Tips and Tricks Using the ADSP-SC59x/ADSP-2159x/ADSP-2156x Processor Boot ROM

Contributed by Juganta Saikia and Madhumadhi Srinivasan *V01 – May 11, 2023*

Introduction

The ADSP-2156x, ADSP-2159x and ADSP-SC59x family products are members of the SHARC+ ™ family of processors. Unlike previous SHARC® processors like the ADSP-212xx, ADSP-213xx, and ADSP-214xx families, the ADSP-2156x/2159x/SC59x processors feature an on-chip boot ROM (mapped to L2 system memory) to control the boot scenario. The boot ROM provides a mechanism using the One-Time Programmable (OTP) memory to customize distinct aspects of the boot process, including enabling or 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, see the *ADSP-21566/21567/21569 SHARC+ Single Core High Performance DSP (up to 1 GHz) Datasheet*[1] and the *ADSP-21591/21593/21594/ADSP-SC591/SC592/SC594: SHARC+ Dual-Core DSP with Arm Cortex-A5*[2].

The non-secure (*standard*) boot process does not verify any signatures or perform any decryption on the application binary boot stream. However, the ADSP-2156x/2159x/SC59x processors support secure booting when an engineer enables security. The boot kernel uses cryptographic algorithms to perform checks of the application binary and to decrypt it. This EE Note highlights the new boot features and enables the engineer to create different boot streams, plus understand how a boot customization can be achieved using the boot ROM and OTP memory. The EE Note also includes boot time estimations for different boot modes.

Take-aways

This EE447 Note summarizes the steps and requirements to generate both a standard and a secure boot stream for all boot modes of the ADSP-2156x, ADSP-SC59x and ADSP-2159x processor families. It also explains the use of the ROM API with examples in use case scenarios such as SSL implementation, secure boot on open parts, and so forth.

Different scopes for optimizing boot times are discussed. Also, the EE447 Note provides linear equations for boot time estimation in different boot modes. Based on different boot image sizes, an engineer can construct an optimized loader stream that uses less boot time.

Booting of the Processor

The boot kernel in an ADSP-2156x, ADSP-2159x, and ADSP-SC59x processor can support the boot from different peripherals, as defined by these `SYS_BMODE` pins:

- SPI2 Controller Boot
- SPI2 Target Boot
- LinkPort0 Target Boot
- UART0 Target Boot
- Octal SPI Controller

Octal SPI flash boot is not supported for power on reset in the ADSP-21565 Low Profile Quad Flat Pack package, it is available using the ROM application program interface (API).

Booting at Power On Reset

Upon reset, the processor begins fetching instructions from the boot ROM. The boot ROM code helps load an application from the boot source. It can automatically initialize peripherals for communication prior to loading the application itself, based on a chosen boot mode. The primary core that starts the Boot ROM code, is the SHARC core in ADSP-2156x and ADSP-2159x families with the Cortex-A5 core used for the ADSP-SC59x processors.

Booting Using the ROM API

The ROM API can be accessed at runtime to boot an application from a boot source. Any core can execute the second-stage loader, which can call the boot ROM to boot the main application image or even a third-stage loader, when needed. The boot kernel API can also implement custom boot modes, using the `dbootcommand` structure. Code Listing 1 and Code Listing 2 demonstrate how to use the ROM API for the SPI flash boot on different processor families.

Code Listing 1: Usage of ROM API for SPI Flash Boot on ADSP-2156x

```
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 MDMA0_SRC */
   *pREG_SPU0_SECUREP110= 0x3;
   /*Secure peripheral register for MDMA0_DST */
   *pREG_SPU0_SECUREP111= 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;
}
```

Configure the SPU_SECUREPx register for the appropriate boot peripheral/DMA should such that it performs secure accesses to the memories, which is the default setting that avoids system fabric errors and boot failure.

Code Listing 2: Usage of ROM API for SPI Flash Boot on ADSP 2159x/ADSP SC59x

```
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 MDMA0_SRC */
   *pREG_SPU0_SECUREP146= 0x3;
   /*Secure peripheral register for MDMA0_DST */
   *pREG_SPU0_SECUREP147= 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;
}
```

The following parameters must be configured when calling the adi_boot_rom() function:

- pAddress–start address of the External Flash, where the boot image is programmed
- flags–different Boot ROM flags
- blockCount–block count value
- pHook–pointer to the hook function defined in the application
- dbootcommand–boot command value for the desired boot mode.
  See the Boot ROM chapter in the specific processor hardware reference for more information about the boot command field for different boot modes.

OTP API Overview

The ADSP-2156x/2159x/SC59x boot ROM includes a set of functions to access OTP memory. This provides a way using the 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 stream creation process.

All OTP accesses use only the following OTP APIs:

- Program the OTP data field

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

- Read the OTP data field

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

- Lock the processor (for secure booting)

```
bool adi_rom_lock();
```

The *EE384 Associated Zip File (EE384.zip)*[3] contains example code in the `/6. OTP API code/` folder to access (program and read) the OTP space from SHARC+ core and Cortex-A5 core. For more information, see the *OTP Controller* chapter of found in the *ADSP-2156x SHARC+ Processor Hardware Reference*[4] and *ADSP-SC59x/ADSP-2159x SHARC+ Processor Hardware Reference*[5].

