Arguments:

Table 3-6. *rsaDecCreate* Arguments

pConfig	in	Points to the configuration data for RSA

Description: The *rsaDecCreate* function creates an instance for RSA decryption. The *pConfig* argument points to the `RSA_sConfigure` structure used to configure RSA decryptor operation. For initialization of the `RSA_sConfigure` structure, refer to *rsaDecInit*, **Section 3.3.7**. During the *rsaDecCreate* call, any dynamic resources required by the RSA decryption algorithm are allocated. The memory allocated is (153 + 26*mod_len) external data memory words; mod_len is the length of modulo N buffer in words. The RSA library is **multichannel** and **re-entrant**.

The library allocates memory dynamically using the *mem* library shown in **Code Example 3-9**.

Code Example 3-9. *mem* Library

```
#include "rsa.h"
#include "mem.h"

RSA_sDecHandle *rsaDecCreate (RSA_sConfigure *pConfig)
{
    RSA_sDecHandle *pRsaDec;
    Word16 DKORG_3, psize;

    /* Memory allocation for RSA Handle */
    pRsaDec = (RSA_sDecHandle *) memMallocEM (sizeof (RSA_sDecHandle));
    if (pRsaDec == NULL) return (NULL);

    DKORG_3 = (pConfig->RsaModNLen+15)>>4;
    DKORG_3 = DKORG_3 + 3;

    /* pOutBuf */
    pRsaDec->pOutBuf = (Word16 *) memMallocEM (DKORG_3 * sizeof (Word16));

    /* ContextBuff */
    pRsaDec->ContextBuff = (Word16 *) memMallocEM (DKORG_3 * sizeof (Word16));

    DKORG_3 = (DKORG_3 * 24) + 1;

    /* Dec_buffer */
    pRsaDec->Buffer = (Word16 *) memMallocEM (DKORG_3 * sizeof (Word16));

    /* Callback structure */
    pRsaDec->DecCallback = (RSA_sCallback *) memMallocEM (sizeof (RSA_sCallback));
```

```
    if (!(pRsaDec->pOutBuf && pRsaDec->ContextBuff &&
          pRsaDec->Buffer && pRsaDec->DecCallback))
    {
        rsaDecDestroy (pRsaDec);
        return (NULL);
    }

    /* Call RSA initialization */
    rsaDecInit (pRsaDec, pConfig);

    return (pRsaDec);
}
```

For details on the *RSA_sDecHandle* structure, see **Code Example 3-1**.

If an *rsaDecCreate* function is called to create an instance, then *rsaDecDestroy*, **Section 3.3.10**, should be used to destroy the instance.

Alternatively, the user can allocate memory statically which requires duplicating all statements in the *rsaDecCreate* function. In this case, the user can call the *rsaDecInit* function directly, bypassing the *rsaDecCreate* function. If the user dynamically allocates memory without calling *rsaDecCreate*, then the user himself must destroy the memory allocated.

Returns: Upon successful completion, the *rsaDecCreate* function will return a pointer to the specific instance of RSA decryptor created. If *rsaDecCreate* is unsuccessful for any reason, it will return “NULL”.

Special Considerations:

- The RSA application is multichannel and re-entrant
- If *rsaDecCreate* is called, then the user need not call *rsaDecInit* function, as it is called internally in the *rsaDecCreate* function
- The user must specify the length of modulo buffer N and V in bits during initialization of configuration structure

In **Code Example 3-10**, the application creates an instance of RSA decryptor.

Code Example 3-10. Use of *rsaDecCreate* Interface

```
#include “rsa.h”
#include "mem.h"

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };

Frac16 v  =  0x0003;
```

```
Frac16 message[] = {
                     0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                     0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                     0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                     0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                     0x5522,0x8833,0x9944,0xc455
                   };
/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);
/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sDecHandle *pRsaDec;

void testRsa ()
{
             Result res;

/* Allocate memory for RSA_sConfigure structure */
pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
if (pConfig == NULL)
             assert (!"Memory error");
....
/* Initialize the pConfig structure */
pConfig->RsaModNLen = 513; /* in bits = 33 words */
pConfig->RsaN = n;
pConfig->RsaVLen = 2;
pConfig->RsaV = &v;
pConfig->Callback.pCallback = Callback;
....
pRsaDec = rsaDecCreate (pConfig); /* Create and initialize the instance */

....
}
```

