

**ANALOG
DEVICES**

Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources <http://www.analog.com/ee-notes> and <http://www.analog.com/processors> or
e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Secure Booting Guide for ADSP-BF70x Blackfin+ Processors

Contributed by Yi, Gabby and Molloy, Seth

Rev 1 – November 26, 2014

Introduction

This guide provides details associated with secure booting as well as accompanying examples. Using this guide one can successfully understand the different aspects of secure booting, and learn how to set up a processor and application for secure boot.

The ADSP-BF70x Blackfin+ Processors series (hereafter referred to as ADSP-BF707) provides a security solution for protecting IP and data. A secure operation environment, in which code can operate in, is also supported. An entire system infrastructure composed of a number of different items including cryptographic hardware accelerators, system protection unit (SPU), system memory protection unit (SMPU), one-time programmable (OTP) memory, secure debug and secure booting make up the complete security solution.

This document contains relevant information for creating a secure application and deploying it on an ADSP-BF707 secure processor. The example projects provided in the associated .ZIP file are referred to throughout this document to demonstrate the different features discussed.

This EE-Note describes the following items:

- The creation of a secure boot loader stream
- Usage of the `signtool.exe` host utility
- The ROM APIs to access OTP memory
- The ROM APIs for booting

Background Information

Terminology

Here are some commonly used terms in this guide:

- ECDSA - Elliptical Curve Digital Signature Algorithm
- BLp - Boot Loader plaintext, plaintext format
- BLx - Boot Loader without key, keyless format
- BLw - Boot Loader wrapped, wrapped format
- SBLS - Secure Boot Loader Stream
- SBH - Secure Boot Header
- SBCR - Secure Boot Confidentiality Root
- AES - Advanced Encryption Standard

More information on the above will be provided throughout this EE-Note. For further details regarding Security and Secure Booting, please refer to the *ADSP-BF70x Blackfin+ Processor Hardware Reference*^[1].

Secure Boot and Protection Types

The secure boot process provides means of integrating security in the processor boot sequence. A chain of trust is established within the system by ensuring the integrity and authenticity of the boot image. Confidentiality protection is also supported.

Secure boot increases protection against malicious, unsecured accesses to critical and confidential resources of the processor. The boot stream application code and data must be

digitally signed in order to build up a chain of trust in the system. This allows the processor to distinguish between authentic and trusted code from non-authentic and untrusted code.

Secure boot also provides confidentiality protection. The digitally signed boot image may be optionally encrypted. When loading an encrypted image the boot kernel will decrypt while loading, then authenticate, before any application code is executed.



Secure boot is an optional processor feature and is disabled by default. The feature is enabled via the Lock API and once enabled cannot be disabled. When security is enabled, developers are *not* dependent upon Analog Devices to provision the devices, sign code or provide security certificates.

The required tools for signing and encrypting the boot images are provided with CrossCore® Embedded Studio software development tools. More details on the usage of the tools for security purposes will be provided later.

Integrity and Authenticity Protection

Integrity and authenticity protection are based on the ECDSA algorithm using the secure hash SHA-2 (224-bit) algorithm. The hash algorithm ensures that only unmodified messages can be verified. Even a change in a single bit will produce a different hash digest resulting in failure.

ECDSA is a public key cryptosystem consisting of two keys, a key pair, one being a private key and the other a public key. The public key is stored in the processor's OTP memory so that the secure boot process can verify the authenticity of the signed boot image. Since the public and private keys are correlated to each other, only parties in possession of the private key are able to sign the images.

Confidentiality Protection

Confidentiality protection uses the Advanced Encryption Standard (AES) algorithm. Two variants are supported, wrapped and unwrapped.

The wrapped variant utilizes a 128-bit Key Encryption Key (KEK) stored on the processor's OTP memory to decrypt the 128-bit AES decryption key embedded in the secure header. The unwrapped variant stores the AES description key on the processor's OTP memory and utilizes it to decrypt the entire image.

Confidentiality protection allows code, data or other intellectual property to be stored securely on an external device prior to booting. Theft of the data will be useless without knowledge of the cipher key(s). Therefore, the privacy of the key stored on the device (whether AES or KEK) is paramount to the security of the system.



Disclosure of the key compromises security of the entire system.

Anti-Cloning Protection

Anti-Cloning protection is based on the confidentiality protection. If each processor uses a unique private key for the confidentiality protection then cloning between these devices can be prevented as the boot image will be incompatible with devices using a different private key for the decryption.

Anti-Rollback Protection

The secure boot process supports anti-rollback protection via a 32-bit counter in OTP memory. A value of 0x00000000 in OTP memory results in anti-rollback being disabled by default. If anti-rollback protection is required, then the user may set the Rollback ID when signing the boot image. Upon successful authentication of the boot image, the secure boot software will then update the counter in OTP memory, if the Rollback ID in the boot image is greater than the value currently stored in the OTP counter.

