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 Blackfin+® and SHARC+® Processors

Contributed by G. Yi and S. Molloy *Rev 2 – October 19, 2016*

Introduction

The ADSP-BF70x Blackfin+® and ADSP-SC58x/ADSP-2158x SHARC+® processors feature a security solution for protecting IP and data. A secure operation environment is also provided for code execution, and an entire system infrastructure is available, composed of a number of functional blocks, including cryptographic hardware accelerators, a system protection unit (SPU), a system memory protection unit (SMPU), a one-time programmable memory (OTP), secure debug, and secure booting.

This EE-note provides details and example code associated with secure booting. Using this guide, one can successfully understand the different aspects of secure booting and learn how to set up the processor and application for secure boot. This document contains relevant information for creating a secure application and deploying it on a secure processor. Several examples are referred to throughout this document that demonstrate the different features discussed.

This document describes:

- secure boot stream creation in the CROSSCORE® Embedded Studio (CCES) development tools,
- usage of the CCES `signtool` host utility to generate keys and create digital signatures,
- the processor ROM APIs to access OTP memory for key storage, and
- the processor ROM APIs for booting the device.

Terminology

The following is a list of some commonly used terms in this EE-Note:

- 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

Further explanation will be provided in the sections that follow.

Secure Boot and Protection Types

The secure boot process provides a means of integrating security into 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 processor resources. 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 as well. 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 enabled via the Lock API. Once enabled, it cannot be disabled, and it is disabled by default. When 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 the processor's development tools.

Integrity and Authenticity Protection

Integrity/authenticity protection is based on the ECDSA algorithm using the secure hash SHA-2 224-bit algorithm along with 224-bit curves. The hash algorithm ensures that only unmodified messages can be verified. Even a change to 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 key and private key 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, supporting wrapped and unwrapped variants. The wrapped variant utilizes a 128-bit Key Encryption Key (KEK) stored on the processor to decrypt the 128-bit AES decryption key embedded in the secure header. The unwrapped variant stores the AES description key on the processor and utilizes it to decrypt the entire image.

Confidentiality protection allows code, data or other intellectual property to be stored on an external device securely 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 this 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 the OTP memory. A value of `0x00000000` in the OTP 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, thus preventing altering of the Rollback ID.

Because the Rollback ID is implemented in OTP memory, there are a number of restrictions regarding its use. It is therefore recommended that only the OTP Program ROM API be used to set the counter.

Secure Boot Image Types

Several different secure boot image types are supported.

Plaintext Format (BLp)

This format provides *integrity* and *authenticity* protection of the boot image. ECDSA 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 unwrap 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 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 protection 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 actually 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 Unsupported Boot Stream Blocks section.

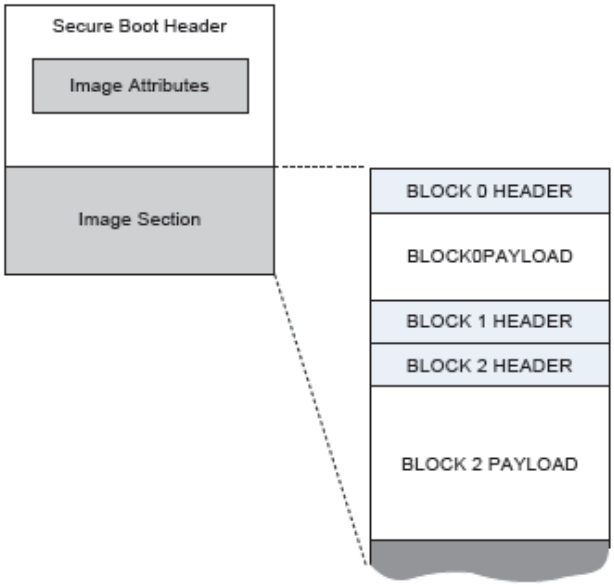

Figure 1. Secure Boot Image

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	
111:108	Length	The length of the image section in bytes	Maximum supported byte count 0x10000000 bytes		
175:112	Attributes	Image Attributes	Support for up to 8 image attributes		
179:176	Reserved	Reserved	Reserved		

Table 1. Secure Boot Header Format

Developing a Secure Boot System Overview

The following are the key steps in developing a system that uses secure booting:

1. Generate the keys.
2. Develop the application and generate the boot stream file.
3. Convert the loader stream file to a secure boot stream file.
4. Program the boot stream into the storage device.
5. Program the keys into OTP memory.
6. Lock the device to enable security.