3.3.7 *rsaDecInit*

Call(s):

```
Result rsaDecInit (RSA_sDecHandle *pRsaDec, RSA_sConfigure *pConfig);
```

Required Header: "rsa.h"

Arguments:

Table 3-7. *rsaDecInit* Arguments

pRsaDec	in	Handle to an instance of RSA decryptor generated by a call to *rsaDecCreate*
pConfig	in	A pointer to a data structure containing data for initializing the RSA algorithm

Description: The *rsaDecInit* function will initialize the RSA decryptor algorithm. During initialization, all resources will be set to their initial values in preparation for the RSA decryption operation. Before calling the *rsaDecInit* function, an RSA decryptor instance must be created. The RSA decryptor instance, *pRsadec*, can be created by either calling the *rsaDecCreate* function, shown in **Section 3.3.1**, or by statically allocating memory, which does not require calling the *rsaDecCreate* function.

The parameter *pConfig* points to a data structure of type `RSA_sConfigure`; its fields initialize RSA decryptor operation in the following manner:

RsaModNLen - length of modulo N buffer in bits

RsaN - pointer to a *modulo N buffer* (see rsa.h file for details)

RsaVLen - length of encryption buffer V in bits

RsaV - pointer to a buffer used for decryption

Callback - a structure of type *RSA_sCallback*; it describes the procedure which RSA will call once the data bytes are processed (encrypted) by the algorithm. The callback procedure has the following declaration:

```
void (*pCallback) (void *pCallbackArg, Word16 *pWords, UWord16
                   NumberWords);
```

The callback procedure parameter, *pCallbackArg*, is supplied by the user in the `RSA_sCallback` structure; this value is passed back to the user during the call to the Callback procedure. Typically, *pCallbackArg* points to context information used by the callback procedure, which the user must write.

pWords - pointer to input data words

NumberWords - Number of words in the input data buffer pointed to by *pWords*

Code Example 3-11. Sample Callback Procedure

```
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords)
{
    Int16 i;

    for (i = 0; i < NumberWords; i++)
    {
       Data_Out[i]  = pWords[i];
    }
    flag = 1;
    return;
}
```

Since this function must be written by the user, its content depends on how the user wants to process the RSA decryptor output.

Returns: Upon successful completion, a value of “TRUE” will be returned; if pConfig->RsaModNLen = 0, a value of “FALSE” will be returned .

Special Considerations:

- If *rsaDecCreate* is called, then the user need not call *rsaDecInit* function as it is called internally in the *rsaDecCreate* function.

In **Code Example 3-12**, the application creates an instance of RSA decryptor. The instance is passed to *rsaDecInit* along with the RSA configuration structure *pConfig*.

Code Example 3-12. Use of *rsaDecInit* Interface

```
#include “rsa.h”
#include “mem.h”

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 v  =  0x0003;

Frac16 message[] = {
                     0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                     0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                     0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                     0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                     0x5522,0x8833,0x9944,0xc455
                   };

/* output buffers */
Word16 Data_Out[NUMWORDS + 2];
Int16 flag; /* Flag to indicate data is available in o/p vector */
```

```
/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);

/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sDecHandle *pRsaDec;

void testRsa ()
{
   Result res;

   /* Allocate memory for RSA_sConfigure structure */
   pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
   if (pConfig == NULL)
      assert (!"Memory error");

   /* Initialise pConfig structure */
   pConfig->RsaModNLen = 0x0201;
   pConfig->RsaN = n;
   pConfig->RsaVLen = 2;
   pConfig->RsaV = &v;
   pConfig->Callback.pCallback = Callback;

   /* Create and init Decode handle */
   pRsaDec = rsaDecCreate (pConfig);
   if (pRsaEnc == NULL)
      assert (!" Decode create failed");

   .......

}
```

3.3.8 *rsaDecrypt*

Call(s):

```
Result rsaDecrypt (RSA_sDecHandle *pRsaDec, Word16 *pInWords, UWord16
                   NumberWords);
```

Required Header: "rsa.h"

Arguments:

Table 3-8. *rsaDecrypt* Arguments

pRsaDec	in	Handle to an instance of RSA decryptor generated by a call to *rsaDecCreate*
pInWords	in	Pointer to the user data words to be decrypted by the RSA decryptor algorithm
NumberWords	in	The number of data words to be decrypted

Description: The *rsaDecrypt* function will decrypt the data supplied. Once decryption is complete, the result is returned to the user by calling the *Callback* procedure. The user can call the *rsaDecrypt* function any number of times, as long as the user has data.

