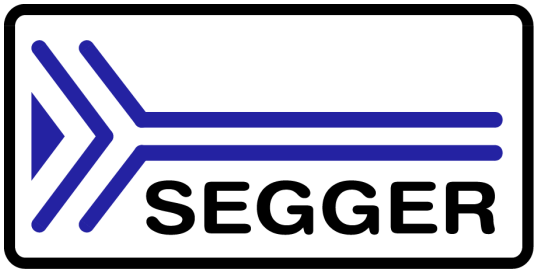
emSecure

Securing products with digital signatures

User & Reference Guide

Document: UM12002
Software version: 1.02
Revision: 0
Date: May 22, 2015



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH & Co. KG (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel.+49 2103-2878-0
Fax.+49 2103-2878-28
E-mail: support@segger.com
Internet: http://www.segger.com

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: May 22, 2015

Software	Revision	Date	By	Description
1.02	0	150522	JL	Chapter "Introduction" * "Package content" updated. Chapter "Working with emSecure" * Utility description updated. Chapter "emSecure API" * Section "Configuring emSecure" updated. * Compile-time configuration added.
1.00	1	141016	JL	Chapter "Support" added. Chapter "Appendix" * Flowchart "emSecuring a product" added.
1.00	0	140827	JL	Initial Release

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler)
- The C programming language
- The target processor
- DOS command line

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table 2.1: Typographic conventions

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device/host stack
A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace
USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Introduction to emSecure

1.1 What is emSecure?

emSecure is a SEGGER software package that allows creation and verification of digital signatures.

One important feature is that it can make it impossible to create a clone of an embedded device by simply copying hardware and firmware.

And It can do much more, such as securing firmware updates for any kind of embedded device, licenses, serial numbers or other sensitive data. A must for critical devices such as election machines, financial applications and many others.

Based on RSA asymmetric encryption with 2 keys, it can not be broken by reverse engineering.

The source code has been created from scratch for embedded systems, to achieve highest portability with a small memory footprint and high performance. However, usage is not restricted to embedded systems.

It is licensed in the same way as other SEGGER middleware products and does not rely on any foreign code or code licensed under an open-source or even "viral" GPL-style license.

With its easy usage, it takes less than half a day to add and integrate emSecure into an existing product.

emSecure is a very complete package, including utilities for generation and conversion of keys.

1.1.1 Why is it better than CRCs or hash functions?

CRCs and hash functions are in general a good way to ensure that a data transmission such as a firmware download has worked flawlessly. They can also be used to make sure that an image has not changed when stored in flash memory. However, they do not add much security, as an attacker can easily compute the CRC or hash value of a modified firmware.

The important difference to digital signatures is that the digital signature is based on an asymmetric encryption algorithm:

The digital signature cannot be forged by analyzing the verification code in the firmware. The digital signature can only be generated with the private key, which is not contained in the firmware. In addition to the integrity check, which is also provided with CRCs and hash functions, a digital signature assures the authenticity of the provider of the signed data, as only he can create a valid signature.

1.2 Features

emSecure is written in ANSI-C and does neither use any OS- or hardware-specific code nor requires additional hardware modules. It can be used on virtually any CPU, PCs, tablets, smartphones, microcontrollers, ...

emSecure can easily be integrated into new and existing products. It provides a simple yet powerful API and requires less than half a day of work for integration.

All used modules are conform to their relevant FIPS specifications. Validation code with defined test vectors are included.

emSecure comes with all needed utilities to secure a product. Pre-compiled and ready to use. Additional example applications and the source code are included.

1.3 Package content

emSecure is provided in Source Code and contains all needed modules.

The following table shows the content of the emSecure Package:

Files	Contents
\Config	emSecure user configurable header file.
\CRYPTO	Shared cryptographic library source code.
\Doc	emSecure documentation.
\Sample\Config	Example emSecure configuration.
\Sample\Keys	Example emSecure key pair.
\SECURE	emSecure implementation source code.
\SEGGER	Shared SEGGER software components.
\SYS	System-independent source files for Windows and embOS.
\Windows\SECURE	emSecure sample applications.

Table 1.1: emSecure package content

Chapter 2

Working with emSecure

This chapter gives some recommendations on how to use emSecure in your applications. It explains the steps of “emSecuring” a product.

2.1 How does emSecure work?

emSecure is working with digital signatures.

Digital signatures provide the authenticity of an author in addition to the integrity of his message.

The author uses his private and secret key to digitally sign a message. He provides his public key, with which the digital signature can be verified.

2.2 emSecuring a product

Securing a product with emSecure is kept as simple as possible. There are basically three steps to make a product secure.

Key generation: The first step is done once for a product which shall be secured.
Data signing: The second step is done once for each unit of the product.
Verification: The third step is usually done on every use of the product.

All steps can be integrated in their corresponding field of application.

2.2.1 Key generation

To secure a product, a pair of keys has to be generated. This is a one-time setup, one key pair can be used for one or even more products.

The keys can be created on any computer. To be safe and secured, a stand-alone computer without network access can be used to generate the keys. The key files can be exchanged and published and converted in other formats to be compiled and used directly in an application.

Key generation is done with the included utility emKeyGen.

The key size can be selected to fit both, size and security requirements. A key length of at least 1024 bits (Commercial Grade) or 2048 bits (Military Grade) is recommended.

2.2.2 Data signing

To secure a product, some piece of data, the message, has to be selected and signed. This can be:

- a programmed serial number,
- a device specific product ID (UID),
- the whole, or a piece of the application,
- ...

or a combination of some of them.

The message is signed with the private key, creating a digital signature with the length of the key size. The signature is then stored in the product.

Data signing should be done in the production of the product and can be done from any production computer. Sample code, how to sign a message is included in the utility emSign and shown in the example codes. See “Signing a message” on page 32.

2.2.3 In-product verification

The digital signature can be verified in the product. This can for example be done to:

- Validate a programmed serial number.
- Verify the firmware is allowed to run on a device with a specific ID.
- Check if the firmware image is valid and correct.

For in-product verification the signature and the public key can be programmed into the product in the production.

How to verify a signature is shown in the example codes. See “Verifying a message” on page 33.

2.2.4 Off-product verification

The digital signature can also be verified in an application using the product, for example a PC application communicating with the product via USB or a production and maintenance tool connected to the product. Off-product verification can be used for the same purposes as in-product verification and additionally to for example:

- Verify the connected product is genuine.

- Check if the product is licensed to be used with the application.

How to verify a signature is shown in the example codes. See “Verifying a message” on page 33.

2.3 Implementing emSecure

The implementation of emSecure in a product and in its production environment is kept as simple as the usage of emSecure.

There are basically three steps to implement emSecure. Example code for all needed implementations is included.

2.3.1 Data signing

Add the code to sign the data to be secured (e.g. a serial number, product ID or the firmware) with the private key to the production application.

The private key is generated with emKeyGen and can be loaded by the production application from the key file or is exported from the keyfile into compilable code and part of the production application.

The data is read from the product or generated from the production utility and will later on be stored in the product.

The signature can be stored in the product or saved as a signature file.

2.3.2 Signature and Key exchange

The public key and the signature have to be added to the product or the verifying application.

emSecure provides simple to use utilities to convert keys and signatures between different human readable and compilable code representations.

2.3.3 Verification

Add the code to verify the signature to the product or verifying application.

Verifying a signature is basically one simple API call with the message, the public key and the signature as parameters and can be used for in-product and off-product verification.

Example code on how to verify a signature as well as the sample utility "emVerify" which verifies a file by its signature file are included.

2.4 Included applications

emSecure includes all basic applications needed for securing a product. Additional applications for benchmark and validations are part of emSecure, too. The applications' source-code is included and provides an easy to use starting point for modifications and integration into other applications.

Application	Target platform	Description
emKeyGen	Windows	Generate a key pair with a given key length, either random or from a given seed.
emSign	Windows	Digitally sign a file with your private key.
emVerify	Windows	Verify the signature of a digital signature with its public key.
emPrintKey	Windows	Export keys and signatures into C source format, to be included into any application.
emBenchmark	Windows	Test the speed of operations with different key lengths.
emValidate	Windows	Validate the modules of emSecure.

Table 2.1: Included applications

2.4.1 Utilities

The utilities are PC applications, ready-to-use to secure your product.

2.4.1.1 Key generation

emKeyGen generates a public and a private key. The generation parameters can be set with command line options. The keys are saved in a common key file format and can be published and exchanged.

Usage:

emKeyGen.exe [<Options>]

```
C:\emSecure>emKeyGen.exe -rsa -l 2048 -f -k MyemSecureKeys

(c) 2014 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com
emSecure KeyGen V1.00 compiled Aug 26 2014 17:57:54

Selecting a random initial seed
Generating proven prime key pair with public modulus of 2048 bits
Public encryption exponent is set to 65537
Initial seed is 0x5F94E229568014D4D288B18DDF729E15
Checking keys are consistent: OK
Writing public key file MyemSecureKeys.pub.
Writing private key file MyemSecureKeys.prv.
```

Options:

```
-h        Print usage
-q        Operate silently
-v        Increase verbosity
-rsa      Generate RSA key pair
-k str    Set key file prefix to str   [default is 'emSecure']
-l n      Set modulus length to n bits [default is 1024]
-f        Generate keys using FIPS 186 proven prime algorithm
-seed n   Set FIPS 186 initial seed to n; forces -f
-pw str   Set FIPS 186 initial seed from PBKDF2 password str; forces -f
```


2.4.1.2 Exporting keys and signatures

emPrintKey exports key and signature files into a format suitable for compilation by a standard C compiler. The output can be linked into your application, so there is no need to load them from a file at runtime. This is especially useful for embedded applications.