The following sections explain each of these steps and how they are accomplished, both manually and when using the accompanying examples.

Keys

A minimum set of keys are required to have a secure system. The keys required for secure booting depend on the security features chosen. Refer to the various image types available for details regarding which keys are required for each image type.

Public/Private Key Pairs for Digital Signatures

Secure booting relies on digital signatures to ensure that the boot stream hasn't been modified and to ensure that it is, in fact, from the trusted developer. The developer must first generate a key pair consisting of a public and private key. The private key is used to sign and create the digital signature for the boot stream. It is used on the host computer and should be kept secret. The public key is programmed into the OTP memory and is used by the boot kernel during secure boot to verify the digital signature.

Signtool for Key Generation

The CCES development tools provide a host utility, `signtool`, to help generate keys and create digital signatures.

Key Generation for Signing

Boot stream files are signed using the private key from a 224-bit ECDSA key pair, stored in DER format. The `signtool` utility's `genkeypair` command can be used to create such a key pair, as shown in Listing 1.

```
signtool genkeypair -algo ecdsa224 -outfile keychain.der
```

Listing 1. Generating 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. Switches for genkeypair Command in signtool Utility

Once the key pair is created, keep it secret, as the private key therein is essential for authenticity.

Extracting the 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 that it can be programmed into the OTP memory for the boot kernel to use during secure boot; therefore the public key needs to be extracted from the key pair.

Use the `getkey` command in the `signtool` utility to extract the public key so that it can be stored in the processor's OTP memory, where it can be used during secure boot.

The relevant switches associated with 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. Switches for getkey Command in signtool Utility

The key can be extracted to binary form or to a text file that can be used as a C header. Listing 2 is an example of extracting the public key in binary form.

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

Listing 2. Extracting the Public Key from the Key Pair into a 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 mode. A single key is used to both encrypt and decrypt the message. The `signtool` utility will use the specified key on the host computer to encrypt the boot stream, and the boot kernel will use the same key, which will have been programmed into the OTP memory, to decrypt the boot stream as part of the secure boot process.

The alternative means is to wrap this key 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, the KEK. This will be the one stored in the OTP memory and used to unwrap the confidentiality key.

Confidentiality Key and KEK Generation

The AES-128 confidentiality key and the KEK are both 128-bit strings. The `signtool` utility does not specifically generate any keys for AES-128. A user can use any means to generate sufficiently random 128-bit strings 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 a 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 OTP memory to be used as a 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 a Secure Boot Stream

A boot stream packages the application so that the boot kernel can load the application into memory. Securing the boot stream to prevent malicious attacks requires additional steps, as discussed in the following sections.

Creating a Boot Stream

Once a CCES project is created via the `New CROSSCORE Project Wizard`, the project's output file (build artifact) type defaults to "Executable" (.DXE file), which allows for development and debugging of the application code in a simulator or on a hardware target. Once satisfied with the application's operation, the next step is to convert this executable into a bootable image to either be programmed to source memory or provided by another external host. To generate a loader boot stream (.LDR file), the build artifact type must be changed in the *Project Properties*, as follows:

1. Right-click the project name in the *Project Explorer* view, and select *Properties*.
2. In the *Properties* window that opens, choose `C/C++ Build`→`Settings`.
3. In the `Settings` dialog box that appears, click the `Build Artifact` tab.
4. In the *Build Artifact* page, choose `Loader File` from the drop-down list for `Artifact Type` (the `Artifact name` should be `${ProjName}`, and the `Artifact extension` should be `ldr`).
5. (Optional) In `Output prefix`, enter a string to prepend to the output file name.
6. Click `OK` to save the project settings and close the dialog box.

Full details regarding the creation of a boot loader stream can be found in the *CCES Loader and Utilities Manual*[2].

The output format of the loader utility MUST be binary in order to be compatible with the `signtool` 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 once of these block types, the boot process will terminate.