Returns: "PASS"

Special Considerations:

- Inplace computation is allowed; *i.e.*, input and output buffers could be the same
- The *rsaDecrypt* function makes a call to the *Callback* procedure only when max_message_len = {(pConfig->RsaModNLen+2) >> 4} of data are decrypted
- The **total length** of data passed for decryption should always be an integer multiple of max_message _len. The data to be passed for decryption is given below:

  Let max_message_len = 32

  ```
  rsaDecrypt (pRsaDec, pInWords, 64); /* for decryption */
  ```

  The function *rsaDecrypt* will make two calls to callback, giving 64 words of decrypted data. If the user does not pass data in multiples of max_message_len, then, after decrypting integer multiples of max_message_len, the remaining words are held. At the end of decryption, when a call is made to *rsaDecControl* with command parameter RSA_DEACTIVATE, the decryptor will **append zeros** and will generate 32 words of decrypted data, which may not be the expected data. A valid decryption is ensured only when the data to be decrypted is a multiple of max_message_len.
- The output buffer must be a multiple of max_message_len = {(pConfig->RsaModNLen+2) >> 4}

Code Example: In **Code Example 3-13**, during decryption, the **total** length of input data is (32 + 80 + 80) = 192 words, which is an integer multiple of max_message_len, meeting the constraint. The decryptor will make 5 calls to the callback function and will return 192 words of valid decrypted data.

Code Example 3-13. Use of *rsaDecrypt* Interface

```
#include “rsa.h”
#include “mem.h”

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 v  =  0x0003;

Frac16 message[] = {
                     0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                     0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                     0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                     0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                     0x5522,0x8833,0x9944,0xc455
                   };
Frac16 pInBuff1[80], pInBuff2[80];

/* output buffers */
Word16 Data_Out[NUMWORDS + 2];
Int16 flag; /* Flag to indicate data is available in o/p vector */

/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);

/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sDecHandle *pRsaDec;

void testRsa ()
{
   Result res;

   /* Allocate memory for RSA_sConfigure structure */
   pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
   if (pConfig == NULL)
      assert (!"Memory error");

   /* Initialise pConfig structure */
   pConfig->RsaModNLen = 0x0201;
   pConfig->RsaN = n;
   pConfig->RsaVLen = 2;
   pConfig->RsaV = &v;
   pConfig->Callback.pCallback = Callback;

   /* Create and init Decode handle */
   pRsaDec = rsaDecCreate (pConfig);
   if (pRsaEnc == NULL)
      assert (!" Decode create failed");

   res = rsaDecrypt (pRsaDec, message, 32);
```

```
    ...

    res = rsaDencrypt (pRsaDec, pInBuff1, 80);

    ...

    res = rsaDecrypt (pRsaDec, pInBuff2, 80);

    ...

    rsaDecDestroy (pRsaDec); /* rsaDecDestroy calls rsaDecControl */

    ...

}
```

3.3.9 *rsaDecControl*

Call(s):

```
Result rsaDecControl (RSA_sDecHandle *pRsaDec, UWord16 Command);
```

Required Header: "rsa.h"

Arguments:

Table 3-9. *rsaDecControl* Arguments

pRsaDec	in	Handle to an instance of RSA decryptor generated by a call to *rsaDecCreate*
Command	in	The command to be executed by the *rsaDecControl* procedure

Description: The *rsaDecControl* function provides control functions to the RSA decryption algorithm.

The parameter *pRsaDec* must have been generated from a call to *rsaDecCreate*. The parameter *Command* determines which action the *rsaDecControl* algorithm will perform; see **Table 3-10**.

Table 3-10. *Command* Parameter

RSA_DEACTIVATE	Deactivates the RSA Decryption operation During decryption, if RSA decryption must be terminated, this command flushes any data in its decryption buffer, even if it doesn't form a proper block required for RSA decryption, **by appending zeros**, decrypting, then calling the *Callback* procedure.

Returns: Upon successful completion, *rsaDecControl* will return "PASS"; otherwise, "FAIL" is returned.

Special Considerations: Calling the *rsaDecControl* function does not free the memory allocated during the *rsaDecCreate* function. To deallocate buffers, the *rsaDecDestroy* function must be called only if *rsaDecCreate* function was used to create the instance. If the user bypassed the *rsaDecCreate* function to create the instance, then the user must free the memory.

