Tips and Tricks Using the ADSP-SC58x/ADSP-2158x Processor Boot ROM

Contributed by Gaharwar, Harshit *Rev 1 – September 30, 2015*

Introduction

The ADSP-SC58x/ADSP-2158x products (hereafter referred to as ADSP-SC58x) are members of the SHARC+ ™ family of processors. Unlike previous SHARC® processors like the ADSP-212xx, ADSP-213xx, and ADSP-214xx families, the ADSP-SC58x/ADSP-2158x processors feature an on-chip boot ROM (mapped to L2 system memory) to control the boot scenario. The boot ROM provides a mechanism via One-Time Programmable (OTP) memory to customize various aspects of the boot process, including enabling/disabling of specific features such as the use of cache memory, overriding default boot peripheral initialization and timing parameters, and disabling of boot modes. For more details on the memory-map, refer to the *ADSP-SC58x SHARC+ Core Embedded Processor Data Sheet* [1].

The non-secure (i.e., "standard") boot process does not verify any signatures nor perform any decryption on the application's binary boot stream; however, the ADSP-SC58x processor supports secure booting when security is enabled, where the boot kernel uses cryptographic algorithms to perform checks of the application binary and to decrypt it. The purpose of this EE-note is to highlight the new boot features and enable the user to create various boot streams and understand how boot customization can be achieved using the boot ROM and OTP memory.

While "ADSP-SC58x" is used throughout this EE-note to include the ADSP-2158x processors, any differences between the two are explicitly highlighted where applicable. For more information related to this, please refer to the *ADSP-SC58x SHARC Processor Hardware Reference*[2].

Booting of the Processor

The boot kernel in the ADSP-SC58x processor provides support to boot from various peripherals, as defined by the `SYS_BMODE` pins:

- SPI2 Master Boot
- SPI2 Slave Boot
- LinkPort0 Slave Boot
- UART0 Slave Boot

Booting at Power-On Reset (POR)

Upon reset, the processor begins fetching instructions from the boot ROM. The boot ROM code facilitates loading an application from the boot source. It can automatically initialize certain peripherals for communication based on a chosen boot mode prior to loading the application itself.

At POR, only the primary booting core is released from reset, while the other cores in the system are held in reset until released by the application running on the primary booting core.

During POR, the primary cores are:

- ADSP-SC58x→Core0 ARM Cortex-A5
- ADSP-2158x→Core1 SHARC

Booting via ROM API

The ROM API can be accessed at runtime by any core to boot an application from a boot source, but booting via the API is completely different as compared with POR, as it depends on which core is calling it:

- If the ROM API is invoked from the ARM Cortex A5 (core0), then the boot kernel can boot applications to all three cores.
- If the ROM API is invoked from the primary SHARC (core1), then the boot kernel can boot applications to both SHARC cores (core1 and core2), but not to the ARM (core0).
- If the ROM API is invoked from the secondary SHARC (core2), then the boot kernel can only boot the core2 SHARC (not the core 0 ARM nor the core1 SHARC).
- The core that is executing the second-stage loader (required to load to the other cores) can execute a call to the boot ROM to boot the main application image or even a third-stage loader, if one is required. The boot kernel API can be used to implement custom boot modes as well, using the `dbootcommand` structure. Usage of the ROM API is shown in Listing 1:

```
void * adi_rom_Boot(void *pAddress, uint32_t flags, int32_t blockCount,
ROM_BOOT_HOOK_FUNC * pHook, uint32_t dbootcommand);

int main(int argc, char *argv[])
{
    adi_initComponents();

/* Configure secure peripheral register to do secure accesses to memory */
    /* Secure peripheral register for SPI2_Rx_DMA */
    *pREG_SPU0_SECUREP106= 0x3;
    /*Secure peripheral register for  MDMA0_SRC */
    *pREG_SPU0_SECUREP88= 0x3;
    /*Secure peripheral register for  MDMA0_DST */
    *pREG_SPU0_SECUREP89= 0x3;
/* Call ROM API to boot via SPI2 configured for memory-mapped mode of operation */
    adi_rom_Boot(0x60000000,0,0,0,0x207);
    return 0;
}
```

Listing 1. Usage of ROM API for SPI Master

The SPU_SECUREPx register for the appropriate boot peripheral/DMA should be configured such that it performs secure accesses to the memories, which is the default setting in order to avoid system fabric errors and boot failure.