The Rollback ID stored in the secure boot image header is integrity protected preventing altering of the Rollback ID.

 Due to the fact that the Rollback ID is implemented in OTP memory, there are a number of restrictions in regards to its use. It is therefore recommended that only the OTP Program ROM API is used to set the counter. More information on this API is provided in a later section.

Secure Boot Image Types

Different secure boot image types are supported.

Plaintext Format (BLp)

This format provides *integrity* and *authenticity* protection of the boot image. Elliptical Curve Digital Signature Algorithm using 224-bit curves is used to provide this protection. The boot image is digitally signed using a private key. In order to authenticate the image, the corresponding public key is used by the boot kernel and must be pre-programmed into the OTP `public_key` field using the OTP Program API.

SBH	Boot Loader Stream
-----	--------------------

Wrapped Format (BLw)

This format provides the highest level of protection; *integrity*, *authenticity*, *confidentiality*, and *anti-cloning* protection. The image contains an ECDSA wrapped image encryption key (denoted by [K]) within the secure header. The image data is encrypted with the wrapped key, preventing cloning. An additional key is required to un-wrap the wrapped key in the header. This key must be pre-programmed into the OTP `pvt_128key` field using the OTP Program API.

SBH [K]	Encrypted Boot Loader Stream
---------	------------------------------

Keyless Format (BLx)

This format is similar to the BLw format except that the image does not contain the key at all.

This format provides anti-cloning protection only if the secure key is unique per device. The decryption key for the data must be pre-programmed into the OTP `pvt_128key` field using the OTP Program API.

SBH	Encrypted Boot Loader Stream
-----	------------------------------

Secure Boot Image Format

Secure Boot images provide *authenticity* and *integrity* during the boot process. A secure boot image is comprised of a secure boot header and an optionally encrypted loader stream.

Signed images consist of the following sections to comprise a complete secure boot image:

- Secure Boot Header
- Image Attributes
- Image Section

Figure 1 shows that the image attributes are encapsulated within the secure boot header. The image attributes are integrity protected along with the image section. The image section contains a standard boot loader stream with the caveat that some block types are not allowed as described in the [Unsupported Boot Stream Blocks](#) section.

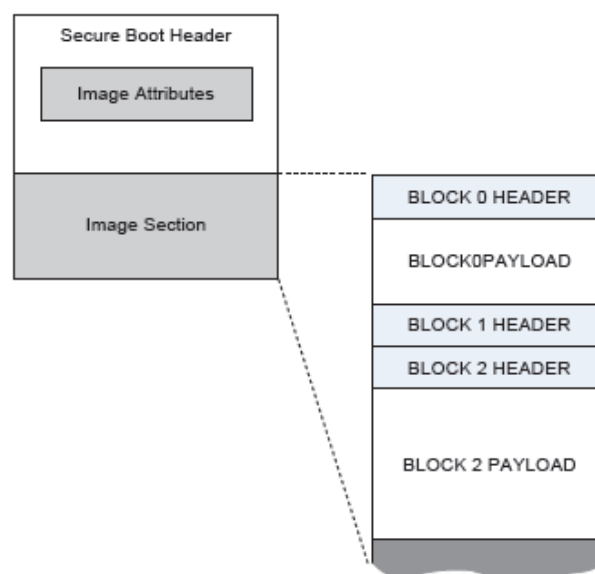


Figure 1. Secure Boot Image

Secure Boot Header

Table 1 shows the Secure Boot Header format.

Bytes	Name	Description	Values		
			Keyless Format (BLx)	Wrapped Format (BLw)	Plaintext Format (BLp)
3:0	Type	Format and version of the image. Upper 24 bits is the image format and lower 8 bits is the image version	0x424c7801	0x424c7701	0x424c7001
67:4	Signature	The ECDSA signature of the image	Two 256-bit numbers		
91:68	Key	Confidentiality (only applicable for certain formats)	Reserved	192-bit AES-WRAP data holding a 128-bit AES key	Reserved
107:92	IV	Initialization Vector (only applicable for certain formats)	Reserved	16-byte IV generated during signing process	
111:107	Length	The length of the image section in bytes	Maximum supported byte count 0x10000000 bytes		
171:112	Attributes	Image Attributes	Support for up to 8 image attributes		
175:172	Reserved	Reserved	Reserved		

Table 1. Secure Boot Header Format

Developing a Secure Boot System

The key steps for developing a system that uses secure booting are:

1. Generate required keys
2. Develop application and generate the boot stream file
3. Convert loader stream file to a secure boot stream file
4. Program the boot stream into the storage device (i.e., external SPI flash memory)
5. Program keys into OTP memory
6. Lock device to enable security

The following sections describe each of these steps along with how they are accomplished both manually and with the accompanied examples.

The supplied projects have been developed and tested with the ADSP-BF707 EZ-KIT® evaluation system^[4].