Usage:

emPrintKey.exe [<Options>] <Input-File>

```
C:\emSecure>emPrintKey.exe -m -p MyPublicKey_ MyemSecureKeys.pub > MyKeys.c

C:\emSecure>emPrintKey.exe -m -p MyPrivateKey_ MyemSecureKeys.prv >> MyKeys.c

C:\emSecure>emPrintKey.exe -h

Usage: emPrintKey [options] <input-file>

Options:
  -v    Verbose
  -x    Define objects with external linkage
  -p    Prefix object names with <str> [default is '_']
  -m    Define objects in preloaded internal MPI form
```

Options:

```
  -v    Verbose
  -x    Define objects with external linkage
  -p    Prefix object names with <str> [default is '_']
  -m    Define objects in preloaded internal MPI form
  -le   Define objects in raw form, little endian (PC byte order)
  -be   Define objects in raw form, big endian (network byte order)   [default]
```

2.4.2 Example applications

The example applications can be used "as is", but are mainly included as a reference to include emSecure in your product. Source code for the example applications is part of emSecure.

2.4.2.1 Signing data

emSign digitally signs the file content, usually the data to be secured, with a given (private) key file and creates a signature file.

Usage:

emSign.exe [<Options>] <InputFile>

```
C:\emSecure>emSign.exe -k MyemSecureKeys.prv SecureData.bin

(c) 2014 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com
emSecure Sign V1.00 compiled Aug 26 2014 17:57:55

Loading private key from MyemSecureKeys.prv
Probing file to determine type of key: RSA key detected
Public modulus is 2048 bits
Loading content from SecureData.bin
Loaded content is 524275 bytes
Writing signature file SecureData.bin.sig
```

2.4.2.2 Verifying data

emVerify decrypts a signature file and verifies if the corresponding data file matches the signature.

Usage:

emVerify.exe [<Options>] <InputFile>

```
C:\emSecure>emVerify.exe -k MyemSecureKeys.pub SecureData.bin

(c) 2014 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com
emSecure Verify V1.00 compiled Aug 26 2014 17:57:55

Loading public key from MyemSecureKeys.pub
Probing file to determine type of key: RSA key detected
Public modulus is 2048 bits
Loading RSA signature from SecureData.bin.sig
Loading content from SecureData.bin
Loaded content is 524275 bytes
Signature OK.
```

2.4.3 Performance analysis applications

The performance analysis applications can be used to test the implementation and speed of emSecure. They can easily be modified to be executed on other platforms, like embedded targets.

2.4.3.1 Speed tests

emBenchmark tests the speed performance of the most common used functions like signing and verifying and prints the results. The benchmark is pure C code without target-specific assembly language optimisation and shows that emSecure is fast enough for boot-time signature validation on embedded devices.

2.4.3.2 Validation tests

emValidate tests the implementation of the algorithms and modules in emSecure with defined test vectors and parameters.

Chapter 3

emSecure API

This chapter explains the API functions of emSecure which are needed so secure a product. The emSecure API is kept as simple as possible to provide a fast and easy to integrate way of securing a product.

3.1 emSecure API functions

The table below lists the routines of the emSecure API. Detailed description of the routines can be found in the section that follows.

Routine	Explanation
SECURE_InitPrivateKey()	Initialize a private key and set its parameters.
SECURE_InitPublicKey()	Initialize a public key and set its parameters.
SECURE_Sign()	Create a signature for a message.
SECURE_Verify()	Verify a message by its signature.

Table 3.1: emSecure API functions

3.1.1 SECURE_InitPrivateKey()

Description

Initialize a private key and set its parameters.

Use this function to set the private key parameters for uninitialized private keys created in RAM.

Prototype

```
void SECURE_InitPrivateKey (      SECURE_PRIVATE_KEY*   pPrivate,
                            const SECURE_KEY_PARAMETER* pParamDP,
                            const SECURE_KEY_PARAMETER* pParamDQ,
                            const SECURE_KEY_PARAMETER* pParamP,
                            const SECURE_KEY_PARAMETER* pParamQ,
                            const SECURE_KEY_PARAMETER* pParamQInv);
```

Parameter	Meaning
pPrivate	Pointer to the private key to hold the parameters.
pParamDP, pParamDQ, pParamP, pParamQ, pParamQInv	Pointers to private key parameters in SECURE_KEY_PARAMETER form.

Table 3.2: Secure_InitPrivateKey parameter list

Additional information

Private key header files created with emPrintKey include the initialized private key. SECURE_InitPrivateKey() does not have to be called.

SECURE_InitPrivateKey() is needed when the key parameters are stored separately, for example when they are stored encrypted.

Example

```
#include "SecureKeys.h"  // Header including the private key parameters

void Func (void) {
  SECURE_PRIVATE_KEY PrivateKey;

  SECURE_InitPrivateKey(&PrivateKey,                                    \
                        &SecureKey_DP, &SecureKey_DQ,                   \
                        &SecureKey_P, &SecureKey_Q, &SecureKey_QINV);
}
```


3.1.2 SECURE_InitPublicKey()

Description

Initialize a public key and set its parameters.

Use this function to set the public key parameters for uninitialized public keys created in RAM.

Prototype

```
void SECURE_InitPublicKey (      SECURE_PUBLIC_KEY* pPublic,
                           const SECURE_KEY_PARAMETER* pParamE,
                           const SECURE_KEY_PARAMETER* pParamN);
```

Parameter	Meaning
pPublic	Pointer to the public key to hold the parameters.
pParamE, pParamN	Pointers to the public key parameters in SECURE_KEY_PARAMETER form.

Table 3.3: SECURE_InitPublicKey parameter list

Additional information

Public key header files created with emPrintKey include the initialized public key. SECURE_InitPublicKey() does not have to be called.

SECURE_InitPublicKey() is needed when the key parameters are stored separately, for example when they are stored encrypted.

Example

```
#include "SecureKeys.h"  // Header including the public key parameters

void Func (void) {
  SECURE_PUBLIC_KEY PublicKey;

  SECURE_InitPrivateKey(&PublicKey, &SecureKey_E, &SecureKey_N);
}
```


3.1.3 SECURE_Sign()

Description

Sign data using a private key.

Creates a digital signature from the hash of the message and the salt encrypted with the private key.

Prototype

```
int SECURE_Sign (const SECURE_PRIVATE_KEY * pPrivate,
                 const U8 * pSalt,      int NumBytesSalt,
                 const U8 * pData,      int NumBytesData,
                       U8 * pSignature, int NumBytesSignature);
```

Parameter	Meaning
pPrivate	Pointer to the private key to sign with. Parameters have to be initialized.
pSalt	Pointer to buffer containing salt bytes for the signature. May be NULL.
NumBytesSalt	Number of bytes to use as salt. May be 0.
pData	Pointer to data byte buffer to sign.
NumBytesData	Number of bytes in the data buffer to sign.
pSignature	Pointer to buffer to hold the signature.
NumBytesSignature	Number of bytes available in the signature buffer.

Table 3.4: SECURE_Sign parameter list

Return value

> 0	Success. Number of bytes in the signature.
< 0	Error.

Add. information

The length of the signature is equivalent to the length of the key modulus. The signature buffer has to be at least the size of the key length.

The salt is used to pad and modify the data and can be random data to create different signatures for same data. Only the size of the salt, but not the salt data has to be known for verification.

Example

```
static const U8 acData[] = {
                            // ... Data here ...
                           };
static const U8 acSalt[] = "SEGGER - The embedded Experts";
static       U8 acSig[512];

int _Sign(void) {
  int r;

  r = SECURE_Sign(&GLOBAL_PrivateKey,                                    \
                  acSalt, 30,                                            \
                  acData, sizeof(acData),                                \
                  acSig,  sizeof(acSig));

  return r;
}
```


3.1.4 SECURE_Verify()

Description

Verify a message by its signature.

Checks if the signature is valid and if the message hash matches the hash in the signature decrypted with the public key.

Prototype

```
int SECURE_Verify (const SECURE_PUBLIC_KEY * pPublic,
                         U8 * pSalt,      int NumBytesSalt,
                   const U8 * pData,      int NumBytesData,
                   const U8 * pSignature, int NumBytesSignature);
```

Parameter	Meaning
pPublic	Pointer to the public key for verification. Parameters have to be initialized.
pSalt	Pointer to buffer to store the recovered salt. May be NULL.
NumBytesSalt	Number of bytes used as salt to sign the data.
pData	Pointer to data byte buffer to verify.
NumBytesData	Number of bytes on the data buffer to verify.
pSignature	Pointer to signature byte buffer to verify.
NumBytesSignature	Number of bytes in the signature buffer.

Table 3.5: SECURE_Verify parameter list

Return value

!= 0	Success. Signature and data match. pSalt contains the recovered salt.
0	Error. Signature and data could not be verified. Content in pSalt is undefined.

Add. information

An error can occur under different circumstances: The public key does not match the private key used for signature generation, the signature is corrupted and does not match the structure of a digital signatrue, the data does not match the signature, the length of the salt is not correct.

To recover the salt used for signature generation pSalt can point to a buffer of at least NumBytesSalt length.

Example

```
static const U8 acData[] = {
                            // ... Data here ...
                           };
static const U8 acSig[]  = {
                            // ... Signature here ...
                           };
static       U8 acSalt[256];

int _Verify(void) {
  int r;

  r = SECURE_Verify (&GLOBAL_PublicKey,                                    \
                     acSalt, 30,                                           \
                     acData, sizeof(acData),                               \
                     acSig, sizeof(acSig));

  return r;
}
```


3.2 Configuring emSecure

emSecure is configurable. It is created to fit any requirement of high performance or low memory usage.

emSecure can be configured via preprocessor flags at compile-time. All compile-time configuration flgas are preconfigured with valid values, which match the requirements of most applications.

In order to configure the shared memory component and to select the way that SHA-1, SHA-256- SHA-512, and MD5 are compiled to balance code size and execution performance you can change the compile-time flags in the main configuration files CRYPTO_Conf.h and SEGGER_MEM_Conf.h.

The cryptography modules (prefixed CRYPTO_), and SEGGER modules (prefixed SEGGER_) are shared with other SEGGER products, for instance emSSL, and should be configured to match all modules which are used in the same application.