Application Types

The ADSP-SC58x processor has three cores, so various combinations of cores for an application can be developed. Depending on the boot image for which core, the following boot options are supported:

- Single-Core Application
  - Core0 ARM Cortex-A5: boot on POR and via ROM API.
  - Core1/Core2 SHARC: boot via ROM API only.
- Dual-Core Application
  - With core0 ARM Cortex-A5 as the primary core: boot on POR and via ROM API.
  - With core1 SHARC as the application core: boot via ROM API only.
- Multicore Application (all three cores)
  - Booting on POR and via ROM API.
- The ADSP-2158x processor has two SHARC+ cores. Depending on the boot image for which core, the following boot options are supported:
- Single-Core Application
  - Core1 SHARC: boot on POR and via ROM API.
  - Core2 SHARC: boot via ROM API only.
- Dual-Core Application
  - With core1 SHARC as the primary core: boot on POR and via ROM API.

OTP API Overview

The ADSP-SC58x boot ROM includes a set of functions to access OTP memory, thus providing a mechanism via OTP memory to customize the boot process, including configuring the Clock Generation Unit (CGU), initializing the Dynamic Memory Controller (DMC0), storing keys for the secure boot process, etc. All OTP accesses are allowed using the following OTP APIs only:

- To program the OTP data field

```
bool adi_rom_otp_pgm(otp_data* data);
```

- To read the OTP data field

```
bool adi_rom_otp_get( OTPCMD cmd, uint32_t*data);
```

- To lock the ADSP-SC58x processor (for secure booting)

```
bool adi_rom_lock();
```

The *Associated ZIP file*[3] for this EE-note contains example code in the "`5. OTP API code`" folder to access (program/read) the ADSP-SC58x processor OTP space from the Cortex-A5 core. For more information, refer to the "*OTP Controller*" chapter of the *Hardware Reference Manual*.

- Once the part is locked, it can only be accessed via JTAG by providing the emulation key.

Boot Stream Generation

As mentioned, the ADSP-SC58x processors support both standard and secure booting. The secure booting feature resident on the ADSP-SC58x processors is the same as on the ADSP-BF707 Blackfin+™ processor, as detailed in the *Secure Booting Guide for ADSP-BF70x Blackfin+ Processors (EE-366)*[4] application note. Please refer to this document for more details regarding secure boot mode support.

Standard Boot Streams

The elfloader utility ('`elfloader.exe`') in the CCES root directory is used to generate standard boot streams for all three cores. The below Listings 2-8 depict invocation of the elfloader utility to generate the boot stream for a SPI Master boot in single-bit SPI mode for numerous scenarios.

For a single-core application running on the Cortex-A5 ARM core, the ARM executable file (e.g., `ARM_Application_Core0`) must be designated for the appropriate core using the `-core0` switch, and an output loader stream (LDR) file name (e.g., `SPIMASTER_Single_ARM.ldr`) can optionally be designated using the `-o` switch, as shown in Listing 2:

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC589 -core0=ARM_Application_Core0
-b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o SPIMASTER_Single_ARM.ldr
```

Listing 2. Elfloader Command Line for Application Created for Core0 ARM Cortex-A5

If the single-core application instead targets either the primary or secondary SHARC core, a similar calling convention is used, but the command line must associate the appropriate SHARC executable file (DXE) with the targeted core, using the following switches:

- `-core1` for the primary SHARC (Listing 3)
- `-core2` for the secondary SHARC (Listing 4)

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC589 -
core1=SHARC0_Application_Core1.dxe -b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o
SPIMASTER_Single_SHARC0.ldr
```