Keys

A minimum set of keys are required to enable a secure system. The required keys for secure booting are dependent on the security features chosen. Refer to [Secure Boot Image Types](#) section for more details on the supported types.

Public/Private Key Pairs for Digital Signatures

Secure booting relies on digital signatures to ensure that the boot stream has not been modified and that in fact it is from the trusted developer. The developer must first generate a key pair consisting of a public and a private key.

It is used to sign and create the digital signature for the boot stream. The private key is used on the host computer and should be kept secret. The public key is programmed into the processor's OTP memory and is used by the boot kernel during secure boot to verify the digital signature.

Signtool for Key Generation

A host utility, `signtool.exe`, provided with CrossCore Embedded Studio (CCES) tools, help generate keys and create digital signatures (default installation path):

```
C:\Analog Devices\CrossCore Embedded Studio 1.1.0\signtool.exe
```

Key Generation for Signing

Boot stream files are signed using the private key from a 224-bit Elliptic Curve Digital Signature Algorithm (ECDSA) key pair, stored in DER (Distinguished Encoding Rules) format. The `signtool.exe` utility's `genkeypair` command can be used to create such a key pair. Table 2 describes the switches available for the `genkeypair` command.

Switch	Description
-algo ecDSA224	Selects the algorithm used to create the keypair. For Analog Devices processors, ecDSA224 is required
-outfile file.der	Directs signtool to create the keypair in the file file.der

Table 2. Genkeypair Command Switches

```
signtool genkeypair -algo ecDSA224 \
-outfile keychain.der
```

Listing 1. Usage of Signtool.exe to Generate Key Pair



Once created, ensure the key pair is kept safe and secret, as the private key therein is essential for authenticity.

Extracting Public Key

When the key pair is generated, the output file contains both the public key and the private key.

The public key will be needed so it can be programmed into the processor's OTP memory for the boot kernel to use during secure boot.

Therefore the public key needs to be extracted from the key pair. Use `getkey` command from `signtool.exe` utility to extract the public key.

The switches relevant to the `getkey` command are listed in Table 3.

Switch	Description
-type BLKey BLKeyC	Determine the output format for the public key.
-key keyfile.der	Specify the file containing the keypair, from which to extract the public key.
-outfile pubkey	Specify the name of the file to which the public key should be written.

Table 3. Getkey Command Switches

The key can be extracted to binary form or to a text file that can be used as a C header file. The following is an example of extracting the public key in binary form:

```
signtool getkey -type BLKey -key \
keychain.der -outfile pubkey.bin
```

Listing 2. Usage of Signtool.exe to Extract Public Key from Key Pair into Binary File

AES-128 Confidentiality Key and Key Encryption Key (KEK)

If confidentiality is required, the boot stream can also be encrypted using the AES-128 cipher in CBC (Cipher Block Chaining) mode. A single key is used to both encrypt and decrypt the message. The `signtool.exe` utility will use the specified key on the host computer to encrypt the boot stream, and the boot kernel will use the same key, pre-programmed into the processor's OTP memory, to decrypt the boot stream as part of the secure boot process.

The alternative is to wrap this key and to send it along with the boot loader stream in the header.

In order to guarantee its security, it needs to be wrapped using another key (KEK). This will be the one stored in the OTP memory and used to un-wrap the confidentiality key.

Confidentiality Key and KEK Generation

The AES-128 confidentiality and the KEK keys are both 128-bit strings. The `signtool.exe` utility does not specifically generate any keys for AES-128. A user can use any means to generate sufficiently random strings of 128 bits to use as an AES-128 key.



Ensure that these keys are kept secret!

Secure Debug Key

The secure debug key is another 128-bit key that allows the user to gain access to the part, specifically to the secure resources, when the part is locked (security enabled). Otherwise, the debugger will only have access to non-secure resources, such as non-secure portions of memory. It is stored in the processor's OTP memory to be used as reference to compare against, when the debugger attempts access.

Secure Debug Key Generation

Like the AES-128 key, no specific provision is made. A sufficiently random number is recommended for this 128-bit key.

Generating Secure Boot Stream

Creating a Boot Stream

In order for the boot kernel to consume the application, it must be packaged as a loader file. The application project can automatically produce a loader file from the project options in the CCES tools.

From the `Properties` dialog box, change the project's output type (`Build Artifact` type) as loader file (`.LDR`):

1. Choose `C/C++ Build/Settings`. The `Tool Settings` page of the `Settings` dialog box appears.
2. Click the `Build Artifact` tab. In `Artifact` type, choose `Loader File` from the drop-down list.
3. In `Artifact` name, the name is `${ProjName}`. In `Artifact` extension, the extension is `ldr`.
4. (Optional) In `Output prefix`, enter a string to prepend to the output.
5. Click `Apply` to save the changes to the project.
6. Click `OK` to close the dialog box.