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

Boot Stream Generation

The ADSP-2156x/2159x/SC59x processors support both standard and secure booting. The secure booting feature resides on the ADSP-2156x, ADSP-2159x and ADSP-SC59x processors is the same as on the ADSP-BF707 Blackfin+™ processor, as detailed in the *EE336: Secure Booting Guide for ADSP-BF70x Blackfin+ Processors*[7] application note. Refer to this document for more details regarding the secure boot mode support.

There was a change in Secure Boot Architecture between ADSP-2156x and ADSP-2159x/SC59x families. See the *EE432: Boot Time Estimation for ADSP-SC59x/ADSP-2159x SHARC+ Processors*[8] for more information.

Standard Boot Streams

Use the elfloader utility (`elfloader.exe`) in the CrossCore® Embedded Studio installation path to generate standard boot streams. Code Listing 3 through Code Listing 6 depict the invocation of the elfloader utility to generate the boot stream for a SPI Flash boot in single bit SPI mode.

Code Listing 3: elfloader Command for Generating an ADSP-21569 Single-core Loader Stream

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-21569 Application.dxe -b SPI -f
BINARY -Width 8 -bcode 0x1 –verbose -o SPIFlash_Single.ldr
```

For the ADSP-2159x and ADSP-SC59x processors, single and multicore boot streams can be generated as shown in Code Listing 4 and Code Listing 5.

Code Listing 4: elfloader Command for Generating an ADSP-21593 Multicore Loader Stream

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-21593 -core1=Application_core1.dxe
-b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o SPIFlash_Single.ldr

"<CCES Root Directory>\elfloader.exe" -proc ADSP-21593 -core1=Application_core1.dxe
-core2= Application_Core2.dxe -NoFinalTag=Application_core1.dxe -b SPI -f BINARY -
Width 8 -bcode 0x1 -verbose -o SPIFlash_MultiCore.ldr
```

Code Listing 5: elfloader Command for Generating an ADSP-SC594 Loader Stream

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC594 -core0=Application_core0 -b
SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o SPIFlash_Single.ldr

"<CCES Root Directory>\elfloader.exe" -proc ADSP-SC594 -core0=Application_core0 -
core1=Application_core1.dxe -core2= Application_Core2.dxe -
NoFinalTag=Application_core0 NoFinalTag=Application_core1.dxe -b SPI -f BINARY -
Width 8 -bcode 0x1 -verbose -o SPIFlash_MultiCore.ldr
```

An engineer can modify the `-bcode` parameter shown in Code Listings four and five that supports single-bit SPI data to select one of the different supported SPI operating modes (for example, `0x5` for dual-bit mode and `0x9` for quad-bit mode). For more details, refer to the corresponding processor hardware reference. Changing the `-b switch` value to `OSPI` generates the loader streams for an OSPI controller boot.

Use the `elfloader` utility to flexibly support other boot sources (`-b` switch) and different LDR file formats (`-f` switch). For more information regarding the different elfloader utility uses, consult the *CrossCore® Embedded Studio 2.9.0 Software*[6]. In Code Listing 6, an engineer can change the boot mode by replacing `-b SPISLAVE` with `-b UARTSLAVE` (for UART) or `-b LPSLAVE` (for Linkport).

Code Listing 6: elfloader Utility to Generate Boot Stream for an SPI Slave Boot

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-21569 Application.dxe -b SPISLAVE
-f BINARY -Width 8 -verbose -o SPIHost_Single.ldr
```

Generating a Secure Boot Stream

Generating the secure boot stream requires converting the standard boot stream to use a private key to create a digital signature. The boot stream stores the signature in the secure header as part of the secure boot stream.

The `signtool.exe` utility, using the `sign` command, works with the ADSP-2156x family for signing and encrypting the boot stream image. The type switch governs the command applied to three secure boot types, which takes values for Plaintext (`BLp`), Wrapped (`BLw`), and Keyless (`BLx`).

The `adi_signtool.exe` command utility works for the ADSP-2159x and ADSP-SC59x processor families, which computes the HASH content of the boot image beforehand and stores it in the secure header. Set the `-add-sbh-hash-224` or `-add-sbh-hash-256` switch to invoke the desired ECDSA algorithm.

Signing Boot Stream for Integrity and Authenticity Protection (IAP)

The ADSP-2156x/2159x/SC59x processor families support both the 224-bit and 256-bit ECDSA algorithm for integrity and authenticity protection. By default, the processors use the 224-bit ECDSA algorithm. For 256-bit ECDSA, set the `-attribute 0x80000003=256` while signing the loader stream.

Code Listing 7 and Code Listing 8 are examples of the command line for an ADSP-2156x/2159x/SC59x processor 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`) designated by the `-outfile` switch.

Code Listing 7: signtool Command to Sign a Boot Stream for IAP in an ADSP 2156x

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

Code Listing 8: signtool Command to Sign a Boot Stream for IAP in an ADSP 2159x/SC59x

```
"<CCES Root Directory>\adi_signtool.exe" -add-sbh-hash-224 sign -type BLp -prikey
keychain.der -infile Normal_Boot_Stream.ldr -outfile BLp_Secure_Boot_Stream.ldr
```

Signing and Encrypting Boot Stream for Integrity, Authenticity, and Confidentiality Protection *(IACP)*

When confidentiality protection is also desired, it can be either Keyless (`-type BLx`) or Wrapped (`-type BLw`). For Keyless encryption, an engineer provides only the encryption key file, using the `-enckey` switch (see Code Listing 9 and Code Listing 10.)