Listing 3. Elfloader Command Line for Application Created for Core1 SHARC

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC589 -
core2=SHARC1_Application_Core2.dxe -b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o
SPIMASTER_Single_SHARC1.ldr
```

Listing 4. Elfloader Command Line for Application Created for Core2 SHARC

For a multi-core application, it is a mix of the above invocations, but there is an additional concern that must be cared for in the generation of the final boot stream. In the defined boot stream, there is a "Final" tag associated with the last block in the stream destined for the specified core, which indicates to the boot ROM during loading that this is the last block to load before exiting the boot ROM. In a multi-core scenario, however, this tag must be omitted if other cores still need to be loaded after the current one, otherwise the boot ROM will exit prematurely during the boot after loading to that core and result in the other cores not booting at all. The capability to remove the "Final" tag from the boot stream generated for a specific core is available via the `-NoFinalTag` switch, which must be applied to the primary core loader stream in each of the possible two-core scenarios:

- Core0 ARM Cortex-A5 and Core1 SHARC (Listing 5)
- Core0 ARM Cortex-A5 and Core2 SHARC (Listing 6)
- Core1 SHARC and Core2 SHARC (Listing 7)

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC589 -core0= ARM_Application_Core0
-core1= SHARC0_Application_Core1.dxe -NoFinalTag= ARM_Application_Core0 -b SPI -f
BINARY -Width 8 -bcode 0x1 -verbose -o SPIMASTER_Single_ARM_SHARC0.ldr
```

Listing 5. Elfloader Command Line for Application Created for Core0 ARM Cortex-A5 and Core1 SHARC

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC589 -core0=
ARM_Application_Core0 -core2= SHARC1_Application_Core2.dxe -NoFinalTag=
ARM_Application_Core0 -b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o
SPIMASTER_Single_ARM_SHARC1.ldr
```

Listing 6. Elfloader Command Line for Application Created for Core0 ARM Cortex-A5 and Core2 SHARC

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC589 -core1=
SHARC0_Application_Core1.dxe -core2 SHARC1_Application_Core2.dxe -NoFinalTag=
SHARC0_Application_Core1.dxe -b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o
SPIMASTER_Single_SHARC1_SHARC2.ldr
```

Listing 7. Elfloader Command Line for Application Created for Core1 SHARC and Core2 SHARC

Finally, for an application loading to all three cores, the same concept applies, except the `-NoFinalTag` switch must be applied twice (once for the ARM core, and once for the primary SHARC core), as shown in Listing 8:

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC589 -core0=
ARM_Application_Core0 -core1= SHARC0_Application_Core1.dxe -core2=
SHARC1_Application_Core2.dxe -NoFinalTag= ARM_Application_Core0 -NoFinalTag=
SHARC0_Application_Core1.dxe -b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o
SPIMASTER_Single_MULTI_CORE.ldr
```

Listing 8. Elfloader Command Line for Application Created for All Three Cores

The `-bcode` switch in each of the above is in support of single-bit SPI data and can be modified to select among the various supported SPI operating modes (e.g., 0x9 for dual-bit mode and 0xD for quad-bit mode). For more details, refer to the *Hardware Reference Manual*.

Per the above, usage of the elfloader utility can be extended to apply to all the supported boot sources (via the `-b` switch) and to support various LDR file formats (via the `-f` switch). For more details regarding the elfloader utility, consult the *CCES Online Help*[5] documentation.

Generation of Secure Boot Stream

Generation of the secure boot stream is just a conversion of the standard boot stream, involving the use of a private key to create a digital signature, which is stored in a secure header as part of the secure boot stream.

The `signtool.exe` utility, via the `sign` command, is used for signing and encrypting the boot stream image, and is applied for all three secure boot types, as governed by the `-type` switch: Plaintext (BLp), Wrapped (BLw), and Keyless (BLx).

Listing 9 is an example command line to sign a standard boot loader stream (`Normal_Boot_Stream.ldr`, identified by the `-infile` switch) for plaintext security (`-type BLp`) using the private key stored in the key pair file `keychain.der` (designated by the `-prikey` switch), and the converted secure LDR stream (`BLp_Secure_Boot_Stream.ldr`) is designated by the `-outfile` switch.