Further details on creating a boot loader stream can be found in the *CrossCore Embedded Studio Loader and Utilities Manual*^[3].



The loader utility output format *must* be binary in order to be compatible with the `signtool.exe` utility.

Unsupported Boot Stream Blocks

In order to ensure the security of the processor the following block types are not supported in a secure boot image. Should the boot kernel come across one of these block types, the boot process will terminate.

Init Block

Init blocks require a call to user application code prior to the authentication of the boot image, and therefore cannot be supported. If customizations or optimizations are necessary to improve the load performance, it's recommended to use a second stage loader style implementation. The first application will contain only the custom code, and then issue a call using the boot routine to boot using the desired device.

Callback Block

Callback blocks require a call to a user defined address prior to the authentication of the boot image, and therefore cannot be supported.

Signing & Optionally Encrypting the Boot Image

The generation of the secure boot stream is just a conversion of a normal boot stream. It involves the use of the private key to create a digital signature, which is stored in a secure header as part of the secure boot stream. This is done for all three secure boot types: Plaintext (BLp), Wrapped (BLw), and Keyless (BLx).

If the BLw or BLx formats are chosen, the boot stream will be encrypted using the 128-bit confidentiality key along with signing the boot

image. For BLw, the confidentiality key used will be wrapped with the Key Encryption Key (KEK) and stored in the secure header of the boot stream.

Signtool for Generating the Secure Boot Stream

The `signtool.exe` utility, via the `sign` command, is used for signing and encrypting the boot stream image. The `sign` command supports various key specific switches (Table 4).

Switch	Description
-type BLp BLw BLx	Indicates which Secure Boot format is used.
-prikey file.der	Identifies the file containing the ECDSA-224 keypair for signing (BLp, BLx, and BLw).
-enckey encfile.key	Identifies the 16-character file containing the AES-128 key string to be used for encrypting the file's content. This switch is required for BLx. It is optional for BLw. If omitted, signtool.exe generates a random key for encryption.
-wrapkey wrapfile.key	Identifies the 16-character file containing the AES-128 key string to be used for wrapping the encryption key (BLw). Must be used with -enckey switch.
-infile bootstream.bin	Specifies the binary file containing the boot stream to be signed (and encrypted).
-outfile secure.bin	Specifies the name of the file to create, containing the signed/encrypted version of bootstream.bin.

Table 4. Sign Command Switches

The following is an example command line usage of the `signtool` utility to sign a boot loader stream, `boot.bin`, using the private key stored in the key pair file `keychain.der`.

```
signtool sign -type BLp -prikey \
  keychain.der -infile boot.bin \
  -outfile secboot.bin
```

Listing 3. Usage of `signtool.exe` to sign a boot stream for integrity and authenticity protection

The final output `secboot.bin` is integrity, as well as authenticity protected. The original boot loader stream is still in plaintext format, meaning it's visible. If the user wants confidentiality protection as well, the `signtool` utility can be invoked in the following two ways:

```
signtool sign -type BLx -prikey \
  keychain.der -enckey aes.bin \
  -infile boot.bin -outfile secboot.bin
```

Listing 4. Usage of `signtool.exe` to sign and encrypt a boot stream for integrity, authenticity and confidentiality protection.

```
signtool sign -type BLw -prikey \
  keychain.der -enckey aes.bin \
  -wrapkey wkey.bin -infile \
  boot.bin -outfile secboot.bin
```

Listing 5. Usage of `signtool.exe` to sign and encrypt a boot stream for integrity, authenticity and confidentiality protection with wrapped encryption key.

The only difference between these two methods of adding confidentiality protection is whether the cipher key is sent along with the secure boot stream (BLw) or not (BLx).

Secure Boot Image Attributes

When the secure boot stream is generated, secure boot image attributes that form part of the secure boot header can also be specified. These attributes are used to provide additional information in regards to the content of the secure boot image.

All image attributes are integrity protected using the same algorithm as the image section, so when the image authentication process completes, its attributes are known to be trusted.

Attributes are specified as type value pairs with both the type and value being a 32-bit value. The boot code supports the following image attributes (Table 5).

ID	Name	Description	Values	
0x00000000	Unused	Unused attribute	Value must be 0x00000000.	
0x00000001	Version	Version of the attribute format	Value must be 0x00000000.	
0x00000002	Rollback ID	Current value of the rollback counter	0x00000000 – 0xFFFFFFFF.	
0x80000000	Secure Scrub	Optionally scrub stack and workspace areas after loading image	0x1	Do not scrub the stack and workspace used for secure boot.
			Other	The stack and workspace will be scrubbed after booting.

Table 5. Secure Boot Image Attributes

Programming Secure Boot Stream into Storage Device

Once the secure boot stream is generated, it can be stored in an external flash memory device or any other storage device that the processor can boot from^[2].