- *Init Block*

Initialization blocks require a call to user application code prior to the authentication of the boot image and are therefore not supported. If customizations or optimizations are necessary to improve the loading time performance, it's recommended to implement a second stage loader. The first application will contain only the custom code, and then it will in turn issue calls to 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 are therefore not supported.

Secure boot streams use double buffer *Page Mode* to optimize the boot process. This allows for decrypt and hash operations to be performed on received data while new data is being fetched from the boot source. This host in the slave boot mode must ensure, after additional data is sent after the boot stream, that the temp buffer is filled completely. The size of the secure boot stream minus the size of the secure boot header must be a multiple of the size of the temp buffer, which by default is 1024 bytes.

Signing the Boot Image

The generation of the secure boot stream is simply a conversion applied to the typical boot stream. It involves the use of the private key to create a digital signature which is stored in a secure header, which will now be part of the secure boot stream. This is done for all three secure boot types, BLp, BLw, and BLx.

Encrypting the Boot Image

If the BLw or BLx format is chosen, the boot stream will also be encrypted using the 128-bit confidentiality key, along with signing the boot image. If BLw is chosen, the confidentiality key used will be wrapped with the KEK and stored in the secure header of the boot stream as well.

Signtool for Generating Secure Boot Stream

The CCES `signtool` host utility is also used for signing/encrypting the boot stream image and converting it into a secure boot stream image. When `signtool` is used with the `sign` command, various switches are used to specify the keys, as summarized in 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 key pair 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 (required for BLx, optional for BLw. If omitted, signtool 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/encrypted
-outfile **secure.bin**	Specifies the name of the file to create, containing the signed/encrypted version of `bootstream.bin`.

Table 4. Switches for sign Command in signtool Utility

Listing 3 is an example command line using 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. Signing a Boot Stream for Integrity and Authenticity Protection

The final `secboot.bin` output file is both integrity and authenticity protected. The original boot loader stream is still in plaintext format, meaning it's visible. If confidentiality protection is also desired, `signtool` can be invoked for both keyless (BLx in Listing 4) or wrapped (BLw in Listing 5) formats.

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

Listing 4. Signing/Encrypting a Boot Stream for Integrity, Authenticity, and Confidentiality Protection (Keyless)

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

Listing 5. Signing/Encrypting a Boot Stream for Integrity, Authenticity, and Confidentiality Protection (Wrapped)

As discussed in Secure Boot Image Types, 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 regarding the content of the secure boot image.

All image attributes are integrity protected using the same algorithm as the image section; therefore, when the image authentication process completes, and the image is successfully authenticated, the image attributes are known to be trusted.

Attributes are specified as type value pairs, where both the type and value are 32-bit entities. The boot code supports the image attributes summarized in 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 Refer to Anti-Rollback Protection	
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 the Secure Boot Stream into a 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. The CCES Command Line Device Programmer (`CLDP`) utility can be used to program memory on a target board. For full details, refer to the *Loader and Utilities Manual*.

The `CLDP` requires the use of a driver that runs on the processor core to access the flash memory. If the part is locked, `CLDP` will need to gain access to the locked part to load the driver. This is done the same way as

gaining access for debugging. The matching secure debug key must be passed in. Further details are provided in the Secure Debug Access section.

With `CLDP` being able to gain access to a locked system, the user is able to continue debugging and program new secure images into the flash device.

Programming Keys into OTP Memory

All keys used by the boot kernel for authentication and decryption must be programmed into the OTP memory. Programming the OTP memory in general can be accomplished in one of two ways:

1. Use the same CLDP used for programming flash memory (see the *Loader and Utilities Manual*).
2. Run an application on the processor that calls the OTP read/write API functions stored in the ROM.

It is recommended that all OTP access is done through the OTP API functions provided in the ROM. To use the API, include the `cdefBF70x_rom.h` header file (for ADSP-BF70x-based projects) or the `cdefSC58x_rom.h` header file (for ADSP-SC58x/ADSP-2158x-based projects) in the source code.

API for OTP Programming

Listing 6 shows 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

However, to avoid potential mistakes with manual programming, the recommended API 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, the details of key location and programming multiple locations for double redundancy are fully hidden.