Code Example 3-14. Use of *rsaDecControl* Interface

```
#include "rsa.h"
#include "mem.h"

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 v  =  0x0003;

Frac16 message[] = {
                     0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                     0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                     0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                     0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                     0x5522,0x8833,0x9944,0xc455
                   };
```

```
/* output buffers */
Word16 Data_Out[NUMWORDS + 2];
Int16 flag; /* Flag to indicate data is available in o/p vector */

/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);

/* RSA instances */
RSA_sConfigure *pConfig;
RSA_sDecHandle *pRsaDec;

void testRsa ()
{
   Result res;
   UWord16 Command = RSA_DEACTIVATE;

   /* Allocate memory for RSA_sConfigure structure */
   pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
   if (pConfig == NULL)
      assert (!"Memory error");

   /* Initialise pConfig structure */
   pConfig->RsaModNLen = 0x0201;
   pConfig->RsaN = n;
   pConfig->RsaVLen = 2;
   pConfig->RsaV = &v;
   pConfig->Callback.pCallback = Callback;

   /* Create and init Decode handle */
   pRsaDec = rsaDecCreate (pConfig);
   if (pRsaEnc == NULL)
      assert (!" Decode create failed");

   res = rsaDecrypt (pRsaDec, message, 32);
   ...
   res = rsaDecControl (pRsaDec, Command);
   ...

}
```

3.3.10 *rsaDecDestroy*

Call(s):

```
void   rsaDecDestroy (RSA_sDecHandle *pRsaDec);
```

Required Header: "rsa.h"

Arguments:

Table 3-11. *rsaDecDestroy* Arguments

pRsaDec	in	Handle to an instance of RSA decryptor generated by a call to *rsaDecCreate*

Description: The *rsaDecDestroy* function destroys the instance of the RSA originally created by a call to *rsaDecCreate*. It also internally calls an encryption or decryption algorithm to complete the operation on any residual non-complete blocks by appending zeros.

Returns: None

Special Considerations: The *rsaDecDestroy* function calls *rsaDecControl* and frees the memory allocated during the *rsaDecCreate* function. Hence, the *rsaDecDestroy* function deactivates RSA and frees the memory allocated during the *rsaDecCreate* function.

Code Example 3-15. Use of *rsaDecDestroy* Interface

```
#include "rsa.h"
#include "mem.h"

Frac16 n[] = { 0x40b7,0x1acd,0x1b04,0xe832,0xa0b8,0x92f0,0x8ce6,
               0x12ec,0x0640,0x66fb,0x19fd,0x8ea1,0xdf35,0xdd7c,
               0x7920,0x2508,0x2b71,0xbae5,0xebbc,0x21c5,0x3c8a,
               0xbb30,0xd15d,0xbba2,0x0000,0x0000,0x0000,0x0000,
               0x0000,0x0000,0x0000,0x0000,0x0001
             };
Frac16 v  =  0x0003;

Frac16 message[] = {
                     0xee06,0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,
                     0xff77,0x0088,0xdd99,0xbbaa,0x66bb,0xe20c,0xee00,
                     0x3311,0x5522,0x8833,0x9944,0x4455,0x2266,0xff77,
                     0x0088,0xdd99,0xbbaa,0x66bb,0xe70c,0xee00,0x3311,
                     0x5522,0x8833,0x9944,0xc455
                   };

/* output buffers */
Word16 Data_Out[NUMWORDS + 2];
Int16 flag; /* Flag to indicate data is available in o/p vector */

/* Callback function prototype */
void Callback (void *pCallbackArg, Word16 *pWords, UWord16 NumberWords);

/* RSA instances */
```

```
RSA_sConfigure *pConfig;
RSA_sDecHandle *pRsaDec;

void testRsa ()
{
   Result res;

   /* Allocate memory for RSA_sConfigure structure */
   pConfig = (RSA_sConfigure *) memMallocEM (sizeof (RSA_sConfigure));
   if (pConfig == NULL)
      assert (!"Memory error");

   /* Initialise pConfig structure */
   pConfig->RsaModNLen = 0x0201;
   pConfig->RsaN = n;
   pConfig->RsaVLen = 2;
   pConfig->RsaV = &v;
   pConfig->Callback.pCallback = Callback;

   /* Create and init Decode handle */
   pRsaDec = rsaDecCreate (pConfig);
   if (pRsaEnc == NULL)
      assert (!" Decode create failed");

   res = rsaDecrypt (pRsaDec, message, 32);

   rsaDecDestroy (pRsaDec);

   ....
}
```

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Chapter 4
Building the RSA Library

4.1 Building the RSA Library

The RSA library combines all of the components described in previous sections into one library: *RSA.lib*. To build this library, a Metrowerks CodeWarrior project, ***rsa.mcp***, is provided. This project and all the necessary components to build the RSA library are located in the ***...\nos\security\rsa*** directory of the SDK directory structure.