Code Listing 9: signtool Command to Sign/Encrypt an ADSP-2156x Boot Stream for ICAP

```
"<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
```

Code Listing 10: signtool Command to Sign/Encrypt an ADSP-2159x/SC59x Boot Stream for ICAP

```
"<CCES Root Directory>\adi_signtool.exe" -add-sbh-hash-224 sign -type BLx -prikey
keychain.der -enckey encrypt_key.bin -infile Normal_Boot_Stream.ldr -outfile
BLx_Secure_Boot_Stream.ldr
```

Signing and Encrypting Boot Stream for Integrity, Authenticity, and Confidentiality Protection (IACP) with a Wrapped Encryption Key (IACP-WEK)

Wrapped encryption, where the cipher key is sent with the secure boot stream, requires both an encryption key file (`-enckey` switch) and a wrap key file, as specified by the `-wrapkey` switch (see X and Y.)

Code Listing 11: signtool Command to Sign/Encrypt an ADSP-2156x Boot Stream for IACP-WEK

```
"<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
```

Code Listing 12: signtool Command to Sign/Encrypt an ADSP-2159x/SC59x Boot Stream for IACP-WEK

```
"<CCES Root Directory>\adi_signtool.exe" -add-sbh-hash-224 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
```

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.

Secure SPI/OSPI Flash Boot

For SPI flash boot mode, the ROM code checks for the `-bcode` value present in the standard boot stream to determine which SPI configuration to use. For a secure boot stream, which can be encrypted, extra steps are needed to perform the same auto-detect functionality. An engineer can sign a standard boot image with the attribute `0x80000002` set to a value of `0x0` through `0xF`, which determines the SPI configuration to use. When the attribute is not found in the secure header, the default `bcode` of `0x1` is applied. For example, when the SPI needs to be configured for the quad-bit mode, the value `0x9` must be associated with the attribute `0x80000002` (Code Listing 13) to sign and encrypt the boot stream for Integrity/Authenticity and Confidentiality Protection with a Wrapped Encryption Key.

Code Listing 13: Sign/Encrypt with Attribute 0x80000002= 0x9 for IACP-WEK (SPI quad-bit mode)

```
"<CCES Root Directory>\signtool.exe" sign -type BLw -prikey keychain.der -enckey
encrypt_key.bin -wrapkey wrapper_key.bin -attribute 0x80000002=0x9 -infile
Normal_Boot_Stream.ldr -outfile BLw_Secure_Boot_Stream.ldr
```

Secure Host Boot

To boot and ADSP-2156x/2159x/SC59x processor in host boot modes, the host code should send an extra 1024 dummy bytes at the end of all secure boot streams. This ensures the host completely sends that the boot stream and is received by the processor to boot an application.

The *EE447Associated Zip File (TTP_BootROM.zip)*[9] contains the `2. Loader Streams` folder, which generates standard and secure boot streams for a simple LED Blink application (`1. Led_Blink_Code` folder) running on an ADSP-21569/ADSP-21593/ADSP-SC594 EZ-KIT® evaluation system.

Boot Support in Open and Locked Parts

By default, the processor is an *open part* (that is, in the non-secure, default state). Open parts support both standard and secure boot. To lock a processor and invoke security, a specific location in OTP memory must be configured.

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

The processors support the following boot stream formats

- A Plaintext format (`BLp`) boot stream that can be authenticated by pre-programming the corresponding public key of the ECDSA 224/256-bit algorithm with 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 boot stream header. This key must be pre-programmed in the OTP `pvt_128key` field using the OTP Program API.
- The Keyless format (`BLx`) boot stream like 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, only the debugger has access when the user key is passed from the debugger matches the emulation key. This must be programmed into the OTP `secure_emu_key` field in ADSP-2156x using the OTP Program API, *before* locking the part. For the ADSP-2159x and ADSP-SC59x families support two sets of user keys, `secure_emu_key0` and `secure_emu_key1` (one user key is active at a time). For the ADSP-2156x, an `emu_key_disable` field is provided in OTP to disable the emulation key. For the ADSP-SC59x and ADSP-2159x `emu_key0_disable` and `emu_key1_disable` disable the emulation keys.

Public and Encryption (Private) Key in OTP Space

There are two instances of Public keys and four instances of encryption keys available in the OTP space. By default, the `public_key0` field and the `pvt_128key0` field are used for authentication and decryption of the secure boot stream. 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 (including emulator keys in ADSP-2159x/SC59x) can be invalidated using the various *key*Inv* fields provided in the `ADI_ROM_OTP_BOOT_INFO` structure. This is useful when a new key is required, as the boot ROM always uses the lowest valid key enumeration. When the key0 is valid, then it is used; when 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 the ROM API

The ROM code provides a mechanism to boot a secure boot stream without writing any keys into OTP memory. This can be extremely useful in validating the generated keys and the application stream before writing to OTP memory and locking the part. Engineers can perform secure booting by loading an application into memory using the emulator, which (**a**) uses the ROM API function `adi_rom_Boot` in conjunction with a hook function, (**b**) configures the kernel for secure boot, and (c) starts the boot process. The *EE447Associated Zip File (TTP_BootROM.zip)*[9] has the `4. ROM_API_Flags` folder, which contains the ROM API Hook function example code.

Useful Boot ROM Features

Booting to External Memory

The following techniques can be used to boot an application that maps 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 boot stream. This code is a small executable that can initialize the DMC controllers prior to any attempt by the boot sequence to load code/data to external DDR memory, and it is supported by using the `-init` switch in the `elfloader` command line (see X.)