CrossCore Embedded Studio provides a Command Line Device Programmer (CLDP), which can be used to program the external memory device on the target board. For further details on the CLDP, refer to the *CrossCore Embedded Studio Loader and Utilities Manual*^[3].

Programming Keys into OTP Memory

All keys used by the boot kernel for authentication and decryption must be programmed into the processor's OTP memory. Programming the OTP memory can be accomplished in two ways. One way is to use the CLDP utility, which also supports programming the processor's OTP memory.

Another method is to run an application on the processor, which calls the OTP read/write API to functions stored in ROM.



It is recommended that all OTP accesses are done through the OTP API functions provided in ROM. To use the API functions, include the `cdefBF70x_rom.h` header file in the source code, which can be found in (default installation path): `C:\Analog Devices\CrossCore Embedded Studio 1.1.0\Blackfin\include`

API for OTP Programming

Here is an example for one of the functions that can be used to program the data manually.

```
bool res = adi_rom_otp_pgm_data(data,
offset, size);
```

Listing 6. Example Use of OTP Programming API to Program a Certain Location

The recommended API function to use is shown in Listing 7.

```
bool res = adi_rom_otp_pgm(data);
```

Listing 7. Use of the adi_rom_otp_pgm() ROM Function

Using this verified API function, the details of key location and programming multiple locations for double redundancy are fully hidden.

```
/*
 * program_public_key
 * Example of programming the public
 * key, used
 * for authentication
 */
bool program_public_key() {
    uint32_t key[ROM_OTP_SZ_public_key0]
    = {
        0xdebb924f, 0x430af76a,
        0xd9e8dac8, 0x4631cbf5,
        0xce2b4164, 0xeb9605f6,
        0xd9cc6fa3, 0xc6f273a3,
        0x4c138c37, 0x05192834,
        0x8762ee55, 0x907b5072,
        0x749d7487, 0xff53f1c3, 0, 0 };
    otp_data data = {0};
    data.public_key0 = &key;

    if(!adi_rom_otp_pgm(&data) ) {
        return false;
    }
    return true;
}
```

Listing 8. Example Code to Program the Public Key into OTP Memory

In Listing 8, the OTP data structure, `otp_data`, contains the element `public_key0`, which is loaded with the address of the key data buffer, `key`, and then programmed into OTP via `adi_rom_otp_pgm()` function. Here, even though 16 32-bit words are written, only 14 are needed for ECDSA 224-bit used in secure booting.

This same API function can be used to program other keys at the same time. In order to do so, prepare `uint32_t` buffers for the secure debug and AES-128 private keys, and load the

addresses of these buffers into the OTP data structure elements `secure_emu_key` and `pvt_128key1`, respectively. Once the OTP data structure is finalized, `adi_rom_otp_pgm()` can be called like in Listing 8.

Listing 9 shows the struct type definition that the `adi_rom_otp_pgm()` API function uses. Refer to the ROM header file for the exact definition.

```
typedef struct {
    uint32_t (*huk)[8];
    uint32_t (*dtcp_ecc_key)[40];
    uint32_t (*dtcp_const_key)[19];
    uint32_t (*dtcp_dev_key)[32];
    uint32_t (*pvt_128key1)[4];
    uint32_t (*pvt_128key2)[4];
    uint32_t (*pvt_128key3)[4];
    uint32_t (*pvt_128key4)[4];
    uint32_t (*pvt_192key1)[6];
    uint32_t (*pvt_192key2)[6];
    uint32_t (*public_key0)[16];
    uint32_t (*public_key1)[16];
    uint32_t (*ek)[8];
    uint32_t (*secure_emu_key)[4];
    uint32_t bootModeDisable;
    uint32_t (*boot_info)[16];
    uint32_t (*gp0)[16];
    uint32_t antiroll_nv_cntr;
    uint32_t stageID;
    uint32_t (*preboot_ddr_cfg)[11];
} otp_data;
```

Listing 9. C Structure Representing the Elements Stored in OTP

All non-zero data in the array provided are programmed into OTP memory. Programming is done according to the OTP controller requirements. Only double redundant mode is supported. For each 32-bit data, 4 locations will be programmed as required by the OTP controller^[1].

False is returned if an error code is observed in the `OTPC_STAT` or `OTPC_PMC_STAT` registers, or the `OTPC_PMC_STAT` indicates that the controller is not in an idle state. If an error occurs, refer to those registers for more detailed information.

There also exists corresponding API to ROM functions allowing the user to read contents in OTP memory.

```
bool res =
adi_rom_otp_get(otpcmd_info, data);
```

Listing 10. Use of the `adi_rom_otp_get_data()` ROM Function

The data specified by the `OTPCMD` enum parameter is fetched from OTP memory and placed in the location specified by `data`. This API function uses `adi_rom_otp_get_data`.

The `OTPCMD` enum contains entries for each field defined in OTP memory. For the most current list, refer to the OTP header file.