There are two methods to execute system library project build: Dependency Build and Direct Build.

4.1.1 Dependency Build

Dependency build is the easiest approach and doesn't require any additional work on the user's part. If you add the RSA library project to your application project, as shown in **Figure 4-1**, the RSA library will automatically build when the application is built.

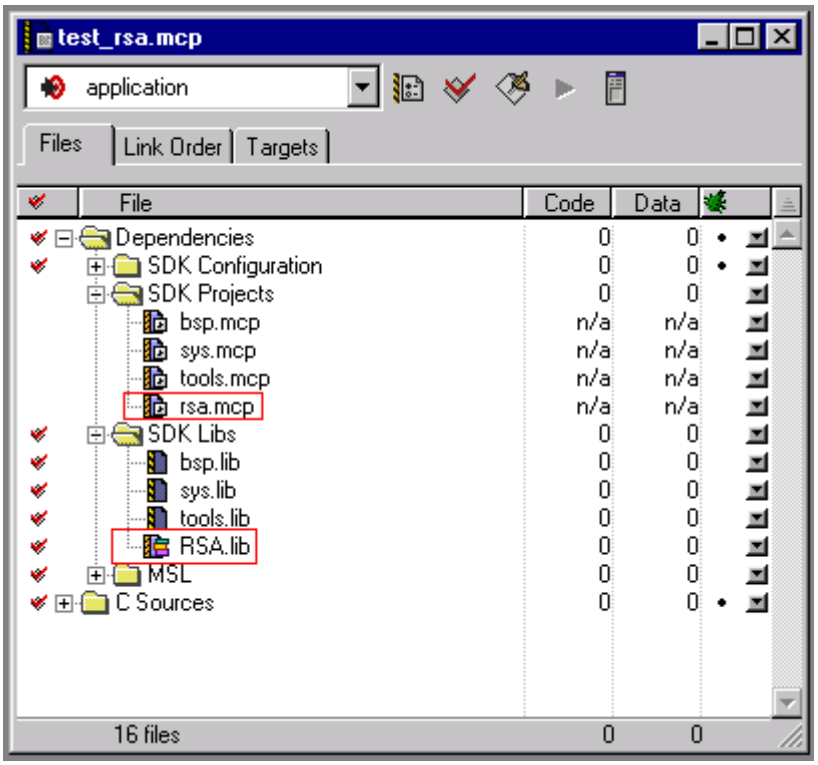

Figure 4-1. Dependency Build for RSA Library

4.1.2 Direct Build

Direct build allows you to build a tools library independently of any other build. To do this:

Step 1. Open the ***rsa.mcp*** project, as shown in **Figure 4-2**.

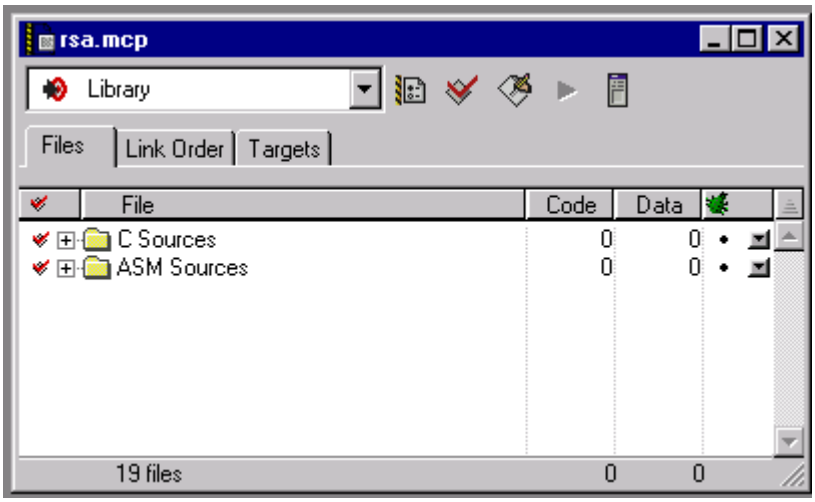

Figure 4-2. *rsa.mcp* Project

Step 2. Execute the build by pressing function key [F7] or by choosing *Make* from the Project menu; see **Figure 4-3**.