In the code example 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 the OTP memory via the `adi_rom_otp_pgm()` function. Here, even though sixteen 32-bit words are written, only 14 are needed for ECDSA 224 bit (used in secure booting).

This same API can be used to program other keys at the same time. In order to do so, prepare `uint32_t` type buffers for the Secure Debug Key and the AES-128 Private Key, 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 as in the code example of Listing 8.

```
/* 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

Listing 9 shows the struct type definition that the `adi_rom_otp_pgm()` API 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 Memory

All non-zero data in the array provided are programmed into OTP memory. Programming is done according to the OTP controller requirements, and only double redundant mode is supported. For each 32-bit data element, four locations will be programmed, as required by the OTP controller.

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

All programming is performed in accordance to the requirements in the OTP Controller (OTPC) documentation, and all recommended values are used. For more information, refer to the OTPC documentation.

Corresponding API to ROM functions exist, allowing the user to read contents in OTP memory, as shown in Listing 10.

```
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_info` parameter (one of several commands enumerated in `OTPCMD`) is fetched from OTP memory and placed at the location specified by `data`. This API uses the `adi_rom_otp_get_data` API.

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

Locking the Part

In order to lock a processor and enable security, a particular location in OTP must be written to. This should be done using the ROM API function `adi_rom_lock` API, as shown in Listing 11.

```
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. For more information, refer to the security documentation regarding a locked device.

At this point, only secure boot streams are allowed, which will be verified through cryptographic means. Normal (non-secure) boot streams will automatically 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, set the core to idle, and not complete the booting process.

Accompanying Examples

This EE-Note is accompanied by example projects in various directories in the *Associated ZIP file*[6] to demonstrate the different stages of developing a secure boot system, as discussed above. Each directory, except for `keygen`, has separate sub-directories for the project configuration (either for ADSP-BF70x or for ADSP-SC589), while sharing a common source code base.

keygen Directory

The *keygen* directory does not contain any source code. It uses post-build commands to execute the `signtool` utility to generate keys and sign/encrypt boot streams.

Using keygen Example Project for Generating Keys

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

Python scripting is required for the `keygen` project to generate all of the required collateral. Update the path to the python executable in the `Makefile`. ActivePython can be downloaded from http://www.activestate.com/activepython/downloads[3]. ActivePython 2.7 was used to develop and test these examples.

The `keygen` project is a simple CCES `makefile` project. By building the project, the `makefile` will generate all required keys and associated header files using the `signtool` utility, as well as some short Python scripts. All other projects will utilize the keys and header files generated by `keygen` located in its project directory.

Build the `keygen` project only once, and do not delete the generated keys! The clean function is disabled to protect the keys because, if the keys are programmed into OTP memory, the newly generated keys will replace the old ones, which will be unrecoverable.

keygen: Random Keys

In the `Makefile`, the `genRandKey.py` script is used to create random 128-bit keys for the encryption, emulation, and wrapper keys; and the `bin2inc.py` script is then used to create a C array in a header file so that the key can be easily programmed into OTP memory. The `Makefile` is invoked as in Listing 12.

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

Listing 12. Using Keygen Project to Generate a Random Key

keygen: Authentication Key

The `signtool` 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 in the `keygen` project, place the keys in the `keygen` folder with the correct names. The filenames can be found at the top of the `Makefile`:

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

After placing whatever custom keys are being used in the directory, simply build the project. Any missing keys will be generated, and all other collateral will be generated based on the provided keys.

Program_OTP Directory

This directory provides examples to setup OTP for the various secure boot image types based on keys generated by `keygen`.

Using the Program_OTP Example Project

The `Program_OTP` example provides a CCES executable project whose application programs the relevant keys from the `keygen` project. Refer to the `keygen` example project for more information on how keys are generated.

The `keygen` project must be built first to provide all the required keys and header files used in the `Program_OTP` project. All keys used by this project are generated by `keygen`, and located in `keygen's` project folder.

As described previously in this EE-note, the project provides three different functions to program the AES-128 authentication key, the encryption key generated using `genRandKey.py`, and the emulation key for debug. The application also contains an additional function that locks the part using the OTP API.

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

In addition to functions to program each individual key, three setup functions are provided: `setupBLp`, `setupBlw`, and `setupBLx`. These functions will program all the required keys from the `keygen` project into the appropriate OTP 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.

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

LedBlink Directory

This directory contains a standard application signed for all image types using the generated keys. The application should be programmed into the SPI2 flash memory on the evaluation platform being used.

Using the LedBlink Signing Example Project

The project is a standard CCES project containing the desired application. There source code of this project itself is trivial in that it simply blinks LEDs on the evaluation board. The project is used to show how the normal non-secure boot stream can be converted into a secure boot stream file.

The `LedBlink` project utilizes the `keygen makefile` features to sign the resulting boot stream in all of the available image types by invoking the `signtool` utility via a post-build step. The *Build Steps* tab is found in the `C/C++ Build→Settings` window of the *Properties* dialog found under the `Project→Properties` menu option.

The post-build setting refers to the `makefile` in the `keygen` project. It 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 generated. Listing 13 is an example of the `makefile` command line.