Locking the Part

In order to lock a processor and enable security, a particular location in OTP memory must be written to. This should be done using the ROM API function Lock API.

```
bool res = adi_rom_lock();
```

Listing 11. OTP API to Lock the Part and Enable Security

Calling this function locks the device, making it secure. Once locked, the `OTPC_SECU_STATE` register indicates that the part is locked, and access is limited to the OTP memory. Other security features are also enabled.



At this point only secure boot streams are allowed, which will be verified through cryptographic means. Standard (non-secure) boot streams will be rejected. In the event that either verification fails on a secure boot stream or a non-secure boot stream is used, the boot kernel will set the fault signal pin active and set the core to idle, not finishing the boot process.

Accompanying Examples

This EE-Note is accompanied by example projects that demonstrate the different stages of developing a Secure Boot system on an ADSP-BF707 EZ-KIT board.

The following sections will describe the use of the various example projects contained in the associated `.ZIP` file.

keygen

Using keygen Example Project for Generating Keys

The `keygen` project provides an example for creating all the required keys for authentication and encryption. Randomly generated keys can be used, or custom keys can be provided.



Python is required for `keygen` project to generate all of the needed collateral. Update the path to your python executable in the supplied `Makefile`.

For development and testing of the provided example projects, ActivePython® was used^[5].

The `keygen` project is a simple CCES `makefile` example. By building the project, the `makefile` will generate all the required keys and associated header files using the `signtool` utility, as well as some python scripts. All other projects use the keys and header files generated by `keygen`.



Build the `keygen` project only once and do not delete the generated keys! The clean function is disabled to protect the keys. Once the generated keys are programmed into OTP memory, we do not want new keys generated, overridden the existing ones.

keygen: Random Keys

`genRandKey.py` is used to create random 128-bit keys for the encryption, emulation, and wrapper keys. `bin2inc.py` is then used to create a C

array in a header file so that the key can be easily programmed into OTP memory.

```
make random_key.bin random_key.h
```

Listing 12. Using Keygen Project to Generate a Random Key

keygen: Authentication Key

The `signtool.exe` utility is used to generate a key pair, and to extract the authentication key into `secure_boot_public_key.h`. This header file is used to program the authentication key directly to OTP memory.

keygen: Custom Keys

If custom keys are used instead of generating them using the `keygen` project, keys must be placed in the `keygen` folder with the correct names. The filenames can be found at the top of the `Makefile` and are:

- `wrapper_key.bin`
- `auth_key.bin` (keypair)
- `encrypt_key.bin` (confidentiality key)
- `jtag_key.bin` (secure debug key)

After placing the custom keys in the directory, the project must be built again. Any missing keys will be generated, and all other collateral will be generated based on the provided keys.

Program_OTP

Provides examples to setup OTP memory for the various secure boot image types based on the previously generated keys by `keygen`.

It's strongly recommended to use this example to program all keys for the desired image type.

Using the Program_OTP Example Project

The `Program_OTP` example provides a CCES executable project that programs the relevant keys from the `keygen` project.

`Program_OTP` example project provides three different functions to program the AES-128

authentication key, the encryption key, and the emulation key.

The application also contains an additional function that locks the part using the OTP API.



Comment out the call to `lock_part()`, if enabling security is not yet desired.

In addition to the functions to program each individual key, three setup functions are also provided: `setupBLp`, `setupBlw`, `setupBLx`. These functions will program all the required keys into the appropriate OTP memory fields for the associated image type.



Do not run all three setup functions. Choose the one depending on which secure boot image type is used and comment out the others.

LED_Blink

A standard application signed for all image types using the generated keys. This application should be programmed into external SPI Flash memory (SPI2).

Using the LED_Blink Signing Example Project

`LED_Blink` is a simple CCES project to blink the LEDs on the ADSP-BF707 EZ-KIT evaluation board^[4]. The project is used to show how the normal non-secure boot stream can be converted into a secure boot stream file.

The `LED_Blink` project utilizes the `keygen` `makefile` features to sign the resulting boot stream in all of the available image types by invoking `signtool.exe`.

A post-build step is used to accomplish this. The `Build Steps` tab is found in the `C/C++ Build/Settings` menu of the properties dialog found from the `Project/Properties` option.

The post-build setting refers to the `makefile` in the `keygen` project, provides the location of the keys, and specifies which image types to build. The image types are specified at the end of the

command such as `${ProjName}.blwldr`. This directs the makefile to build a BLW image type using the keys it previously generated.

```
Make -f
"${workspace_loc:/keygen/Makefile}"
KEYDIR="${workspace_loc:/keygen}"
${ProjName}.blpldr ${ProjName}.blxldr
${ProjName}.blwldr
```

Listing 13. Post-build Command in Keygen Project to Create Secure Boot Streams

By default, this project will build all three secure boot streams types, producing `LED_Blink.blpldr`, `LED_Blink.blxldr`, and `LED_Blink.blwldr`.