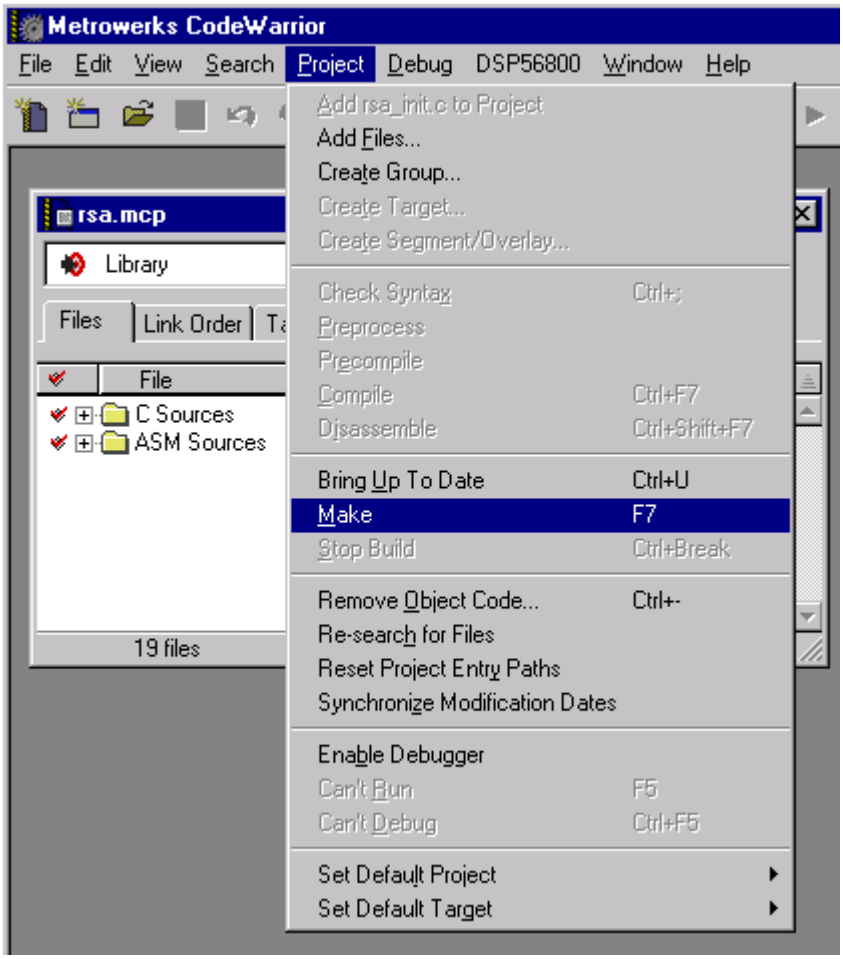

Figure 4-3. Execute *Make*

At this point, if the build is successful, the *RSA.lib* file is created in the ***...\nos\security\rsa\Debug*** directory.

Chapter 5
Linking Applications with the RSA Library

5.1 RSA Library

The RSA library consists of RSA code and APIs (which provide the interface between the user application and the RSA modules). To invoke RSA (encryption/decryption), the following APIs must be called in the following order:

- rsaEncCreate (.....);
- rsaEncInit (.....);
- rsaEncrypt (.....);.
- rsaEncControl (.....); or rsaEncDestroy (.....);
- rsaDecCreate (.....);
- rsaDecInit (.....);
- rsaDecrypt (.....);
- rsaDecControl (.....); or rsaDecDestroy (.....);

5.1.1 Library Sections

The linker command file used in the test application is called *linker.cmd*, and is shown in **Code Example 5-1**.