Code Listing 14: elfloader Command Line with Initialization Block

```
"<CCES Root Directory>\elfloader.exe" -proc ADSP-21569 Application.dxe -init
Initcode.dxe -b SPI -f BINARY -Width 8 -bcode 0x1 -verbose -o SPIFlash_Single.ldr”
```

- 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 the 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.

A secure boot to external memory with BLx and BLw image formats is not supported by default. But this can be enabled using a custom error-handler function that needs to bypass the error ID of `0xC` in a multi-stage boot scenario.

Optimizing Boot Time

Engineers can improve overall boot time performance using the following techniques:

- Program the clock generation unit (CGU) to increase clocks throughout the device by configuring 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 controller, the initialization block can be used to modify the CGU and the 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 are configured 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 flash boot mode can be done in dual-bit or quad-bit mode when the flash supports it. This is supported using the `-bcode` switch when generating the boot stream.
- Similarly, the OSPI flash boot mode can be done in dual-STR, quad-STR, single-DTR, single-DTR, and quad-DTR -bit mode, when the flash supports it. This is supported using 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 issues a call using the boot routine to boot at the desired speed.

  Engineers can configure the CGU and DMC by using `adi_rom_CguInit()` and `adi_rom_DmcInit()` APIs prior to calling the boot application using `adi_rom_Boot()`.

  The associated ZIP file contains the `4. ROM_API_Flags` folder, which has example code for CGU and DMC configuration using the ROM API.

OSPI Boot also supports Octal-STR and Octal DTR modes through second stage boot using custom boot approach which will further speed up the boot process. See the Custom Boot Mode section in this EE Note for more information.

Debugging the Boot Using the 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 control behavior after the boot stream processes. For more information, refer to the **Boot ROM** and **Booting the Processor** chapter of the *ADSP-2156x SHARC+ Processor Hardware Reference*[4] or *ADSP-SC59x/ADSP-2159x SHARC+ Processor Hardware Reference*[5].

ROM_BFLAG_RETURN

This flag can be set in the boot routine call from the ROM API application in either SHARC core or Cortex-A5 core. When set, the boot code does not vector to the entry address of the loaded application once booted. Instead, it returns to the original calling routine just like any other regular function call.

The *EE447Associated Zip File (TTP_BootROM.zip)*[9] contains the `4. ROM_API_Flags` folder, which has the example code for using the ROM API function with the ROM_BFLAG_RETURN.

ROM_BFLAG_HOOK

When enabled, this flag allows for calling a hook routine 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 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 validating a boot peripheral configuration at boot-time. Software can call the boot API with a specific configuration, and—in the hook routine—verify the passing of the correct configuration parameters. Then, the routine can pass or fail boot validation without progressing through the entire boot sequence.

Default Booting Peripheral Pin-mux Combinations

For the ADSP-2156x, ADSP-SC59x and ADSP-2159x processors, all default SPI2, UART0, LP0 and OSPI0 boot peripherals support booting through default pin-mux combinations. Table 1 through Table 4 show the default pin-mux combinations for SPI2, OSPI0, UART0, and LP0 peripherals.

Table 1: SPI2 Default Pin-mux For Processors

Signals	ADSP-2156x	ADSP-2159x	ADSP-SC59x
SPI2_MOSI	PA_01	PA_01	PA_01
SPI2_MISO	PA_00	PA_00	PA_00
SPI2_D2	PA_02	PA_02	PA_02
SPI2_D3	PA_03	PA_03	PA_03
SPI2_CLK	PA_04	PA_04	PA_04
SPI2_SEL1b	PA_05	PA_05	PA_05
SPI2_RDY	PB_05	PB_05	PB_05

Table 2: OSPI0 Default Pin-mux For Processors

Signals	ADSP-2156x	ADSP-2159x	ADSP-SC59x
OSPI0_MOSI	PA_01	PA_01	PC_11
OSPI0_MISO	PA_00	PA_00	PC_12
OSPI0_D2	PA_02	PA_02	PC_10
OSPI0_D3	PA_03	PA_03	PC_09
OSPI0_D4	PA_06	PA_06	PD_00
OSPI0_D5	PA_07	PA_07	PC_15
OSPI0_D6	PA_08	PA_08	PC_14
OSPI0_D7	PA_09	PA_09	PC_13
OSPI0_CLK	PA_04	PA_04	PC_08
OSPI0_SEL1b	PA_05	PA_05	PD_01

Table 3: UART0 Default Pin-mux for Processors

Signals	ADSP-2156x	ADSP-2159x	ADSP-SC59x
UART0_CTS	PA_09	PA_09	PA_09
UART0_RTS	PA_08	PA_08	PA_08
UART0_TX	PA_06	PA_06	PA_06
UART0_RX	PA_07	PA_07	PA_07

Table 4: LP0 Default Pin-mux for Processors

Signals	ADSP-2156x	ADSP-2159x	ADSP-SC59x
LP0_DAT0	PB_07	PB_07	PB_07
LP0_DAT1	PB_08	PB_08	PB_08
LP0_DAT2	PB_09	PB_09	PB_09
LP0_DAT3	PB_10	PB_10	PB_10
LP0_DAT4	PB_11	PB_11	PB_11
LP0_DAT5	PB_12	PB_12	PB_12
LP0_DAT6	PB_13	PB_13	PB_13
LP0_DAT7	PB_14	PB_14	PB_14
LP0_CLK	PB_06	PB_06	PB_06
LP0_ACK	PB_04	PB_04	PB_04

Booting Using Non-default Peripheral Instances

By default, the ADSP-2156x, ADSP-SC59x and ADSP-2159x processors support booting using the SPI2, UART0, LP0, and OSPI0 peripherals. To boot an application using an alternate instance of a supported peripheral (for example, SPI0, UART1, LP1), the `dbootcommand` in the boot routine call can be changed.