3.2.1 Configuring keys

3.2.1.1 SECURE_MAX_KEY_LENGTH

Configure the maximum length of keys. This define is used to reserve enough memory when signing or verifying a message. It has to be at least the size of the modulus length used in CRYPTO_MPI_LIMB_ALLOCATION_UNIT.

The default is 2048.

Configuration file: SECURE_Conf.h

3.2.2 Configuring multiprecision integers

The following definitions must be set for the multiprecision integer (MPI) cryptographic component to work correctly with the best performance. The MPI component is required for RSA algorithms.

3.2.2.1 CRYPTO_MPI_BITS_PER_LIMB

Configure number of bits per limb for multiprecision integer algorithms. The default is 32 which matches 32-bit targets well, such as ARM and PIC32. In general, it is best to set the number of bits per limb to the number of bits in the standard int or unsigned type used by the target compiler.

Supported configurations are:

- **32**, which requires the target compiler to support 64-bit types natively (i.e. unsigned long long or unsigned __int64),
- **16**, which should run on any ISO compiler whose native integer types are 16 or 32 bit and supports 32-bit unsigned long .
- **8**, 8-bit limb sizes are supported and selecting this size may well lead to better multiplication performance on 8-bit architectures.

Configuration file: CRYPTO_Conf.h

3.2.2.2 CRYPTO_MPI_LIMB_ALLOCATION_UNIT

Configure number of limbs to allocate as a unit to avoid continual reallocation of MPI limb data. The default is `(1024/CRYPTO_MPI_BITS_PER_LIMB+2)` .

This is a trade-off between minimal RAM use and potentially wasted space. For cryptographic computations and fixed key sizes, it's best to set this to a small multiple (1 or 2) of the most common modulus size that you will deal with plus a two limb overhead, i.e.


```
<common_modulus_lenth>/CRYPTO_MPI_BITS_PER_LIMB + 2
```

So, if you were most commonly dealing with 1024-bit moduli and selected 16-bit limbs, but wanted to deal with any other size also, you would set this to:

```
(1024/CRYPTO_MPI_BITS_PER_LIMB + 2)
```

which evaluates to 1024/16 + 2 = 66.

Configuration file: CRYPTO_Conf.h

3.2.3 Configuring Hashes

The following definitions must be set for the Hash components in order to balance code size and performance.

3.2.3.1 CRYPTO_MD5_OPTIMIZE_SIZE

Set this preprocessor symbol nonzero to optimize the MD5 hash functions for size rather than for speed. When optimized for speed, the MD5 function is open coded and faster, but is significantly larger.

The default is to compile for small size.

Configuration file: CRYPTO_Conf.h

3.2.3.2 CRYPTO_SHA1_OPTIMIZE_SIZE

Set this preprocessor symbol nonzero to optimize the SHA-1 hash functions for size rather than for speed. When optimized for speed, the SHA-1 function is open coded and faster, but is significantly larger.

The default is to compile for small size.

Configuration file: CRYPTO_Conf.h

3.2.3.3 CRYPTO_SHA256_OPTIMIZE_SIZE

Set this preprocessor symbol nonzero to optimize the SHA-256 hash functions for size rather than for speed. When optimized for speed, the SHA-256 function is open coded and faster, but is significantly larger.

The default is to compile for small size.

Configuration file: CRYPTO_Conf.h

3.2.3.4 CRYPTO_SHA512_OPTIMIZE_SIZE

Set this preprocessor symbol nonzero to optimize the SHA-512 hash functions for size rather than for speed. When optimized for speed, the SHA-512 function is open coded and faster, but is significantly larger.

The default is to compile for small size.

Configuration file: CRYPTO_Conf.h

3.2.4 Configuring checks

The following definitions must be set for the whole of the library.

3.2.4.1 CRYPTO_API_CHECKS

Set this preprocessor symbol nonzero to enable checks of parameters and computations in the CRYPTO API.

The default is for the API check symbol to follow the value of the DEBUG preprocessor symbol, to enable API checks for debug builds and to disable them for release builds.

Configuration file: CRYPTO_Conf.h

Chapter 4

Example Code

This chapter gives some recommendations on how to use emSecure in your applications. It explains “emSecuring” a product with some example code.

4.1 Example codes

The example codes show how to use emSecure for signing and verifying data.

The referenced key files and header files are attached to this manual: See “Appendix” on page 41.

4.1.1 Signing a message

```
/*
-----------------------------------------------------------------------
File        : Sign.c
Purpose     : Sign test data.
--------  END-OF-HEADER  ----------------------------------------------
*/
#include "SECURE.h"
#include "PrivateKey.h" // Header generated from private key file.
#include <stdio.h>

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/
static const unsigned char acData[] = {
  0xE0, 0x00, 0x00, 0x00, 0xE1, 0x00, 0x00, 0x00, 0xE2, 0x00, 0x00, 0x00,
  0xE3, 0x00, 0x00, 0x00, 0xE4, 0x00, 0x00, 0x00, 0xE5, 0x00, 0x00, 0x00,
  0xE6, 0x00, 0x00, 0x00, 0xE7, 0x00, 0x00, 0x00, 0xE8, 0x00, 0x00, 0x00,
  0xE9, 0x00, 0x00, 0x00, 0xEA, 0x00, 0x00, 0x00, 0xEB, 0x00, 0x00, 0x00,
  0xEC, 0x00, 0x00, 0x00, 0xED, 0x00, 0x00, 0x00, 0xEE, 0x00, 0x00, 0x00,
  0xEF, 0x00, 0x00, 0x00, 0xF0, 0x00, 0x00, 0x00, 0xF1, 0x00, 0x00, 0x00,
  0xF2, 0x00, 0x00, 0x00, 0xF3, 0x00, 0x00, 0x00, 0xF4, 0x00, 0x00, 0x00,
  0xF5, 0x00, 0x00, 0x00, 0xF6, 0x00, 0x00, 0x00, 0xF7, 0x00, 0x00, 0x00
};
static const unsigned char acSaltIn[18]  = "SEGGER emSecure";

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static unsigned char acSig[256];

/*********************************************************************
*
*       Code
*
**********************************************************************
*/
/*********************************************************************
*
*       _Sign()
*
*/
static int _Sign(void) {
  int r;

  r = SECURE_Sign (&SecureKey_PrivateKey,                                  \
                   acSaltIn, sizeof(acSaltIn), acData, sizeof(acData),     \
                   acSig, sizeof(acSig));
  return r;
}

/*********************************************************************
*
*       main()
*
*/
int main(void) {
  if (_Sign() != 0) {
    printf("Message signed.\n");
  } else {
    printf("Failed to sign message!\n");
    return 1;
  }
  return 0;
}
```



```
/****** End Of File *************************************************/
```

4.1.2 Verifying a message

```
/*
-----------------------------------------------------------------------
File        : Verify.c
Purpose     : Verify test data and signature.
--------  END-OF-HEADER  ----------------------------------------------
*/
#include "SECURE.h"
#include "PublicKey.h"  // Header generated from public key file.
#include <stdio.h>

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/
static const unsigned char acData[] = {
  0xE0, 0x00, 0x00, 0x00, 0xE1, 0x00, 0x00, 0x00, 0xE2, 0x00, 0x00, 0x00,
  0xE3, 0x00, 0x00, 0x00, 0xE4, 0x00, 0x00, 0x00, 0xE5, 0x00, 0x00, 0x00,
  0xE6, 0x00, 0x00, 0x00, 0xE7, 0x00, 0x00, 0x00, 0xE8, 0x00, 0x00, 0x00,
  0xE9, 0x00, 0x00, 0x00, 0xEA, 0x00, 0x00, 0x00, 0xEB, 0x00, 0x00, 0x00,
  0xEC, 0x00, 0x00, 0x00, 0xED, 0x00, 0x00, 0x00, 0xEE, 0x00, 0x00, 0x00,
  0xEF, 0x00, 0x00, 0x00, 0xF0, 0x00, 0x00, 0x00, 0xF1, 0x00, 0x00, 0x00,
  0xF2, 0x00, 0x00, 0x00, 0xF3, 0x00, 0x00, 0x00, 0xF4, 0x00, 0x00, 0x00,
  0xF5, 0x00, 0x00, 0x00, 0xF6, 0x00, 0x00, 0x00, 0xF7, 0x00, 0x00, 0x00
};
static const unsigned char acSig[] = {
  0x31, 0xc6, 0x93, 0xd6, 0x36, 0xa3, 0x5e, 0xcf, 0x9f, 0x22, 0xca, 0x37,
  0x0e, 0x63, 0x37, 0x8f, 0x9c, 0x95, 0x28, 0xcc, 0x25, 0xc6, 0x26, 0xc5,
  0xb0, 0xca, 0x0d, 0xa3, 0x8a, 0xbd, 0xba, 0xf8, 0xb3, 0xf0, 0x31, 0xcc,
  0xb9, 0x07, 0x71, 0xa3, 0xde, 0xb3, 0x98, 0xd6, 0xb4, 0x06, 0x28, 0x5d,
  0xa8, 0x56, 0x5c, 0x51, 0x20, 0xc7, 0x50, 0xf1, 0xff, 0xb2, 0xe9, 0x31,
  0x3d, 0x93, 0x5a, 0x83, 0x01, 0x4b, 0xf0, 0x37, 0x81, 0x2d, 0xbd, 0xff,
  0x5c, 0x24, 0x67, 0x29, 0x66, 0x6e, 0x1a, 0x99, 0xfc, 0x08, 0x41, 0x1a,
  0xda, 0x87, 0xca, 0x1e, 0xeb, 0x04, 0x13, 0x14, 0x36, 0xd7, 0xa4, 0xf8,
  0x8a, 0x06, 0x0b, 0x6c, 0x06, 0xa1, 0xac, 0x0e, 0x7e, 0xbe, 0x34, 0x7a,
  0x38, 0x38, 0x16, 0xaa, 0xc8, 0xd9, 0x2d, 0x55, 0x13, 0x04, 0x29, 0x46,
  0x55, 0xc8, 0x31, 0x32, 0xb1, 0x9f, 0x81, 0x5c, 0xa5, 0xc7, 0x74, 0x08,
  0x64, 0x87, 0x1c, 0x48, 0x87, 0x4d, 0xab, 0x7b, 0x59, 0x9f, 0x89, 0x17,
  0xc7, 0xeb, 0x24, 0x46, 0xba, 0x0b, 0xbd, 0x6d, 0x53, 0xe8, 0x2c, 0x64,
  0x36, 0xec, 0x78, 0x5c, 0x15, 0x0d, 0x6d, 0x65, 0xe3, 0x82, 0xaf, 0x4d,
  0xd8, 0xee, 0x38, 0xd0, 0xb4, 0xfd, 0xd8, 0x75, 0x8d, 0x3b, 0xd7, 0x28,
  0x93, 0x8d, 0x3b, 0x18, 0x35, 0xf9, 0xe1, 0xbb, 0x2e, 0xff, 0xc9, 0xa0,
  0xd1, 0x5f, 0x6f, 0xf6, 0x47, 0xcd, 0x4c, 0xb5, 0x02, 0x0b, 0xc5, 0x2e,
  0x08, 0xe0, 0x32, 0x82, 0x32, 0xc7, 0x1a, 0xa8, 0x90, 0x4a, 0x23, 0x4f,
  0x72, 0xe8, 0x95, 0xad, 0x66, 0x73, 0xdf, 0xf0, 0xf1, 0x4d, 0x17, 0xff,
  0x11, 0x15, 0xf6, 0x72, 0x6f, 0x96, 0xcf, 0xaf, 0x58, 0x4d, 0xfa, 0xc2,
  0x12, 0x1c, 0x7d, 0x38, 0x96, 0xe3, 0xb2, 0x13, 0x43, 0x00, 0x94, 0x3b,
  0x7e, 0x46, 0xcf, 0x50
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static unsigned char acSaltOut[16];

/*********************************************************************
*
*       Code
*
**********************************************************************
*/
/*********************************************************************
*
*       _Verify()
*
*/
static int _Verify(void) {
  int r;

  r = SECURE_Verify (&SecureKey_PublicKey,                                \
                acSaltOut, sizeof(acSaltIn), acData, sizeof(acData),      \
                           acSig, sizeof(acSig));
```



```
  return r;
}

/*************************************************************************
*
*       main()
*
*/
int main(void) {
  if (_Verify() == 1) {
    printf("%s", acSaltOut);
    printf("Message verified.\n");
  } else {
    printf("Failed to verify message!\n");
    return 1;
  }
  return 0;
}

/****** End Of File *************************************************/
```