```
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 types of secure boot streams. It will produce the `LedBlink.blpldr`, `LedBlink.blxldr`, and `LedBlink.blxldr` boot stream output files. These boot streams can then be programmed into the flash memory using the `CLDP` utility, as mentioned previously.

SecureBoot_OpenPart Directory

This is an example project that can securely boot a loader stream from SPI2 flash memory using keys generated by `keygen` without writing any OTP memory.

Using SecureBoot_OpenPart Project

Assuming that the secure boot stream has already been programmed into the SPI2 flash memory, this application will:

1. configure the SPI peripheral,
2. configure the boot parameter for secure booting, and
3. securely boot the secure boot stream in the SPI2 flash.

Since the part has not been locked (security is not enabled), the boot kernel will not boot the secure boot stream upon reset. But, when the boot API is used and configured for secure booting, the secure boot stream will be verified and booted, and the LEDs will start blinking.

Finally, the project reads the keys from the files generated from the `keygen` project. There is no need to program the keys into OTP memory to use this project.

Secure Boot on an Open Part

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

The ROM code provides a mechanism to boot a secure boot stream on an open part without writing any keys into the OTP memory. This can be very useful in validating the generated key and application stream before writing to OTP memory. As such, the `SecureBoot_OpenPart` simulation code does not rely on the `Program_OTP` project. The keys used are read in from the files generated by the `keygen` project.

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 booting chapter in the HRM.

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

- `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.
- `keyType` – setting this to `ADI_ROM_CUSTOM_SECURITY` essentially disables reading keys from OTP memory. This allows the keys to be explicitly set in the application.
- `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 associated with 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 indicating the point within the boot process at which 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

Listing 14 provides an application that can be loaded and executed using the emulator. The application's main function calls the ROM API initializing SPI Master boot mode with BCODE = 0. The `ConfigureForSecureBoot` function illustrates the key kernel configurations.

```
/**
 * ConfigureForSecureBoot
 * @param pBootConfig boot configuration structure
 * @param cause the reasong 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 section is only executed after the boot mode's init function has been run */
                pBootConfig->pSource+=16; /* kernel anomally workaround for 0.0 silicon only */

                /* 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 memory.

By utilizing the boot routine in conjunction with a hook function, it is possible to exercise the secure boot process without having written any data into OTP memory. This is accomplished by creating an application that uses the boot routine API to boot the processes, and a hook routine alters the internal configurations of the boot process to shift the boot ROM's context to a secure one.

The example provided utilizes SPI Master boot mode. The generated secure loader file must be programmed into the base of the 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 settings, security status, 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 function. Each function will use the keys generated by the `keygen` project to configure the processor for the associated secure boot image type, and only one can be used at a time (uncomment the desired image type), which must match the type loaded into the SPI flash memory.

There are other code snippets provided for anomalies, which also must be included for correct operation.

Boot Routine

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

Secure Debug Access

The Test Access Port Controller (TAPC) provides a means of restricting access to the processor's secure resources. 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 OTP memory by the user for access.

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

The key is set in OTP memory using the OTP program API to program the `secure_emu_key` field in the 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 are stored to `TAPC_SDBGKEY0[31:0]`
2. Bits 63:32 of the key in OTP are stored to `TAPC_SDBGKEY1[31:0]`
3. Bits 95:64 of the key in OTP are stored to `TAPC_SDBGKEY2[31:0]`
4. Bits 127:96 of the key in OTP are stored `to TAPC_SDBGKEY3[31:0]`

The boot kernel will then set the `TAPC_SDBGKEY_CTL.VALID` bit to enable comparisons of the keys for allowing debug access.

The TAPC registers are accessible by the application. If the processor is locked and the application booted from flash clears the `VALID` bit in `TAPC_SDBGKEY_CTL` register, key comparisons will be disabled. As such, debug access will be permanently disabled.

Once the boot kernel has loaded the user key, a secure debug key can be provided to the TAPC via JTAG. This is done by providing the secure debug in either the `ADSP-BF70[x]-resets.xml` file in the `System\ArchDef` directory of the CCES installation path or in a custom board XML file.

In the XML file, the secure debug key is passed in by setting the `userkey[x]` registers, as shown in Listing 15. More information can be found in the `CCES-ReleaseNotes-1.1.0.pdf`, located in the `\Docs` sub-directory in the CCES installation.