Engineers can program the `dbootcommand` for example, into the OTP `bcmd` field of the `ADI_ROM_OTP_BOOT_INFO` structure using the OTP Program API.

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

Custom Boot Mode

In comparison to the previous ADSP-SC5xx/2158x/2157x families of processors, the ADSP-2156x/2159x/SC59x processors support a custom boot mode. The kernel provides a mechanism to customize supported boot modes or implement completely new boot modes as second-stage boot loaders. This helps programs to customize booting while taking advantage of the rest of the booting framework. A custom boot mode can provide support for booting through any peripheral that is not supported by the Boot ROM, or it can support one of the same peripherals, but with a different configuration.

All the same security features can be supported when using a custom boot mode.

A full boot mode is a collection of following driver functions, as defined by the boot implementation.

1. Register–installs the remaining driver functions so they can be accessed by the boot process

2. Initialization–initialize the boot source
3. Configuration–configure the boot source
4. Load–read from the boot source
5. Cleanup–called after booting

To install a custom boot mode:

1. Create a first stage boot application to define a Load function.
2. Use the `adi_rom_BootKernel()` API to call the boot kernel once the boot peripheral and pin-muxing has configured. Ensure all the fields of the data structure `ADI_ROM_BOOT_CONFIG` are configured prior to performing the API call.

The boot mode can use the `pModeData` member of `ADI_ROM_BOOT_CONFIG` to preserve and access shared data across the different function calls if required.

All functions have the following prototype:

```
void apiFunction(ADI_ROM_BOOT_CONFIG* pBootStruct);
```

Another way to support custom boot mode is:

1. Create a first stage boot application to define all Init, Config, Load, and Cleanup routines.
2. Use `adi_rom_Boot()` API with hook function installed to update the Init, Config, Load and Cleanup functions after completing the preregister initialization. This is checked by the boot ROM to override the above functions inside the hook function with `ROM_HOOK_CALL_CAUSE: ROM_HOOK_REG_COMPLETE`.

For custom boot mode using run-time API, set `ROM_BCMD_SPIM_DEVICE= 0xF` in the Boot Command.

The *EE447Associated Zip File (TTP_BootROM.zip)*[9] contains the `5. Custom Boot` folder, which has example code for the custom boot mode to implement an Octal STR boot. The Octal STR boot is not supported by the default boot modes.

For more information, see the *ADSP-2156x SHARC+ Processor Hardware Reference*[4] and *ADSP-SC59x/ADSP-2159x SHARC+ Processor Hardware Reference*[5].

Using a Second-stage Loader that Supports Boot Extensions

To support extensions to the boot process, a second-stage loader (SSL) can be use, in which an engineer loads a small application into the processor with a natively supported boot mode. This SSL kernel can be used to customize the configuration of the processor or perform automated tasks as part of the boot process.

An SSL is a stand-alone application started at boot time before the actual application dynamically loads 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 secondary application mapped to external memory allowing the primary

application to perform DMC initialization. The *EE447Associated Zip File (TTP_BootROM.zip)*[9] contains the `SSL` folder, which implements this example.

The boot kernel requires 8k of SRAM for the stack and buffers (`0x200fe000` to `0x200fffff` for the ADSP-2156x processors and `0x201fe000` to `0x201fffff` for the ADSP-2159x/SCC59x processors). This space is reserved until after the boot process completes. The boot kernel flags an exception when this reserved-space rule is violated. The stack region can also be remapped to another memory region for any incoming image conflicts to the stack space.

The `Example 3, SSL_Code_Example` in the *EE447Associated Zip File (TTP_BootROM.zip)*[9] shows how stack can be remapped in the SHARC core application.

Boot Time Estimation for ADSP-2156x SHARC+ Processors

A loader stream is a set of linked blocks, and each block type is responsible for performing a function. The boot time for a particular boot stream primary depends on its size and the distinct kinds of blocks present in the loader stream.

Total boot time for an ADSP-2156x SHARC+ processors is the sum of the pre boot time of the processor plus the boot time consumed for loading the application in each boot mode.

- Pre-boot time measures the configuration of all system resources prior to executing the required boot operation.
- Application loading time is dependent on the loader stream blocks plus the peripheral being used for a particular boot mode.

Considering the pre-boot takes a fix amount of time under certain conditions, application loading time (boot time) can be computed separately for any loader stream using the Boot ROM API. The following figures (Figure 1 through Figure 11) depict the durations for the boot modes (SPI, OSPI, UART, and Linkport) plotted against loader stream sizes. The measurements were taken using the clock settings (CCLK=1 GHz, SYSCLK=500 MHz, and SCLK=125 MHz). The plots show that the boot time is linear with respect to the loader stream size (larger than 10 KB). The linear equations were derived for each boot mode using the linear model of $y = ax + b$, where $\boldsymbol{x}$ is the size of loader stream (KB) and $\boldsymbol{y}$ is boot time in milliseconds. This permits a calculation to estimate the boot time for any loader stream size.