Chapter 5

Background information

This chapter provides background information about the the modules of emSecure.

To use emSecure knowledge about these modules and their functionality is not necessarily required.

5.1 RSA

RSA (named after Ron Rivest, Adi Shamir and Leonard Adleman) is the first published asymmetric public-key cryptosymstem algorithm.

RSA uses a set of two keys. One key, the public key, for encryption of data and verification of digital signatures, the other key, the private key, for decrypting and digitally signing data.

RSA keys are generated using two large prime numbers (P and Q). With the knowledge of the public key, consisting of the product of the two prime factors (the modulus, M) and an auxiliary value (E) it is possible to encrypt a message which can then only be decrypted with the private key, but due to the problem of factoring a number with large enough prime factors, there is no known efficient method to get the private key's value (D) or the two primes from the public key, when the prime factors are kept secret.

Encryption algorithm	Decryption algorithm
$c = m^E \bmod(N)$	$m=c^D \bmod(N)$

```
m: Message
c: Ciphertext
E: Public value
D: Private value
N: Modulus (P*Q)
```

Table 5.1: RSA algorithms

For emSecure RSA the modulus (N) can be chosen to be of a width between 512 and 4096 bits. In practice RSA is presumed to be safe if N is large enough. In 2010 an RSA modulus of 768 bits has been factorized, there is no known factorization of a higher number.

A modulus of 1024 bits is graded as commercial grade, whereas a 2048 bit modulus is of military grade. Both widths are presumed to be safe for now.

RSA is often used in hybrid crptosystems because it is known to be 1000 times slower than symmetric algorithms like AES. Therefore RSA is for example used to exchange the encrypted password for symmetric encrypted communication.

5.2 MPI

MPI, Multi-Precision Integers, is the an implementation of Big number (also called arbtrary-precision) arithmetic.

It is used to store integers of huge lengths and provides all arithmetic capabilities needed in emSecure.

The MPI module includes:

- Formatted output, loading of formatted representations.
- Assignment of basic integer types and bignums.
- Arithmetic: Addition, subtraction, multiplication, division, modular operation, exponentiation.
- Bit and Byte manipulation, shifting and logical functions.
- Comparison of numbers.
- Number theoretic functions.

MPI is not meant to be a full-featured bignum library.

5.3 Helper Modules

The following modules are used by the main emSecure modules and provide helping functions as needed by the emSecure implementation.

5.3.1 MGF

MGF, a Mask Generation Function, generates a mask, an arbitrary number of bits, using a random seed and a message's hash value.

5.3.2 PKCS #1

PKCS #1, the Public-Key Cryptography Standards, is the first set of standards for cryptography, published by RSA Laboratories. It describes the definitions and recommendations for RSA as well as the properties of public and private keys, primitive operations and secure cryptographic schemes.

5.3.3 SHA-1

SHA, Secure Hash Algorithm, is a set of cryptographic hash functions. The hash functions were designed by the U.S. National Security Agency (NSA) and NIST as a FIP Standard.

Secure Hash Algorithms are used to create a unique hash value for any message which is used as the base for digital signatures and provides the integrity of a message. This requires the collision resistance of hash values, the should be practically no two different messages with the same hash value.

SHA-1 produces a 160 bit secure hash of a message of up to 2^{64}-1 bit.

Chapter 6

Support

This chapter includes general help to emSecure and answers frequently asked questions.

6.1 FAQ

Q: I want to prevent copying a whole firmware from one product hardware to another cloned one. How can I prevent it to be run from the cloned version with emSecure?

A: Nearly every modern MCU includes a unique ID, which is different on every device. When the signature covers this UID it is only valid on one single device and cannot be run on a cloned or copied product. The firmware can verify the signature at boot-time.

Q: I added a digital signature to my product. Where should I verify it?

A: Signature verification can be done in-product or off-product. With in-product verification the firmware for example verifies the digital signature at boot-time and refuses to run when the signature cannot be verified. With off-product verification an external application, e.g. a PC application communicating with the device, reads the signature and data from the product and verifies it.

Q: I want my product to only run genuine firmware images. How can I achieve this with emSecure?

A: To make sure a firmware image is genuine, the complete image can be signed with a digital signature. Like when using a CRC for integrity checks, the signature is sent with the firmware data upon a firmware update. The application or bootloader programming the firmware onto the device validates the firmware data with its signature. The signature can only be generated with the private key and should be provided by the developer with the firmware data.

Q: I am providing additional licenses for my product which shall be locked to a specific user or computer. Can I generate license keys with emSecure?

A: Yes. emSecure can generate unique license keys for any data, like a computer ID, a user name, e-mail address or any other data.

Q: My product is sending data to a computer application. Can I make sure the computer application is getting data only from my product with emSecure?

A: Yes. In this case the product is used to sign the data and the computer applications verifies it. To prevent the private key from being read from the product it might be stored encrypted on the product or in the application and decrypted prior to signing the data.

Chapter 7

Appendix

7.1 Example key pair

The example key pair was generated with emKeyGen, key length 2048 bit, proven prime algorithm, initial seed password "SEGGER - The embedded experts":

```
C:\>emKeyGen.exe -rsa -f -l 2048 -pw "SEGGER - The embedded experts"

(c) 2014 SEGGER Microcontroller GmbH & Co. KG
         www.segger.com
emSecure KeyGen V1.00 compiled Sep 17 2014 10:47:52

Generating proven prime key pair with public modulus of 2048 bits
Public encryption exponent is set to 65537
Initial seed is 0x9A1E8FE407FA6A5437331E1696341657
Checking keys are consistent: OK
Writing public key file emSecure.pub.
Writing private key file emSecure.prv.
```

7.1.1 emSecure.prv - Example private key file

```
#
# RSA Private Key - Generated by SEGGER emSecure V1.00
#

D=0x0DEE48B25D8EF7745E38E20ABAA4925FE0C24280AAD62E02B47828E58B193E
B77FAC274EEDE412B42159557EF92DB0921213237CF273BD722C4BA1D65F24A0E4
0CD62BAECC6BB02DC09592862011589CF71FBFC410851300B0FC2241309529B76F7
56E0C1D47D79FD2873DFD5264E1596E30183719A1C82BBD32B3C2D3FC71D157511
84F2F4C27297A84D78C9AF202C9AA115302BB71FD8D70A42C71D6C6AC8B702E077
23A3E65F06CA885DA709805DAB27F3EA6F155566BC3A39843038D6A32B53CD2455
162EE9B2F9A968DF4D9D455D5E72A238527CFFB0EF8F952B65D200093AC3D9B1B9
9D6A89646FF48E787552D14B62208896CEEFC41B432868BD137A1

P=0xC111B29C77B1A2AB50DA6F2315B5777B1663F935C3553B2FEA905F28A4591CF
A449501A3688BF44F41C54D562B82C22BA249873507A9F9964D967BC4C2BC9D80E
5D78D914B0A366A4A1777E9CFEE4B1CD8243783E16B6D13FD604D4DA9860D82D4
B8E16258150C26FB18AAD33C209B739EB252C1EC13C462C681660B14CDED19

Q=0xD35A03CAEC450819D4B7FFA48EEF6A008DBDA368F3B3B65568C374DEB7B10
C792B1A550D83A6619F59A116D6D19FC763C1C99CA61CE4820021CE71F38AB112A
24CE0431D5D44FAB078B3BD019837581B14BD326AAC2E6CEEACE1D7BA4E78A85D
0B4553F0DBA69219485E77902F903C0D2ED1C46778729F9739AB80F633240611

DP=0xA02465DA930DACB81D2091FEB00B0D47F392892BA712133DF37A3CF4211E2
9830D4FEB43F77BDEF1BEC44119B08D8D315433590B0B885995EA555FE41D03064
1DC63A6F15524EB6DCE7718E7BFA91074A473A5F8A609CD383F7A99A44970FFD3F
D4E3CE4ADDB07716DE5500C565B5595D9946040A9E8DB8472D2F2294EE06041

DQ=0xACB023E12DE3C996B18415D13328D387D84856B86E472C77E4BDAF443AFA
E2A22E61B994ED388913567C94D8B936C007F652F13798EBEC7C8722D36096B5CFE
FE4B8689E19933EF1E9ED92453746212B1F6AE742D9A4A544EEE5290B050AF925ED
0B9C667452499576D819012A9BC3355247FB1E400007FDDC1511AE18631AC1

QINV=0x3C88C18E6BB44F6801A1AE5CB71C56BC5703B01C2D829F3B4CCFCF364CD
8948CB44D56B1B9C4A45A494EE654C8B0A9EB028B5F9E045EB4E58CDB4C88EBFC7
68DE7EBD118E05643A20A7007B1E4EEB7185CC8CBC4A85CE97376144074A0C8D3A
19FD68E17F20B4C3328835F46E2C2AE43599ED2A3685439F41B5EB6D37BFC3410

#
# End Of File
#
```