The example is otherwise a standard CCES project containing the desired application.

SecureBoot_OpenPart

This section provides more details on how to set up secure booting on an open, non-locked part for development and testing purposes.

The ROM code provides a mechanism to boot a secure boot stream on an open part without writing any keys into OTP memory. This can be very useful in validating the generated keys and application stream before writing to OTP memory.

Secure booting is accomplished by loading an application into memory using the emulator, which utilizes the ROM API function `adi_rom_Boot` in conjunction with a hook function, configuring the kernel for secure boot, and initiating the boot process.

Configuring the Kernel

In order to configure the kernel a hook function is used during the initial boot sequence to alter the internal configurations at the appropriate time. For more information on the hook function provision in the ROM API, refer to the Boot chapter in the hardware reference manual^[1].

When the hook function is called, a structure is passed, allowing access to the kernel's configurations. There are three data of interest: `bootType`, `keyType`, and `pKey`.

The data `bootType` defines the type of boot, by setting this to `ADI_ROM_SECURE_BOOT`, the kernel will proceed assuming a secure boot. In a normal boot process this is set when the part is locked.

By setting `keyType` to `ADI_ROM_CUSTOM_SECURITY`, reading keys from OTP memory is essentially disabled. This allows the keys to be explicitly set in our application. The data `pKey` defines the key to be used for authentication.

Booting Using the ROM API

The ROM API provides the ability to initiate a boot sequence at any point during execution. As the API is called at runtime, it can provide an additional level of flexibility that is not provided by the default boot options given by the defined boot modes.

In this case, the most important customization is the ability to pass a custom hook function. Called strategically during the boot process, customization of many kernel operations is possible. The hook function is called with a parameter to indicate at what point of the boot process the call is being made.

To configure the kernel for secure boot, the hook function must perform the configuration after default initialization is completed.

Application Example

The following code example provides an application that can be loaded and executed using the emulator. The application's main function calls the ROM API initializing an SPI Master Boot and `BCODE` 0. The `ConfigureForSecureBoot` function is included below illustrating the key kernel configurations.

```

/**
 * ConfigureForSecureBoot
 * @param pBootConfig boot configuration structure
 * @param cause the reason the hook was called
 * @brief The hook function is called twice by the ROM after configuration and
after initialization
 * Here we will setup the boot Kernel to do a secure boot.
 */

int32_t ConfigureForSecureBoot(ADI_ROM_BOOT_CONFIG * pBootConfig,
ROM_HOOK_CALL_CAUSE cause) {
if (cause == ROM_HOOK_CALL_INIT_COMPLETE) {
    /* This bit is only executed after the boot modes init function has be run */

    pBootConfig->pSource+=16; /* kernel anomaly workaround */

    /* 1. Set the boot type to secure */
    pBootConfig->bootType=ADI_ROM_SECURE_BOOT;

    /* 2. Set they key type to custom (this disables reading the key from OTP) */
    pBootConfig->keyType = ADI_ROM_CUSTOM_SECURITY;

    /* 3. load the key to be used, this key is normally read from OTP */
    uint8_t * pKey = (uint8_t *)&pBootConfig->publicKey;

    for(uint8_t i=0; i<56; i++) {
        pKey[i] = publickey[i];
    }
} else {
    /* This section is called after the config routine is finished */

    setupMemoryMappedSPI(pBootConfig);

    /* change dBootCommand to use SPI XIP mode */
    pBootConfig->dBootCommand &= (!BITM_ROM_BCMD_DEVICE);
    pBootConfig->dBootCommand |= ENUM_ROM_BCMD_DEVICE_SPIXIP;
}

return 0;
}

```

Listing 14. Example Code to Set Up for Secure Boot Using the ROM API

Understanding the SecureBoot_OpenPart Example Project

The SecureBoot_OpenPart project illustrates secure boot on an open part. By utilizing various ROM APIs it is possible to perform limited testing of the secure boot without locking the part, and without writing any keys into OTP.

The example provided utilizes SPI Master boot mode. The generated secure loader file must be programmed into the base of SPI flash memory.

ConfigureForSecureBoot Hook Function

The ConfigureForSecureBoot(...) hook function illustrates how to configure the ROM

for secure boot and provides different functions to utilize the different secure boot types. There are two basic operations: changing the boot and key type, and loading the keys into memory.

To modify the boot type and the key type the associated fields are modified in the `pBootConfig` structure. This structure contains all the settings extracted by the ROM from various sources, such as OTP memory settings, security status, Reset Control Unit (RCU) registers, etc. The `bootType` can be modified to `ADI_ROM_SECURE`, and the `keyType` can be modified to `ADI_ROM_CUSTOM_SECURITY`. This causes the ROM to operate in a secure mode, and disables reading any key from OTP memory.