For example, consider a loader stream size of 1000 KB. For the SPI2 Quad boot mode, the boot time plotted in Figure 1 can be estimated using this calculation:

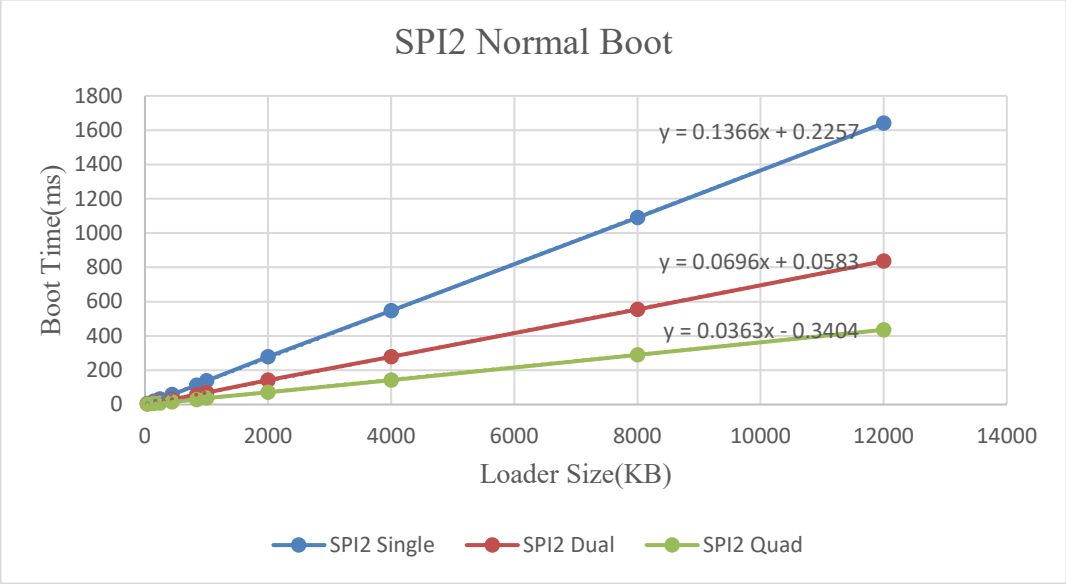
Total Boot time = 0.0363x - 0.3404, where x is loader stream size in KB
Boot time= (0.0363*1000) - 0.3404 = 35.95 ms

For Boot Time information on the ADSP-2159x and ADSP-SC59x processor families, see the *EE432: Boot Time Estimation for ADSP-SC59x/ADSP-2159x SHARC+ Processors*[8] for more information.

SPI Flash Boot

SPI Flash Boot (`SYS_BMODE` 1) mode supports booting from a flash device using an SPI peripheral. In ADSP-2156x SHARC+ processors, the SPI2 instance drives the default SPI boot, which supports single, dual, and quad single transfer rate (STR) modes. Figure 1 through Figure 4 provide linear boot time equations for all the SPI flash boot modes that use a 62.5 MHz SPI clock. The figures include a normal boot and secure boot with ECDSA-256 authentication.

Figure 1: SPI2 Normal Boot at 62.5 MHz



The plot figures for the normal boot are good only for loader stream sizes larger than 10 KB. For smaller boot image sizes, boot time linearity is not maintained due to a smaller number of samples. For stream sizes less than 10 KB, ADI advises the engineer to manually test and evaluate the boot time.