7.1.2 emSecure.pub - Example public key file

```
#
# RSA Public Key - Generated by SEGGER emSecure V1.00
#
```



```
N=0x9F65794C00E8A2ED563199BE5A0D077ECB2A75E104F727A0BB7CF8B6718992
BB7D40A38AA401DBEB0A60F54C55CC7A68753E5739BB50084F2FFB7273BC6BF06E
8FFE6A0969B0F08594F2C9504F2CE489B14D5F6D0E7F9AB61BA2C70FF003DB6BFAA
16FFEE5B4D77546462BE49E8A850C31300802513A31E15565D2E7878F5C531BE056
77D20FC89EF0DE7344F1C7F7225B66BF2EC394ED56B7B4A409D3DDE241DE646706
FDA2177C53BACC392A6E6A59C8BF38A5BE40750B5ECB88308B92C049F7C7395A2E
2B80B6074A1A1DAC4645E463F27A54F639489CE1CB0077D07A434F06FDFC71BA04
0347B0F0066A90401EAA15A115FEFC9BC5E0AAD1527087BF54A9

E=0x010001

#
# End Of File
#
```

7.1.3 PrivateKey.h - Compilable example private key

PrivateKey.h was generated from the example private key using emPrintKey, parameter prefix: "PrivateKeyParam_":

```
C:\>emPrintKey.exe -p PrivateKeyParam_ -m emSecure.prv > PrivateKey.h
```

```
//
// Generated by emSecure V1.00
//

#include "MPI.h"

#include "RSA.h"

#define PrivateKey_D_LITERAL_DATA \
  __MPI_LITERAL_DATA(0xa1, 0x37, 0xd1, 0x8b), \
  __MPI_LITERAL_DATA(0x86, 0x32, 0xb4, 0x41), \
  __MPI_LITERAL_DATA(0xfc, 0xee, 0x6c, 0x89), \
  __MPI_LITERAL_DATA(0x08, 0x22, 0xb6, 0x14), \
  __MPI_LITERAL_DATA(0x2d, 0x55, 0x87, 0xe7), \
  __MPI_LITERAL_DATA(0x48, 0xff, 0x46, 0x96), \
  __MPI_LITERAL_DATA(0xa8, 0xd6, 0x99, 0x1b), \
  __MPI_LITERAL_DATA(0x9b, 0x3d, 0xac, 0x93), \
  __MPI_LITERAL_DATA(0x00, 0x20, 0x5d, 0xb6), \
  __MPI_LITERAL_DATA(0x52, 0xf9, 0xf8, 0x0e), \
  __MPI_LITERAL_DATA(0xfb, 0xcf, 0x27, 0x85), \
  __MPI_LITERAL_DATA(0x23, 0x2a, 0xe7, 0xd5), \
  __MPI_LITERAL_DATA(0x55, 0xd4, 0xd9, 0xf4), \
  __MPI_LITERAL_DATA(0x8d, 0x96, 0x9a, 0x2f), \
  __MPI_LITERAL_DATA(0x9b, 0xee, 0x62, 0x51), \
  __MPI_LITERAL_DATA(0x45, 0xd2, 0x3c, 0xb5), \
  __MPI_LITERAL_DATA(0x32, 0x6a, 0x8d, 0x03), \
  __MPI_LITERAL_DATA(0x43, 0x98, 0xa3, 0xc3), \
  __MPI_LITERAL_DATA(0x6b, 0x56, 0x55, 0xf1), \
  __MPI_LITERAL_DATA(0xa6, 0x3e, 0x7f, 0xb2), \
  __MPI_LITERAL_DATA(0xda, 0x05, 0x98, 0x70), \
  __MPI_LITERAL_DATA(0xda, 0x85, 0xa8, 0x6c), \
  __MPI_LITERAL_DATA(0xf0, 0x65, 0x3e, 0x3a), \
  __MPI_LITERAL_DATA(0x72, 0x07, 0x2e, 0x70), \
  __MPI_LITERAL_DATA(0x8b, 0xac, 0xc6, 0xd6), \
  __MPI_LITERAL_DATA(0x71, 0x2c, 0xa4, 0x70), \
  __MPI_LITERAL_DATA(0x8d, 0xfd, 0x71, 0xbb), \
  __MPI_LITERAL_DATA(0x02, 0x53, 0x11, 0xaa), \
  __MPI_LITERAL_DATA(0xc9, 0x02, 0xf2, 0x9a), \
  __MPI_LITERAL_DATA(0x8c, 0xd7, 0x84, 0x7a), \
  __MPI_LITERAL_DATA(0x29, 0x27, 0x4c, 0x2f), \
  __MPI_LITERAL_DATA(0x4f, 0x18, 0x51, 0x57), \
  __MPI_LITERAL_DATA(0xd1, 0x71, 0xfc, 0xd3), \
  __MPI_LITERAL_DATA(0xc2, 0xb3, 0x32, 0xbd), \
  __MPI_LITERAL_DATA(0x2b, 0xc8, 0xa1, 0x19), \
  __MPI_LITERAL_DATA(0x37, 0x18, 0x30, 0x6e), \
  __MPI_LITERAL_DATA(0x59, 0xe1, 0x64, 0x52), \
  __MPI_LITERAL_DATA(0xfd, 0x3d, 0x87, 0xd2), \
  __MPI_LITERAL_DATA(0x9f, 0xd7, 0x47, 0x1d), \
  __MPI_LITERAL_DATA(0x0c, 0x6e, 0x75, 0x6f), \
  __MPI_LITERAL_DATA(0xb7, 0x29, 0x95, 0x30), \
  __MPI_LITERAL_DATA(0x41, 0x22, 0xfc, 0xb0), \
  __MPI_LITERAL_DATA(0x00, 0x13, 0x85, 0x10), \
  __MPI_LITERAL_DATA(0xc4, 0xbf, 0x1f, 0xf7), \
```



```
  __MPI_LITERAL_DATA(0x9c, 0x58, 0x11, 0x20), \
  __MPI_LITERAL_DATA(0x86, 0x92, 0x95, 0xc0), \
  __MPI_LITERAL_DATA(0x2d, 0xb0, 0x6b, 0xcc), \
  __MPI_LITERAL_DATA(0xae, 0x2b, 0xd6, 0x0c), \
  __MPI_LITERAL_DATA(0xe4, 0xa0, 0x24, 0x5f), \
  __MPI_LITERAL_DATA(0xd6, 0xa1, 0x4b, 0x2c), \
  __MPI_LITERAL_DATA(0x72, 0xbd, 0x73, 0xf2), \
  __MPI_LITERAL_DATA(0x7c, 0x23, 0x13, 0x12), \
  __MPI_LITERAL_DATA(0x92, 0xb0, 0x2d, 0xf9), \
  __MPI_LITERAL_DATA(0x7e, 0x55, 0x59, 0x21), \
  __MPI_LITERAL_DATA(0xb4, 0x12, 0xe4, 0xed), \
  __MPI_LITERAL_DATA(0x4e, 0x27, 0xac, 0x7f), \
  __MPI_LITERAL_DATA(0xb7, 0x3e, 0x19, 0x8b), \
  __MPI_LITERAL_DATA(0xe5, 0x28, 0x78, 0xb4), \
  __MPI_LITERAL_DATA(0x02, 0x2e, 0xd6, 0xaa), \
  __MPI_LITERAL_DATA(0x80, 0x42, 0xc2, 0xe0), \
  __MPI_LITERAL_DATA(0x5f, 0x92, 0xa4, 0xba), \
  __MPI_LITERAL_DATA(0x0a, 0xe2, 0x38, 0x5e), \
  __MPI_LITERAL_DATA(0x74, 0xf7, 0x8e, 0x5d), \
  __MPI_LITERAL_DATA(0xb2, 0x48, 0xee, 0x0d)

__MPI_LITERAL_BEGIN(static, PrivateKey_D)
  PrivateKey_D_LITERAL_DATA
__MPI_LITERAL_END(static, PrivateKey_D, 2044)

#define PrivateKey_P_LITERAL_DATA \