```
"<CCES Root Directory>\signtool.exe" sign -type BLp -prikey keychain.der -infile
Normal_Boot_Stream.ldr -outfile BLp_Secure_Boot_Stream.ldr
```

Listing 9. Signtool Command Line to Sign Boot Stream for Integrity/Authenticity Protection

If confidentiality protection is also desired, it can be either keyless (`-type BLx`) or wrapped (`-type BLw`). For keyless encryption, only the encryption key file is provided, using the `-enckey` switch (Listing 10):

```
"<CCES Root Directory>\signtool.exe" sign -type BLx -prikey keychain.der -enckey
encrypt_key.bin -infile Normal_Boot_Stream.ldr -outfile BLx_Secure_Boot_Stream.ldr
```

Listing 10. Signtool Command Line to Sign/Encrypt Boot Stream for Integrity/Authenticity/Confidentiality Protection

Wrapped encryption, where the cipher key is sent along with the secure boot stream, requires both an encryption key file (using the `-enckey` switch) and a wrap key file, as specified by the `-wrapkey` switch (Listing 11):5]

```
"<CCES Root Directory>\signtool.exe" sign -type BLw -prikey keychain.der -enckey
encrypt_key.bin -wrapkey wrapper_key.bin -infile Normal_Boot_Stream.ldr -outfile
BLw_Secure_Boot_Stream.ldr
```

Listing 11. Signtool Command Line to Sign/Encrypt Boot Stream for Integrity/Authenticity/Confidentiality Protection with Wrapped Encryption Key

The `Normal_Boot_Stream.ldr`, `keychain.der`, `BLp_Secure_Boot_Stream.ldr`, `encrypt_key.bin`, `wrapper_key.bin`, `BLx_Secure_Boot_Stream.ldr` and `BLw_Secure_Boot_Stream.ldr` files are in binary format.

Tips for Secure SPI Master Boot

For SPI master boot mode, the ROM code checks for the `-bcode` value present in the standard boot stream to determine the SPI configuration to be used. For a secure boot stream, which may be encrypted, extra steps are required to perform the same auto-detect functionality. The standard boot image can be signed

with attribute 0x80000002 set to a value of 0x0 through 0xF, which will determine the SPI configuration to use. If the attribute was not found in the secure header, the default `bcode` of 0x1 is applied. For example, if the SPI needed to be configured for quad-bit mode, the value 0xD must be associated with attribute 0x80000002, as in Listing 12:

```
"<CCES Root Directory>\signtool.exe" sign -type BLw -prikey keychain.der -enckey
encrypt_key.bin -wrapkey wrapper_key.bin -attribute 0x80000002=0xD -infile
Normal_Boot_Stream.ldr -outfile BLw_Secure_Boot_Stream.ldr
```

Listing 12. Signtool Command Line to Sign/Encrypt Boot Stream for Integrity/Authenticity/Confidentiality Protection with Wrapped Encryption Key for Attribute 0x80000002 = 0xD.

Tips for Secure Slave Boot

In order to boot the ADSP-SC58x processor in slave boot mode, the host code should send an extra 1024 dummy bytes at the end of all secure boot streams. This will ensure that the boot stream is completely sent from the host and is received by the processor to boot an application.

The *associated ZIP file* has a '`2. Loader Streams`' folder, which generates both standard and secure boot streams for a simple LED_Blink application (located in the '`1. Led_Blink_Code`' folder) running on an ADSP-SC58x EZ-KIT® evaluation system.

Boot Support in Open/Locked Parts

By default, the processor is considered to be an "open" part (i.e., in the non-secure, default state). Open parts support both standard and secure boot. In order to lock a processor and enable security, a particular location in OTP memory must be written.

Standard booting is no longer supported once the part is locked.

- The plaintext format (BLp) boot stream can be authenticated by pre-programming the corresponding public key of the ECDSA-224-bit algorithm into the OTP `public_key` field using the OTP Program API.
- The wrapped key format (BLw) boot stream 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.
- The keyless format (BLx) is similar to the wrapped key format, except the image does not contain the key. The decryption key for the data must be pre-programmed into the OTP `pvt_128key` field using the OTP Program API.
- Once the part is locked, the debugger will only have access when the userkey passed from the debugger matches the emulation key that must be programmed into the OTP `secure_emu_key` field using the OTP Program API *before* locking the part.

Public and Encryption (Private) Key in OTP Space

- There are two instances of public keys and four instances of encryption keys in the OTP space available. By default, the `public_key0` field and the `pvt_128key0` field are used for authentication and decryption of the secure boot stream. In order to use the other instances of the keys, like `public_key1` and `pvt_128key1`, the previous instances need to be invalidated in the OTP space by setting the `pubkey0Inv` and `privkey0Inv` bits in the OTP space using the OTP Program API.
- All the public and private keys can be invalidated using the various *key*Inv* fields provided in the `ADI_ROM_OTP_BOOT_INFO` structure, which is useful when a new key is required, as the boot ROM will always use the lowest valid key enumeration. If `key0` is valid, then it is used, if `key0` is invalid and `key1` is valid, then `key1` is used.

Once a key is invalidated, it cannot be used again.

Testing Secure Boot Using ROM API

The ROM code provides a mechanism to boot a secure boot stream without writing any keys into OTP memory, which can be very useful in validating the generated keys and application stream before writing to OTP memory and locking the part. 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. The *associated ZIP file* contains a '`4. ROM_API_Flags`' folder, which has example code for using the ROM API Hook function.

Useful Boot ROM Features

Booting to External Memory

The following techniques can be used to boot an application which is mapped to external memory:

- The boot ROM supports initialization blocks to load code on-chip and run it prior to attempting to load the next block in the stream. This code is a small executable that can initialize the DMC controllers prior to any attempt by the boot sequence to actually load code/data to external DDR memory, and it is supported using the `-init` switch in the elfloader command line (Listing 13).

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC589  -core0=ARM_Application_Core0
-init Initcode_Core0 -b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o
SPIMASTER_Single_ARM.ldr"
```

Listing 13. Elfloader Command Line for Application in Cortex-A5 Core with Initialization Block

- DMC initialization can be pre-programmed into the OTP `dmcEn` field using the OTP Program API. By setting the `dmcEn` field of the `ADI_ROM_BOOT_CONFIG` structure, the `ADI_ROM_OTP_DMC_CONFIG` structure is read from OTP and used to configure the DMC peripheral.

- A second-stage loader can be implemented, where the first application configures the external memory controller and then issues a call using the boot routine to boot an application into external memory.

Optimizing Boot Time

Overall boot time performance can be increased using the following techniques:

- Program the CGU to increase clocks throughout the device by programming the OTP `cgu` field using the OTP Program API.
- Use an initialization block to customize boot mechanisms exposed by the boot kernel. In addition to configuring the external memory controllers, initialization block code can also be used to modify the CGU and peripheral bit rates/settings. Because this code is executed at the start of the boot process, the rest of the application can load much faster with whatever optimized settings set in the initialization block.

  - Initialization blocks require a call to user application code prior to the authentication of the boot image; therefore, it is not supported for secure boot streams.

- SPI master boot mode can done in dual-bit or quad-bit mode, if the flash supports it, which is supported via the `-bcode` switch when generating the boot stream.
- A second-stage loader can be implemented, where the first application configures the CGU to run the processor at maximum speed and then issues a call using the boot routine to boot at the desired speed.
- Secure boot image authentication can be disabled by setting the `ADI_ROM_OTP_BOOT_INFO` field `decryptOnlyEn` to 1 and the `decryptOnlyEnInv` field to 0, which will remove the overhead associated with authenticating the boot stream.

Debugging Boot Using Global Boot Flags in adi_rom_Boot()

The `adi_rom_Boot()` API supports additional flags which can impact the processing of the boot stream or modify some control behavior after the boot stream has been processed. For more information, please refer to the "*Boot ROM and Booting the Processor*" chapter of the *ADSP-SC58x SHARC Processor Hardware Reference*.