Figure 2: SPI Secured BLp-256 Boot at 62.5 MHz

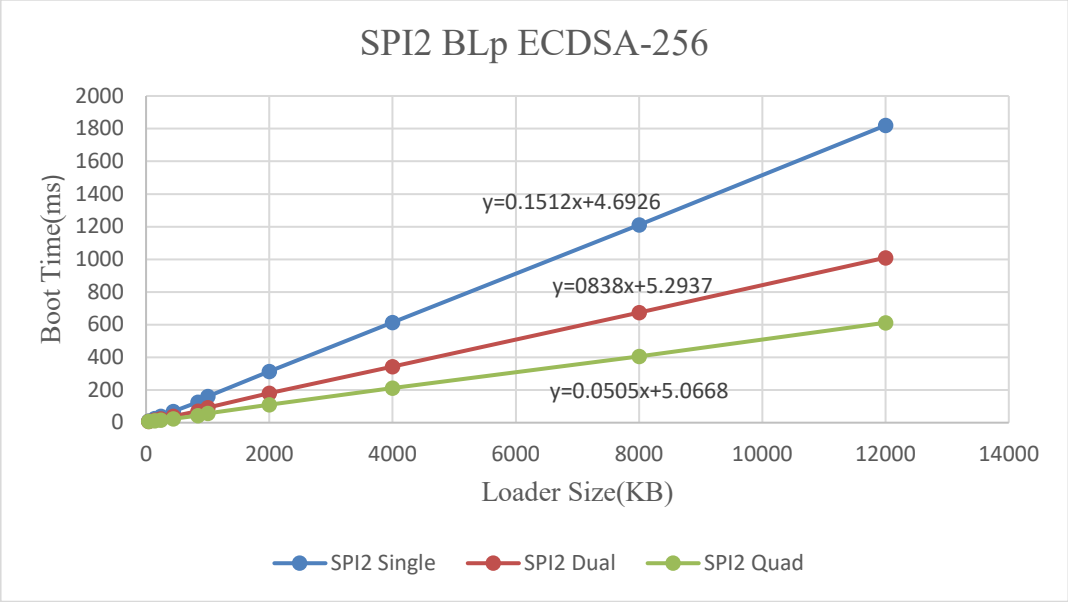


Figure 3 SPI2 BLx-256 Boot at 62.5 MHz

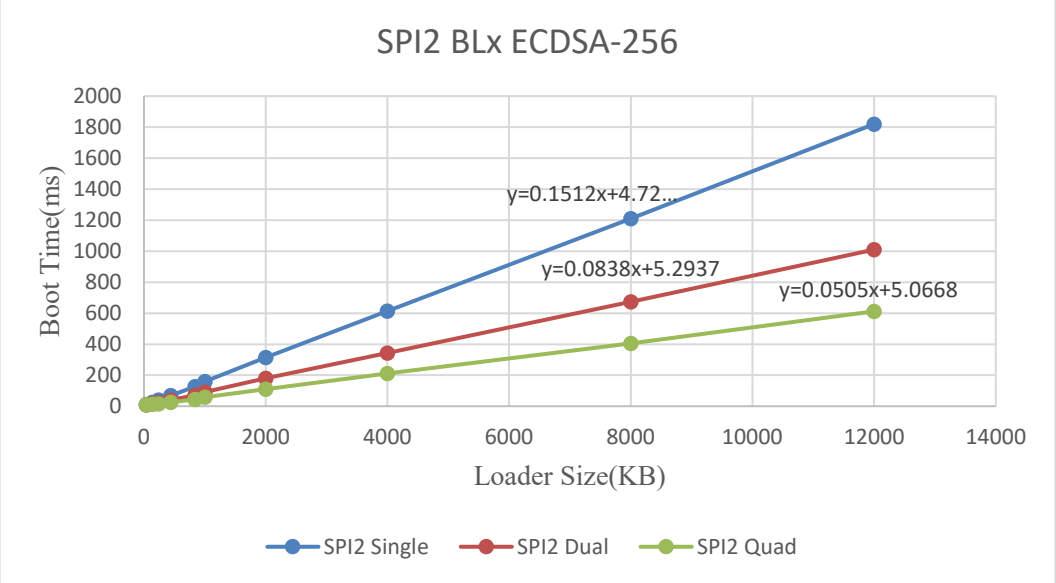
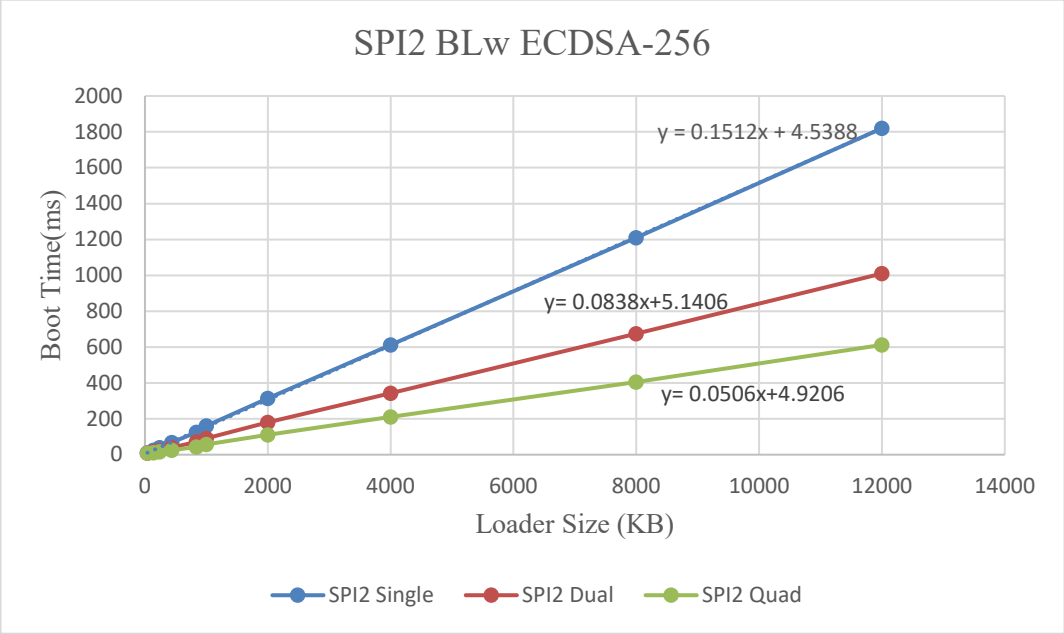