  __MPI_LITERAL_DATA(0x19, 0xed, 0xcd, 0x14), \
  __MPI_LITERAL_DATA(0x0b, 0x66, 0x81, 0xc6), \
  __MPI_LITERAL_DATA(0x62, 0xc4, 0x13, 0xec), \
  __MPI_LITERAL_DATA(0xc1, 0x52, 0xb2, 0x9e), \
  __MPI_LITERAL_DATA(0x73, 0x9b, 0x20, 0x3c), \
  __MPI_LITERAL_DATA(0xd3, 0xaa, 0x18, 0xfb), \
  __MPI_LITERAL_DATA(0x26, 0x0c, 0x15, 0x58), \
  __MPI_LITERAL_DATA(0x62, 0xe1, 0xb8, 0xd4), \
  __MPI_LITERAL_DATA(0x82, 0x0d, 0x86, 0xa9), \
  __MPI_LITERAL_DATA(0x4d, 0x4d, 0x60, 0xfd), \
  __MPI_LITERAL_DATA(0x13, 0x6d, 0x6b, 0xe1), \
  __MPI_LITERAL_DATA(0x83, 0x37, 0x24, 0xd8), \
  __MPI_LITERAL_DATA(0x1c, 0x4b, 0xee, 0xcf), \
  __MPI_LITERAL_DATA(0xe9, 0x77, 0x17, 0x4a), \
  __MPI_LITERAL_DATA(0x6a, 0x36, 0x0a, 0x4b), \
  __MPI_LITERAL_DATA(0x91, 0x8d, 0xd7, 0xe5), \
  __MPI_LITERAL_DATA(0x80, 0x9d, 0xbc, 0xc2), \
  __MPI_LITERAL_DATA(0xc4, 0x7b, 0x96, 0x4d), \
  __MPI_LITERAL_DATA(0x96, 0xf9, 0xa9, 0x07), \
  __MPI_LITERAL_DATA(0x35, 0x87, 0x49, 0xa2), \
  __MPI_LITERAL_DATA(0x2b, 0xc2, 0x82, 0x2b), \
  __MPI_LITERAL_DATA(0x56, 0x4d, 0xc5, 0x41), \
  __MPI_LITERAL_DATA(0x4f, 0xf4, 0x8b, 0x68), \
  __MPI_LITERAL_DATA(0xa3, 0x01, 0x95, 0x44), \
  __MPI_LITERAL_DATA(0xfa, 0x1c, 0x59, 0xa4), \
  __MPI_LITERAL_DATA(0x28, 0x5f, 0x90, 0xea), \
  __MPI_LITERAL_DATA(0x2f, 0x3b, 0x55, 0xc3), \
  __MPI_LITERAL_DATA(0x35, 0xf9, 0x63, 0x16), \
  __MPI_LITERAL_DATA(0x7b, 0x77, 0xb5, 0x15), \
  __MPI_LITERAL_DATA(0x23, 0x6f, 0xda, 0x50), \
  __MPI_LITERAL_DATA(0xab, 0xa2, 0xb1, 0x77), \
  __MPI_LITERAL_DATA(0x9c, 0xb2, 0x11, 0xc1)

__MPI_LITERAL_BEGIN(static, PrivateKey_P)
  PrivateKey_P_LITERAL_DATA
__MPI_LITERAL_END(static, PrivateKey_P, 1024)

#define PrivateKey_Q_LITERAL_DATA \
  __MPI_LITERAL_DATA(0x11, 0x06, 0x24, 0x33), \
  __MPI_LITERAL_DATA(0xf6, 0x80, 0xab, 0x39), \
  __MPI_LITERAL_DATA(0x97, 0x9f, 0x72, 0x78), \
  __MPI_LITERAL_DATA(0x67, 0xc4, 0xd1, 0x2e), \
  __MPI_LITERAL_DATA(0x0d, 0x3c, 0x90, 0x2f), \
  __MPI_LITERAL_DATA(0x90, 0x77, 0x5e, 0x48), \
  __MPI_LITERAL_DATA(0x19, 0x92, 0xa6, 0xdb), \
  __MPI_LITERAL_DATA(0xf0, 0x53, 0x45, 0x0b), \
  __MPI_LITERAL_DATA(0x5d, 0xa8, 0x78, 0x4e), \
  __MPI_LITERAL_DATA(0xba, 0xd7, 0xe1, 0xac), \
  __MPI_LITERAL_DATA(0xee, 0x6c, 0x2e, 0xac), \
  __MPI_LITERAL_DATA(0x6a, 0x32, 0xbd, 0x14), \
  __MPI_LITERAL_DATA(0x1b, 0x58, 0x37, 0x98), \
  __MPI_LITERAL_DATA(0x01, 0xbd, 0xb3, 0x78), \
  __MPI_LITERAL_DATA(0xb0, 0xfa, 0x44, 0x5d), \
  __MPI_LITERAL_DATA(0x1d, 0x43, 0xe0, 0x4c), \
  __MPI_LITERAL_DATA(0xa2, 0x12, 0xb1, 0x8a), \
```



```
  __MPI_LITERAL_DATA(0xf3, 0x71, 0xce, 0x21), \
  __MPI_LITERAL_DATA(0x00, 0x82, 0xe4, 0x1c), \
  __MPI_LITERAL_DATA(0xa6, 0x9c, 0xc9, 0xc1), \
  __MPI_LITERAL_DATA(0x63, 0xc7, 0x9f, 0xd1), \
  __MPI_LITERAL_DATA(0xd6, 0x16, 0xa1, 0x59), \
  __MPI_LITERAL_DATA(0x9f, 0x61, 0xa6, 0x83), \
  __MPI_LITERAL_DATA(0x0d, 0x55, 0x1a, 0x2b), \
  __MPI_LITERAL_DATA(0x79, 0x0c, 0xb1, 0xb7), \
  __MPI_LITERAL_DATA(0xde, 0x74, 0xc3, 0x68), \
  __MPI_LITERAL_DATA(0x55, 0xb6, 0xb3, 0xf3), \
  __MPI_LITERAL_DATA(0x68, 0xa3, 0xbd, 0x8d), \
  __MPI_LITERAL_DATA(0x00, 0x6a, 0xef, 0x8e), \
  __MPI_LITERAL_DATA(0xa4, 0xff, 0xb7, 0xd4), \
  __MPI_LITERAL_DATA(0x19, 0x08, 0x45, 0xec), \
  __MPI_LITERAL_DATA(0xca, 0x03, 0x5a, 0xd3)

__MPI_LITERAL_BEGIN(static, PrivateKey_Q)
  PrivateKey_Q_LITERAL_DATA
__MPI_LITERAL_END(static, PrivateKey_Q, 1024)

#define PrivateKey_DP_LITERAL_DATA \
  __MPI_LITERAL_DATA(0x41, 0x60, 0xe0, 0x4e), \
  __MPI_LITERAL_DATA(0x29, 0xf2, 0xd2, 0x72), \
  __MPI_LITERAL_DATA(0x84, 0xdb, 0xe8, 0xa9), \
  __MPI_LITERAL_DATA(0x40, 0x60, 0x94, 0xd9), \
  __MPI_LITERAL_DATA(0x95, 0x55, 0x5b, 0x56), \
  __MPI_LITERAL_DATA(0x0c, 0x50, 0xe5, 0x6d), \
  __MPI_LITERAL_DATA(0x71, 0x07, 0xdb, 0xad), \
  __MPI_LITERAL_DATA(0xe4, 0x3c, 0x4e, 0xfd), \
  __MPI_LITERAL_DATA(0xd3, 0xff, 0x70, 0x49), \
  __MPI_LITERAL_DATA(0xa4, 0x99, 0x7a, 0x3f), \
  __MPI_LITERAL_DATA(0x38, 0xcd, 0x09, 0xa6), \
  __MPI_LITERAL_DATA(0xf8, 0xa5, 0x73, 0xa4), \
  __MPI_LITERAL_DATA(0x74, 0x10, 0xa9, 0xbf), \
  __MPI_LITERAL_DATA(0xe7, 0x18, 0x77, 0xce), \
  __MPI_LITERAL_DATA(0x6d, 0xeb, 0x24, 0x55), \
  __MPI_LITERAL_DATA(0xf1, 0xa6, 0x63, 0xdc), \
  __MPI_LITERAL_DATA(0x41, 0x06, 0x03, 0x1d), \
  __MPI_LITERAL_DATA(0xe4, 0x5f, 0x55, 0xea), \
  __MPI_LITERAL_DATA(0x95, 0x59, 0x88, 0x0b), \
  __MPI_LITERAL_DATA(0x0b, 0x59, 0x33, 0x54), \
  __MPI_LITERAL_DATA(0x31, 0x8d, 0x8d, 0xb0), \
  __MPI_LITERAL_DATA(0x19, 0x41, 0xc4, 0xbe), \
  __MPI_LITERAL_DATA(0xf1, 0xde, 0x7b, 0xf7), \
  __MPI_LITERAL_DATA(0x43, 0xeb, 0x4f, 0x0d), \
  __MPI_LITERAL_DATA(0x83, 0x29, 0x1e, 0x21), \
  __MPI_LITERAL_DATA(0xf4, 0x3c, 0x7a, 0xf3), \
  __MPI_LITERAL_DATA(0x3d, 0x13, 0x12, 0xa7), \
  __MPI_LITERAL_DATA(0x2b, 0x89, 0x92, 0xf3), \
  __MPI_LITERAL_DATA(0x47, 0x0d, 0x0b, 0xb0), \
  __MPI_LITERAL_DATA(0xfe, 0x91, 0x20, 0x1d), \
  __MPI_LITERAL_DATA(0xb8, 0xac, 0x0d, 0x93), \
  __MPI_LITERAL_DATA(0xda, 0x65, 0x24, 0xa0)

__MPI_LITERAL_BEGIN(static, PrivateKey_DP)
  PrivateKey_DP_LITERAL_DATA
__MPI_LITERAL_END(static, PrivateKey_DP, 1024)

#define PrivateKey_DQ_LITERAL_DATA \