Code Example 5-1. *linker.cmd* File

```
# Linker.cmd file for DSP56824EVM External RAM
# using both internal and external data memory (EX = 0)
# and using external program memory (Mode = 3)

MEMORY {
              .pram   (RWX) : ORIGIN = 0x0000, LENGTH = 0xFF80  # ? external program
                                  memory
              .avail  (RW)  : ORIGIN = 0x0000, LENGTH = 0x0030  # available
              .cwregs (RW)  : ORIGIN = 0x0030, LENGTH = 0x0010  # C temp registrs in
                                  CodeWarrior
              .im1    (RW)  : ORIGIN = 0x0040, LENGTH = 0x07C0  # data 1
              .rom    (R)   : ORIGIN = 0x0800, LENGTH = 0x0800  # internal data ROM
              .im2    (RW)  : ORIGIN = 0x1000, LENGTH = 0x0600  # data 2
              .hole   (R)   : ORIGIN = 0x1600, LENGTH = 0x0A00  # hole
              .data   (RW)  : ORIGIN = 0x2000, LENGTH = 0xC000  # data segment
              .em     (RW)  : ORIGIN = 0xE000, LENGTH = 0x1000  # data 3
              .stack  (RW)  : ORIGIN = 0xF000, LENGTH = 0x0F80  # stack
              .onchip1(RW)  : ORIGIN = 0xFF80, LENGTH = 0x0040  # on-chip peripheral
                                  registers
              .onchip2(RW)  : ORIGIN = 0xFFC0, LENGTH = 0x0040  # on-chip peripheral
                                  registers

}

FORCE_ACTIVE {FconfigInterruptVector}

SECTIONS {
             #
             # Data (X) Memory Layout
             #
                    _EX_BIT     = 0;

                    # Internal Memory Partitions (for mem.h partitions)
                    _NUM_IM_PARTITIONS = 2;  # .im1 and .im2

                    # External Memory Partition (for mem.h partitions)
                    _NUM_EM_PARTITIONS = 1;   # .em

             .main_application_code :
             {
                    # .text sections

                    #  config.c MUST be placed first, otherwise the Interrupt Vector
                    #  configInterruptVector will not be located at the correct
                    address, P:0x0000

                    config.c (.text)
                    * (.text)
                    * (rtlib.text)
                    * (fp_engine.text)
                    * (user.text)
             } > .pram
```

```
.main_application_data :
{
      #
      # Define variables for C initialization code
      #
      F_Xdata_start_addr_in_ROM = ADDR(.rom) + SIZEOF(.rom) / 2;
      F_StackAddr               = ADDR(.stack);
      F_StackEndAddr            = ADDR(.stack) + SIZEOF(.stack) / 2  - 1;

      F_Xdata_start_addr_in_RAM = .;

      #
      # Memory layout data for SDK INCLUDE_MEMORY (mem.h) support
      #

      FmemEXbit = .;
            WRITEH(_EX_BIT);
      FmemNumIMpartitions = .;
            WRITEH(_NUM_IM_PARTITIONS);
      FmemNumEMpartitions = .;
            WRITEH(_NUM_EM_PARTITIONS);
      FmemIMpartitionList = .;
            WRITEH(ADDR(.im1));
            WRITEH(SIZEOF(.im1) / 2);
            WRITEH(ADDR(.im2));
            WRITEH(SIZEOF(.im2) / 2);
      FmemEMpartitionList = .;
            WRITEH(ADDR(.em));
            WRITEH(SIZEOF(.em) /2);

      # .data sections
      * (.data)
      * (fp_state.data)
      * (rtlib.data)

      F_Xdata_ROMtoRAM_length = 0;

      F_bss_start_addr = .;
      _BSS_ADDR = .;

    * (rtlib.bss.lo)
     * (.bss)

      F_bss_length = . - _BSS_ADDR;  # Copy DATA

} > .data

FArchIO   = ADDR(.onchip2);

}
```

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Chapter 6
RSA Applications

6.1 Test and Demo Applications

To verify the RSA algorithm, test and demo applications have been developed. Refer to the **Targeting Motorola DSP568xx Platform** Manual for the DSP you are using to see if the test and demo applications are available for your target.

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Chapter 7
License

7.1 Limited Use License Agreement

LIMITED USE LICENSE AGREEMENT

PLEASE READ THIS AGREEMENT CAREFULLY BEFORE USING THIS SOFTWARE. BY USING OR COPYING THE SOFTWARE, YOU AGREE TO THE TERMS OF THIS AGREEMENT.

The software in either source code form ("Source") or object code form ("Object") (cumulatively hereinafter "Software") is provided under a license agreement ("Agreement") as described herein. Any use of the Software including copying, modifying, or installing the Software so that it is usable by or accessible by a central processing unit constitutes acceptance of the terms of the Agreement by the person or persons making such use or, if employed, the employer thereof ("Licensee") and if employed, the person(s) making such use hereby warrants that they have the authority of their employer to enter this license agreement,. If Licensee does not agree with and accept the terms of this Agreement, Licensee must return or destroy any media containing the Software or materials related thereto, and destroy all copies of the Software.