```
<register name="userkey0" reset-value="0x00001111" core="Common" />
<register name="userkey1" reset-value="0x22223333" core="Common" />
<register name="userkey2" reset-value="0x44445555" core="Common" />
<register name="userkey3" reset-value="0x66667777" core="Common" />
```

Listing 15. Providing Secure Debug Key for Entry into Locked Part

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 memory 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 unless it is being sent remotely, in which case it should be transmitted over a secure connection.

Errors 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. A failure to match the `secure_emu_key`, however, will not result in a failure; rather, booting will continue as normal.

Boot ROM API

The `adi_rom_Boot` boot routine provides access to boot an application at run-time through a supported peripheral and provides the ability to further customize each boot mode's process. It can be used for any kind of second-stage boot for supported boot modes, providing options to boot from any device and any channel, whereas booting directly limits the choice to the default devices and channels. Often, any auto-configuration or detection of the device can also be disabled. For full details regarding the `adi_rom_Boot` API, refer to the Boot ROM chapter in the processor's hardware reference manual.

Using Secure Debug Key with CCES

When the Secure Debug Key is programmed into the OTP memory and the part is locked, the part is secure with security features enabled. In order to gain access with the host debugger, a key must be passed in to be compared against the Secure Debug Key stored in OTP memory. If the keys match, then access is granted, otherwise access is denied.

A key can be passed in from the debugger via a custom board support XML file or in the relevant evaluation platform XML file (`ADSP-BF70x-resets.xml` or `ADSP-SC589-registers.xml`) found in the `\System\ArchDef` sub-directory in the CCES install path. The benefit of using a custom board support XML is the ability to use different keys for different boards. A different custom board support XML file would be created for every board and for 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, consider the key defined as shown in Listing 16.

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

Listing 16. Example Key

With the above key, the XML must contain the register settings as shown in Listing 17 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" />
```

Listing 17. XML Modifications Required after Key Is Known

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 set up the secure system.

References

[1] *ADSP-BF70x Blackfin+ Processor Hardware Reference*. Rev 1.0, October 2016. Analog Devices, Inc.

[2] *CrossCore Embedded Studio 2.4 Loader and Utilities Manual*. Rev 1.8, September 2016. Analog Devices, Inc.

[3] *ActivePython (http://www.activestate.com/activepython). 2.7, ActiveState.*

[4] *ADSP-SC58x SHARC+ Processor Hardware Reference Manual*. Rev 0.3, April 2016. Analog Devices, Inc.

[5] *Tips and Tricks Using the ADSP-SC58x/ADSP-2158x Processor Boot ROM (EE-384)*. Rev 1, September 2015. Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – November 26th, 2014 by G. Yi and S. Molloy	Initial Release
Rev 2 – October 13th, 2016 by G. Yi	Added support for ADSP-SC58x/ADSP-2158x SHARC+ processors. Corrected byte offsets in Table 1.