  __MPI_LITERAL_DATA(0xc1, 0x1a, 0x63, 0x18), \
  __MPI_LITERAL_DATA(0xae, 0x11, 0x15, 0xdc), \
  __MPI_LITERAL_DATA(0xfd, 0x07, 0x00, 0x40), \
  __MPI_LITERAL_DATA(0x1e, 0xfb, 0x47, 0x52), \
  __MPI_LITERAL_DATA(0x35, 0xc3, 0x9b, 0x2a), \
  __MPI_LITERAL_DATA(0x01, 0x19, 0xd8, 0x76), \
  __MPI_LITERAL_DATA(0x95, 0x49, 0x52, 0x74), \
  __MPI_LITERAL_DATA(0x66, 0x9c, 0x0b, 0xed), \
  __MPI_LITERAL_DATA(0x25, 0xf9, 0x0a, 0x05), \
  __MPI_LITERAL_DATA(0x0b, 0x29, 0xe5, 0xee), \
  __MPI_LITERAL_DATA(0x44, 0xa5, 0xa4, 0xd9), \
  __MPI_LITERAL_DATA(0x42, 0xe7, 0x6a, 0x1f), \
  __MPI_LITERAL_DATA(0x2b, 0x21, 0x46, 0x37), \
  __MPI_LITERAL_DATA(0x45, 0x92, 0xed, 0xe9), \
  __MPI_LITERAL_DATA(0xf1, 0x3e, 0x93, 0x19), \
  __MPI_LITERAL_DATA(0x9e, 0x68, 0xb8, 0xe4), \
  __MPI_LITERAL_DATA(0xef, 0xcf, 0xb5, 0x96), \
  __MPI_LITERAL_DATA(0x60, 0xd3, 0x22, 0x87), \
  __MPI_LITERAL_DATA(0x7c, 0xec, 0xeb, 0x98), \
  __MPI_LITERAL_DATA(0x37, 0xf1, 0x52, 0xf6), \
  __MPI_LITERAL_DATA(0x07, 0xc0, 0x36, 0xb9), \
  __MPI_LITERAL_DATA(0xd8, 0x94, 0x7c, 0x56), \
```



```
  __MPI_LITERAL_DATA(0x13, 0x89, 0x38, 0xed), \
  __MPI_LITERAL_DATA(0x94, 0xb9, 0x61, 0x2e), \
  __MPI_LITERAL_DATA(0xa2, 0xe2, 0xfa, 0x3a), \
  __MPI_LITERAL_DATA(0x44, 0xaf, 0xbd, 0xe4), \
  __MPI_LITERAL_DATA(0x77, 0x2c, 0x47, 0x6e), \
  __MPI_LITERAL_DATA(0xb8, 0x56, 0x48, 0xd8), \
  __MPI_LITERAL_DATA(0x87, 0xd3, 0x28, 0x33), \
  __MPI_LITERAL_DATA(0xd1, 0x15, 0x84, 0xb1), \
  __MPI_LITERAL_DATA(0x96, 0xc9, 0xe3, 0x2d), \
  __MPI_LITERAL_DATA(0xe1, 0x23, 0xb0, 0xac)

__MPI_LITERAL_BEGIN(static, PrivateKey_DQ)
  PrivateKey_DQ_LITERAL_DATA
__MPI_LITERAL_END(static, PrivateKey_DQ, 1024)

#define PrivateKey_QINV_LITERAL_DATA \
  __MPI_LITERAL_DATA(0x10, 0x34, 0xfc, 0x7b), \
  __MPI_LITERAL_DATA(0xd3, 0xb6, 0x5e, 0x1b), \
  __MPI_LITERAL_DATA(0xf4, 0x39, 0x54, 0x68), \
  __MPI_LITERAL_DATA(0xa3, 0xd2, 0x9e, 0x59), \
  __MPI_LITERAL_DATA(0x43, 0xae, 0xc2, 0xe2), \
  __MPI_LITERAL_DATA(0x46, 0x5f, 0x83, 0x28), \
  __MPI_LITERAL_DATA(0x33, 0x4c, 0x0b, 0xf2), \
  __MPI_LITERAL_DATA(0x17, 0x8e, 0xd6, 0x9f), \
  __MPI_LITERAL_DATA(0xa1, 0xd3, 0xc8, 0xa0), \
  __MPI_LITERAL_DATA(0x74, 0x40, 0x14, 0x76), \
  __MPI_LITERAL_DATA(0x73, 0xe9, 0x5c, 0xa8), \
  __MPI_LITERAL_DATA(0xc4, 0xcb, 0xc8, 0x5c), \
  __MPI_LITERAL_DATA(0x18, 0xb7, 0xee, 0xe4), \
  __MPI_LITERAL_DATA(0xb1, 0x07, 0x70, 0x0a), \
  __MPI_LITERAL_DATA(0xa2, 0x43, 0x56, 0xe0), \
  __MPI_LITERAL_DATA(0x18, 0xd1, 0xeb, 0xe7), \
  __MPI_LITERAL_DATA(0x8d, 0x76, 0xfc, 0xeb), \
  __MPI_LITERAL_DATA(0x88, 0x4c, 0xdb, 0x8c), \
  __MPI_LITERAL_DATA(0xe5, 0xb4, 0x5e, 0x04), \
  __MPI_LITERAL_DATA(0x9e, 0x5f, 0x8b, 0x02), \
  __MPI_LITERAL_DATA(0xeb, 0xa9, 0xb0, 0xc8), \
  __MPI_LITERAL_DATA(0x54, 0xe6, 0x4e, 0x49), \
  __MPI_LITERAL_DATA(0x5a, 0xa4, 0xc4, 0xb9), \
  __MPI_LITERAL_DATA(0xb1, 0x56, 0x4d, 0xb4), \
  __MPI_LITERAL_DATA(0x8c, 0x94, 0xd8, 0x4c), \
  __MPI_LITERAL_DATA(0x36, 0xcf, 0xcf, 0x4c), \
  __MPI_LITERAL_DATA(0x3b, 0x9f, 0x82, 0x2d), \
  __MPI_LITERAL_DATA(0x1c, 0xb0, 0x03, 0x57), \
  __MPI_LITERAL_DATA(0xbc, 0x56, 0x1c, 0xb7), \
  __MPI_LITERAL_DATA(0x5c, 0xae, 0xa1, 0x01), \
  __MPI_LITERAL_DATA(0x68, 0x4f, 0xb4, 0x6b), \
  __MPI_LITERAL_DATA(0x8e, 0xc1, 0x88, 0x3c)

__MPI_LITERAL_BEGIN(static, PrivateKey_QINV)
  PrivateKey_QINV_LITERAL_DATA
__MPI_LITERAL_END(static, PrivateKey_QINV, 1022)

static const RSA_PRIVATE_KEY PrivateKey_PrivateKey = {
  {
#if MPI_MAX_BIT_COUNT >= 32
    { PrivateKey_D_LITERAL_DATA },
#else
    (MPI_LIMB *)&PrivateKey_D_Limbs,
#endif
    (2044+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    (2044+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    0,
    1 },
  {
#if MPI_MAX_BIT_COUNT >= 32
    { PrivateKey_P_LITERAL_DATA },
#else
    (MPI_LIMB *)&PrivateKey_P_Limbs,
#endif
    (1024+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    (1024+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    0,
    1 },
  {
#if MPI_MAX_BIT_COUNT >= 32
    { PrivateKey_Q_LITERAL_DATA },
#else
    (MPI_LIMB *)&PrivateKey_Q_Limbs,
#endif
    (1024+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
```



```
    (1024+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    0,
    1 },
  {
#if MPI_MAX_BIT_COUNT >= 32
    { PrivateKey_DP_LITERAL_DATA },
#else
    (MPI_LIMB *)&PrivateKey_DP_Limbs,
#endif
    (1024+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    (1024+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    0,
    1 },
  {
#if MPI_MAX_BIT_COUNT >= 32
    { PrivateKey_DQ_LITERAL_DATA },
#else
    (MPI_LIMB *)&PrivateKey_DQ_Limbs,
#endif
    (1024+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    (1024+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    0,
    1 },
  {
#if MPI_MAX_BIT_COUNT >= 32
    { PrivateKey_QINV_LITERAL_DATA },
#else
    (MPI_LIMB *)&PrivateKey_QINV_Limbs,
#endif
    (1022+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    (1022+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    0,
    1 },
};
```