The Software is licensed to Licensee by Motorola Incorporated ("Motorola") for use under the terms of this Agreement. Motorola retains ownership of the Software. Motorola grants only the rights specifically granted in this Agreement and grants no other rights. Title to the Software, all copies thereof and all rights therein, including all rights in any intellectual property including patents, copyrights, and trade secrets applicable thereto, shall remain vested in Motorola.

For the Source, Motorola grants Licensee a personal, non-exclusive, non-assignable, revocable, royalty-free right to use, copy, and make derivatives of the Source solely in a development system environment in order to produce object code solely for operating on a Motorola semiconductor device having a central processing unit ("Derivative Object").

For the Object and Derivative Object, Motorola grants Licensee a personal, non-exclusive, non-assignable, revocable, royalty-free right to copy, use, and distribute the Object and the Derivative Object solely for operating on a Motorola semiconductor device having a central processing unit.

Licensee agrees to: (a) not use, modify, or copy the Software except as expressly provided herein, (b) not distribute, disclose, transfer, sell, assign, rent, lease, or otherwise make available the Software, any derivatives thereof, or this license to a third party except as expressly provided herein, (c) not remove obliterate, or otherwise defeat any copyright, trademark, patent or proprietary notices, related to the

Software (d) not in any form export, re-export, resell, ship or divert or cause to be exported, re-exported, resold, shipped, or diverted, directly or indirectly, the Software or a direct product thereof to any country which the United States government or any agency thereof at the time of export or re-export requires an export license or other government approval without first obtaining such license or approval.

THE SOFTWARE IS PROVIDED ON AN "AS IS" BASIS AND WITHOUT WARRANTY OF ANY KIND INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL MOTOROLA BE LIABLE FOR ANY LIABILITY OR DAMAGES OF ANY KIND INCLUDING, WITHOUT LIMITATION, DIRECT OR INDIRECT OR INCIDENTAL OR CONSEQUENTIAL OR PUNITIVE DAMAGES OR LOST PROFITS OR LOSS OF USE ARISING FROM USE OF THE SOFTWARE OR THE PRODUCT REGARDLESS OF THE FORM OF ACTION OR THEORY OF LIABILITY (INCLUDING WITHOUT LIMITATION, ACTION IN CONTRACT, NEGLIGENCE, OR PRODUCT LIABILITY) EVEN IF MOTOROLA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. THIS DISCLAIMER OF WARRANTY EXTENDS TO LICENSEE OR USERS OF PRODUCTS AND IS IN LIEU OF ALL WARRANTIES WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR PARTICULAR PURPOSE.

Motorola does not represent or warrant that the Software is free of infringement of any third party patents, copyrights, trade secrets, or other intellectual property rights or that Motorola has the right to grant the licenses contained herein. Motorola does not represent or warrant that the Software is free of defect, or that it meets any particular requirements or need of the Licensee, or that it conforms to any documentation, or that it meets any standards.

Motorola shall not be responsible to maintain the Software, provide upgrades to the Software, or provide any field service of the Software. Motorola reserves the right to make changes to the Software without further notice to Licensee.

The Software is not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Software could create a situation where personal injury or death may occur. Should Licensee purchase or use the Software for any such unintended or unauthorized application, Licensee shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the Software.

The term of this Agreement is for as long as Licensee uses the Software for its intended purpose and is not in default of any provisions of this Agreement. Motorola may terminate this Agreement if Licensee is in default of any of the terms and conditions of this Agreement.

This Agreement shall be governed by and construed in accordance with the laws of the State of Arizona and can only be modified in a writing signed by both parties. Licensee agrees to jurisdiction and venue in the State of Arizona.

By using, modifying, installing, compiling, or copying the Software, Licensee acknowledges that this Agreement has been read and understood and agrees to be bound by its terms and conditions. Licensee agrees that this Agreement is the complete and exclusive statement of the agreement between Licensee and Motorola and supersedes any earlier proposal or prior arrangement, whether oral or written, and any other communications relative to the subject matter of this Agreement.

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

Index

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1–303–675–2140 or 1–800–441–2447

JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3–20–1, Minami–Azabu. Minato–ku, Tokyo 106–8573 Japan. 81–3–3440–3569

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852–26668334

Technical Information Center: 1–800–521–6274

HOME PAGE: http://www.motorola.com/semiconductors/