Three functions are provided as helper functions to the configuration hook. Each function will use the keys generated by the `keygen` project to configure the processor for the associated secure boot image type. Only one can be used at a time. Uncomment the desired image type. This must match the type loaded into SPI flash memory.

Boot Routine

The main application code simply calls the boot routine providing the `ConfigureForSecureBoot(...)` function as a hook routine. The `LED_Blink` application will validate that it has been loaded by blinking the corresponding LEDs.

Secure Debug Access

The TAPC controller provides a means of restricting access to secure resources of the processor. Secure access via the debug port is protected via a 128-bit security key that must match a key that has been pre-loaded into the processor's OTP memory.

In order to gain access to a locked processor the TAPC must allow access to the part. The TAPC will only allow access to the part if it is provided with a matching key to the data loaded into its `TAPC_USERKEYn` registers.

The key is set using the OTP Program API to program the `secure_emu_key` field in OTP memory. This key is read and loaded by the boot kernel in the following sequence:

1. Bits [31:0] of the key in OTP memory are stored to `TAPC_USERKEY0` bits [31:0]
2. Bits [63:32] of the key in OTP memory are stored to `TAPC_USERKEY1` bits [31:0]
3. Bits [95:64] of the key in OTP memory are stored to `TAPC_USERKEY2` bits [31:0]
4. Bits [127:96] of the key in OTP memory are stored to `TAPC_USERKEY3` bits [31:0]

Once the boot kernel has loaded the user key, a test key can be provided to the TAPC via the JTAG interface.

A key failure indication can be detected via the `TAPC_USERKEY_STAT` register. The boot code does not check the key status, nor does it enable any associated interrupts to signal key failure. The boot code will continue to boot upon a key failure in a secure manner. The key failure status will remain intact so that the application loaded can check for a failed challenge on the debug port.

The boot code can be configured to bypass the loading of the key during the boot sequence by setting the value of the `secure_emu_key` in the OTP to all ones. In this scenario the only means to gain access to the secure resources via the debug port is to load an alternate key via the application. The application will need to write a key into the `TAPC_USERKEYn` registers. The alternative key should reside in a secure region of memory at all times or if sent remotely should be transmitted over a secure connection.

Using Secure Debug Key with CrossCore Embedded Studio

A key can be passed from the debugger via a custom board support XML file or in one of the `ADSP-BF70x-resets.xml` files found in `<CCES install path>\System\ArchDef` directory.

The benefit of using a custom board support XML file is the ability to use different keys for different boards. A different custom board support XML file would be created for every board and every CCES debug session.

Key Format

When a key is created using `keygen`, a 16-byte array is provided in `jtag_key.h`. For example, if the key is defined as:

```
const static uint8_t jtag_key[] = {
    0x00,
    0x11,
    0x22,
    0x33,
    0x44,
    0x55,
    0x66,
    0x77,
    0x88,
    0x99,
    0xaa,
    0xbb,
    0xcc,
    0xdd,
    0xee,
    0xff} ;
```

Then, the XML file should contain the following register settings to pass in the correct key to gain entry into the secure system:

```
<register name="userkey0" core="Common"
reset-value="0x33221100" />
<register name="userkey1" core="Common"
reset-value="0x77665544" />
<register name="userkey2" core="Common"
reset-value="0xbbaa9988" />
<register name="userkey3" core="Common"
reset-value="0xffeeddcc" />
```

Error and Failures

Any error encountered while processing a secure boot image will result in the ROM jumping to the Error Handler. This includes decryption failures, authentication failures, and configuration errors. Custom error handlers are not supported for secure booting. The default boot kernel error handler will trigger the fault signal and go to idle state.

Conclusion

Secure Boot allows the user to safeguard their sensitive IP and data stored on an external storage device. The system uses cryptographic functions to decrypt and verify the boot stream prior to loading it as a useable format into the processor.

This EE-Note reviewed all the steps and requirements necessary to set up a Secure Boot system, as well as explained the accompanied examples that setup the secure system.

References

- [1] *ADSP-BF70x Blackfin+ Processor Hardware Reference*. Rev 0.2, May 2014. Analog Devices, Inc.
- [2] *ADSP-BF70x Blackfin+ Core Embedded Processor Preliminary Data Sheet*. Rev PrC, May 2014. Analog Devices, Inc.
- [3] *CrossCore Embedded Studio Loader and Utilities Manual*. Rev 1.3, May 2014. Analog Devices, Inc.
- [4] *ADSP-BF707 EZ-KIT Lite Evaluation System Manual*. Rev 1.1, October 2014. Analog Devices Inc.
- [5] *ActivePython* (<http://www.activestate.com/activepython>). Version 2.7, ActiveState.

Document History

Revision	Description
<i>Rev 1 – November 26, 2014 by G. Yi and S. Molloy</i>	Initial Release