7.1.4 PublicKey.h - Compilable example public key

PublicKey.h was generated from the example public key using emPrintKey, parameter prefix: "PublicKeyParam_":

```
C:\>emPrintKey.exe -p PublicKeyParam_ -m emSecure.pub > PublicKey.h
```

```
//
// Generated by emSecure V1.00
//

#include "MPI.h"

#include "RSA.h"

#define PublicKey_N_LITERAL_DATA \
  __MPI_LITERAL_DATA(0xa9, 0x54, 0xbf, 0x87), \
  __MPI_LITERAL_DATA(0x70, 0x52, 0xd1, 0xaa), \
  __MPI_LITERAL_DATA(0xe0, 0xc5, 0x9b, 0xfc), \
  __MPI_LITERAL_DATA(0xfe, 0x15, 0xa1, 0x15), \
  __MPI_LITERAL_DATA(0xaa, 0x1e, 0x40, 0x90), \
  __MPI_LITERAL_DATA(0x6a, 0x06, 0xf0, 0xb0), \
  __MPI_LITERAL_DATA(0x47, 0x03, 0x04, 0xba), \
  __MPI_LITERAL_DATA(0x71, 0xfc, 0xfd, 0x06), \
  __MPI_LITERAL_DATA(0x4f, 0x43, 0x7a, 0xd0), \
  __MPI_LITERAL_DATA(0x77, 0x00, 0xcb, 0xe1), \
  __MPI_LITERAL_DATA(0x9c, 0x48, 0x39, 0xf6), \
  __MPI_LITERAL_DATA(0x54, 0x7a, 0xf2, 0x63), \
  __MPI_LITERAL_DATA(0xe4, 0x45, 0x46, 0xac), \
  __MPI_LITERAL_DATA(0x1d, 0x1a, 0x4a, 0x07), \
  __MPI_LITERAL_DATA(0xb6, 0x80, 0x2b, 0x2e), \
  __MPI_LITERAL_DATA(0x5a, 0x39, 0xc7, 0xf7), \
  __MPI_LITERAL_DATA(0x49, 0xc0, 0x92, 0x8b), \
  __MPI_LITERAL_DATA(0x30, 0x88, 0xcb, 0x5e), \
  __MPI_LITERAL_DATA(0x0b, 0x75, 0x40, 0xbe), \
  __MPI_LITERAL_DATA(0xa5, 0x38, 0xbf, 0xc8), \
  __MPI_LITERAL_DATA(0x59, 0x6a, 0x6e, 0x2a), \
  __MPI_LITERAL_DATA(0x39, 0xcc, 0xba, 0x53), \
  __MPI_LITERAL_DATA(0x7c, 0x17, 0xa2, 0xfd), \
  __MPI_LITERAL_DATA(0x06, 0x67, 0x64, 0xde), \
  __MPI_LITERAL_DATA(0x41, 0xe2, 0xdd, 0xd3), \
  __MPI_LITERAL_DATA(0x09, 0xa4, 0xb4, 0xb7), \
```



```
  __MPI_LITERAL_DATA(0x56, 0xed, 0x94, 0xc3), \
  __MPI_LITERAL_DATA(0x2e, 0xbf, 0x66, 0x5b), \
  __MPI_LITERAL_DATA(0x22, 0xf7, 0xc7, 0xf1), \
  __MPI_LITERAL_DATA(0x44, 0x73, 0xde, 0xf0), \
  __MPI_LITERAL_DATA(0x9e, 0xc8, 0x0f, 0xd2), \
  __MPI_LITERAL_DATA(0x77, 0x56, 0xe0, 0x1b), \
  __MPI_LITERAL_DATA(0x53, 0x5c, 0x8f, 0x87), \
  __MPI_LITERAL_DATA(0xe7, 0xd2, 0x65, 0x55), \
  __MPI_LITERAL_DATA(0xe1, 0x31, 0x3a, 0x51), \
  __MPI_LITERAL_DATA(0x02, 0x08, 0x30, 0x31), \
  __MPI_LITERAL_DATA(0x0c, 0x85, 0x8a, 0x9e), \
  __MPI_LITERAL_DATA(0xe4, 0x2b, 0x46, 0x46), \
  __MPI_LITERAL_DATA(0x75, 0xd7, 0xb4, 0xe5), \
  __MPI_LITERAL_DATA(0xfe, 0x6f, 0xa1, 0xfa), \
  __MPI_LITERAL_DATA(0x6b, 0xdb, 0x03, 0xf0), \
  __MPI_LITERAL_DATA(0x0f, 0xc7, 0xa2, 0x1b), \
  __MPI_LITERAL_DATA(0xb6, 0x9a, 0x7f, 0x0e), \
  __MPI_LITERAL_DATA(0x6d, 0x5f, 0x4d, 0xb1), \
  __MPI_LITERAL_DATA(0x89, 0xe4, 0x2c, 0x4f), \
  __MPI_LITERAL_DATA(0x50, 0xc9, 0xf2, 0x94), \
  __MPI_LITERAL_DATA(0x85, 0xf0, 0xb0, 0x69), \
  __MPI_LITERAL_DATA(0x09, 0x6a, 0xfe, 0x8f), \
  __MPI_LITERAL_DATA(0x6e, 0xf0, 0x6b, 0xbc), \
  __MPI_LITERAL_DATA(0x73, 0x72, 0xfb, 0x2f), \
  __MPI_LITERAL_DATA(0x4f, 0x08, 0x50, 0xbb), \
  __MPI_LITERAL_DATA(0x39, 0x57, 0x3e, 0x75), \
  __MPI_LITERAL_DATA(0x68, 0x7a, 0xcc, 0x55), \
  __MPI_LITERAL_DATA(0x4c, 0xf5, 0x60, 0x0a), \
  __MPI_LITERAL_DATA(0xeb, 0xdb, 0x01, 0xa4), \
  __MPI_LITERAL_DATA(0x8a, 0xa3, 0x40, 0x7d), \
  __MPI_LITERAL_DATA(0xbb, 0x92, 0x89, 0x71), \
  __MPI_LITERAL_DATA(0xb6, 0xf8, 0x7c, 0xbb), \
  __MPI_LITERAL_DATA(0xa0, 0x27, 0xf7, 0x04), \
  __MPI_LITERAL_DATA(0xe1, 0x75, 0x2a, 0xcb), \
  __MPI_LITERAL_DATA(0x7e, 0x07, 0x0d, 0x5a), \
  __MPI_LITERAL_DATA(0xbe, 0x99, 0x31, 0x56), \
  __MPI_LITERAL_DATA(0xed, 0xa2, 0xe8, 0x00), \
  __MPI_LITERAL_DATA(0x4c, 0x79, 0x65, 0x9f)

__MPI_LITERAL_BEGIN(static, PublicKey_N)
  PublicKey_N_LITERAL_DATA
__MPI_LITERAL_END(static, PublicKey_N, 2048)

#define PublicKey_E_LITERAL_DATA \
  __MPI_LITERAL_DATA(0x01, 0x00, 0x01, 0x00)

__MPI_LITERAL_BEGIN(static, PublicKey_E)
  PublicKey_E_LITERAL_DATA
__MPI_LITERAL_END(static, PublicKey_E, 17)

static const RSA_PUBLIC_KEY PublicKey_PublicKey = {
  {
#if MPI_MAX_BIT_COUNT >= 32
    { PublicKey_N_LITERAL_DATA },
#else
    (MPI_LIMB *)&PublicKey_N_Limbs,
#endif
    (2048+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    (2048+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    0,
    1 },
  {
#if MPI_MAX_BIT_COUNT >= 32
    { PublicKey_E_LITERAL_DATA },
#else
    (MPI_LIMB *)&PublicKey_E_Limbs,
#endif
    (17+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    (17+MPI_BITS_PER_LIMB-1) / MPI_BITS_PER_LIMB,
    0,
    1 },
};
```

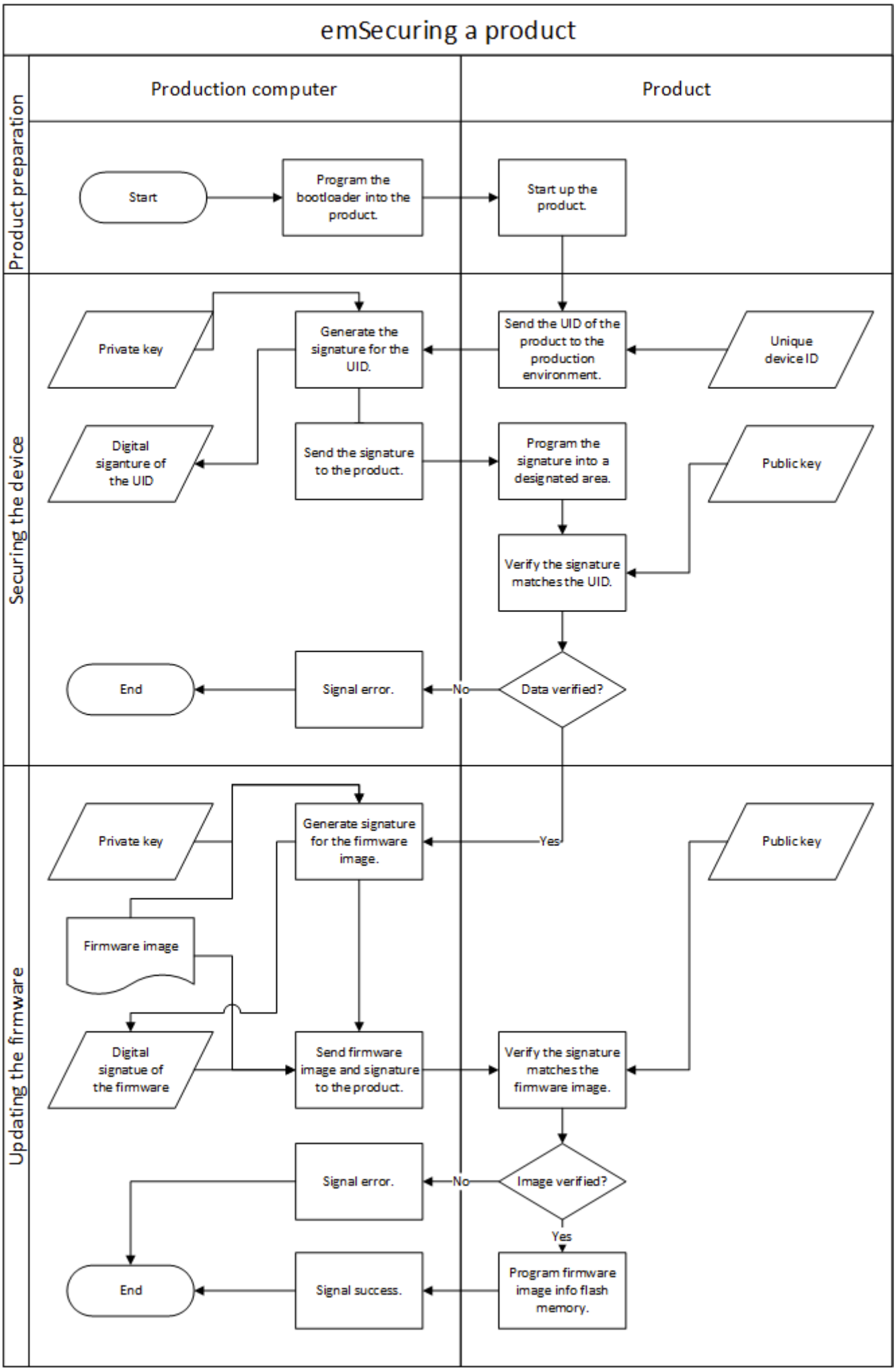

7.2 emSecuring a product - Flowchart

The following flowchart shows how to use emSecure in a product by securing a device and securing a firmware update.

The used key pairs are generated before and integrated in the applications.

The mentioned bootloader in the example starts when powering the product and includes following functions:

- Communication with the production appliction.
- Reading the device specific ID (UID) of the product’s device.
- Verifying the programmed digital signature if present and stop working when the data (the UID) cannot be verified.
- Receiving firmware images from the production application.
- Check and verification of a firmware image by its digital signature and programming it to flash when it is valid.
- Booting the firmware.



Index