ROM_BFLAG_RETURN

This flag can be set in the boot routine call from the Cortex-A5 core to boot an application to SHARC cores. When set, the boot code will not vector to the entry address of the loaded application once booted. Instead, it will simply return to the original calling routine like any other regular function call and return. This mechanism allows for a core to load a single-core boot stream intended for another core.

The *associated ZIP file* contains a '`4. ROM_API_Flags`' folder, which has example code for using the ROM API function.

ROM_BFLAG_HOOK

When enabled, this flag allows for a callback hook routine to be called after the execution of the initialization and configuration functions that were registered with the boot kernel. The address of the hook function to execute is passed as a parameter when calling the boot routine.

When using the ROM API, this allows for user routines in SRAM to be registered and called. The boot software passes a flag to the hook routine, indicating whether the call was due to completion of the initialization routine or the configuration routine. In addition, a pointer to the entire boot configuration structure is passed, allowing the hook routine to reconfigure the boot process.

Hook routines provide an efficient means of performing validation of a boot peripheral's configuration at boot-time. Software can call the boot API with a specific configuration, and - in the hook routine - verify that the configuration is correct for the parameters passed, at which point it can then pass or fail without actually progressing through the entire boot sequence.

Booting via Non-Default Peripheral Instances

By default, the ADSP-SC58x processor supports booting via the SPI2, UART0, and LP0 peripherals. In order to boot an application using an alternate instance of a supported peripheral (e.g., SPI0, UART1, LP1, etc.), the `dbootcommand` in the boot routine call can be modified accordingly.

The `dbootcommand` can also be programmed into the OTP `bcmd` field of the `ADI_ROM_OTP_BOOT_INFO` structure using the OTP Program API.

The ROM code also provides the option to disable a particular boot mode by programming the OTP `bootModeDisable` field using the OTP Program API.

Using ROM API for a Second-Stage Loader (SSL)

To support extensions to the boot process, a Second-Stage Loader (SSL) can be introduced, where a small application is loaded into the processor using a natively supported boot mode. This SSL kernel can be used to customize the configuration of the processor and/or perform automated tasks as part of the boot process.

An SSL is a stand-alone application that is executed at boot time before the actual application is dynamically loaded into memory. The SSL can be used to invoke a ROM API to boot a second application. For example, this approach can be used to boot an application from external memory mapped to the DMC1 controller into the core(s) of the ADSP-SC58x processor. The *associated ZIP file* contains a '`3. SSL_Code_Example`' folder, which implements this approach.

The boot kernel requires 8k of SRAM (0x200be000 to 0x200bffff) for the stack and buffers. This space is reserved until after the boot process has completed, and the boot kernel will flag an exception if this rule is violated.

Conclusions

This EE-Note summarizes the steps and requirements to generate both standard and secure boot streams for all boot modes of the ADSP-SC58x processor. It also explains the usage of the ROM API with accompanying examples in use case scenarios like SSL implementation, secure boot on open parts, etc.

References

[1] *ADSP-SC582/583/584/587/589/ADSP-21583/584/587 SHARC+ Dual Core DSP with ARM Cortex-A5 Preliminary Data Sheet*. Rev PrC, June 2015. Analog Devices, Inc.

[2] *ADSP-SC58x SHARC Processor Hardware Reference*. Preliminary Revision 0.2, June 2015. Analog Devices, Inc.

[3] *Associated ZIP File for EE-384*. Rev 1, September, 2015. Analog Devices, Inc.

[4] *Secure Booting Guide for ADSP-BF70x Blackfin+ Processors (EE-366)*. Rev 1, November 2014. Analog Devices, Inc.

[5] CrossCore® Embedded Studio On-Line Help > SHARC® Development Tools Documentation > Loader and Utilities Manual > Loader for ADSP-SC58x/ADSP-2158x Multicore Processors.

Document History

Revision	Description
Rev 1 – September 30, 2015 *by Harshit Gaharwar*	Initial Release