Figure 4: SPI2 BLw-256 Boot at 62.5 MHz

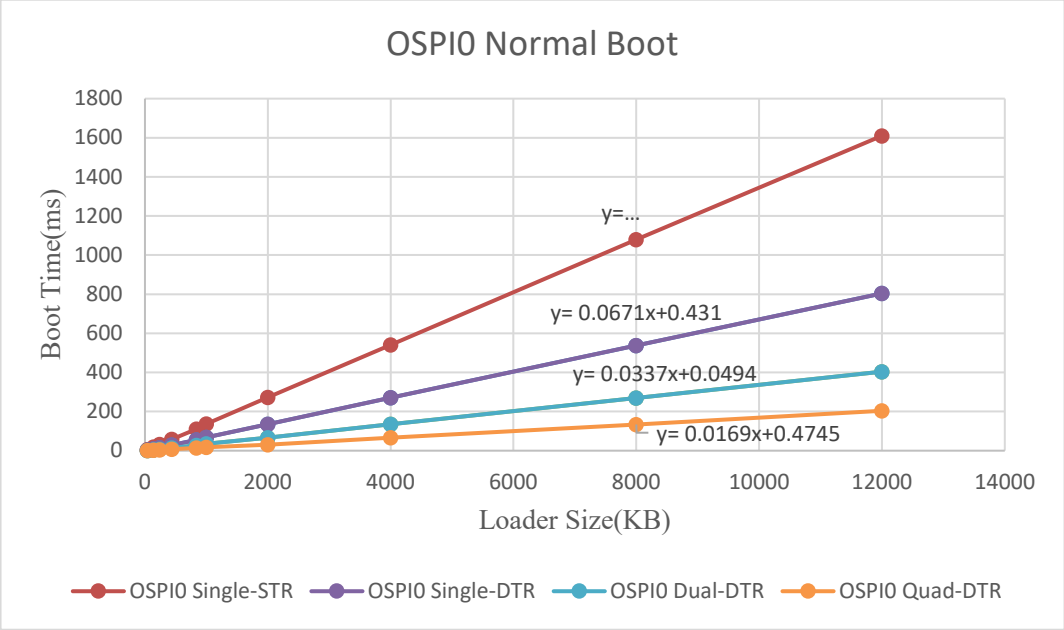


Secure boot with the BLw and the BLx format for the ADSP-2156x/2159x/SC59x SHARC+ processors does not support the application mapped to an off-chip RAM space. The `-chipmemsplitter` switch in CCES elfloader utility can be used, which produces additional loader file outputs for the on-chip and off-chip memory along with the usual loader file output. In that case, the on-chip loader stream can be booted with the BLw/BLx format and off-chip loader stream can be booted with the BLp format. Please refer to the *EE432: Boot Time Estimation for ADSP-SC59x/ADSP-2159x SHARC+ Processors*[8] for more information on how to use this feature with example codes. This is incorporated for larger size loader streams in the linear equations provided in the next sections, where the application data is mapped to an off-chip RAM space.

OSPI Flash Boot

The OSPI Flash Boot (`SYS_BMODE` 5) supports booting from the SPI flash or Octal flash device using the OSPI peripheral which includes single-STR, dual-STR, quad-STR, single-DTR (Double Transfer Rate), dual-DTR, and quad-DTR modes. OSPI boot supports a maximum of 62.5 MHz OSPI Clock. Linear boot time equations are shown in Figure 5 through Figure 9 for all the OSPI flash boot modes at 62.5 MHz OSPI clock The figures include a normal boot and secure boot with ECDSA 256 authentication.

Figure 5: OSPI Normal Boot at 62.5 MHz



Boot time in the OSPI Single-DTR and Dual-STR boot modes are the same. Similarly, Single-Quad and Dual-DTR boot timings are also similar.

Figure 6: OSPI BLp-256 Boot at 62.5 MHZ

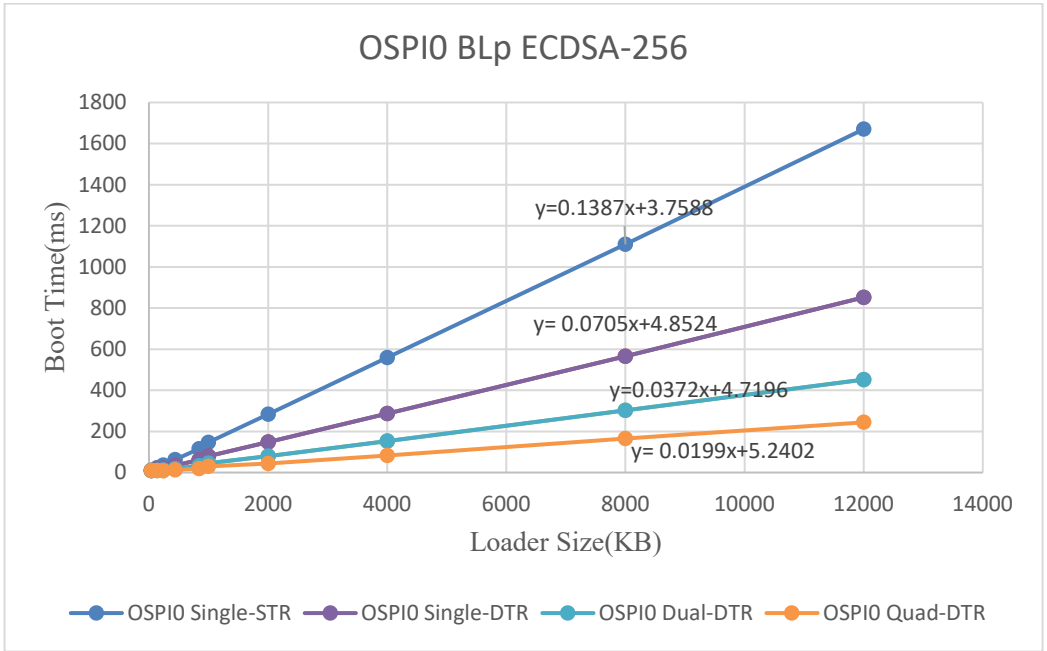


Figure 7: OSPI0 BLx-256 Boot at 62.5 MHz

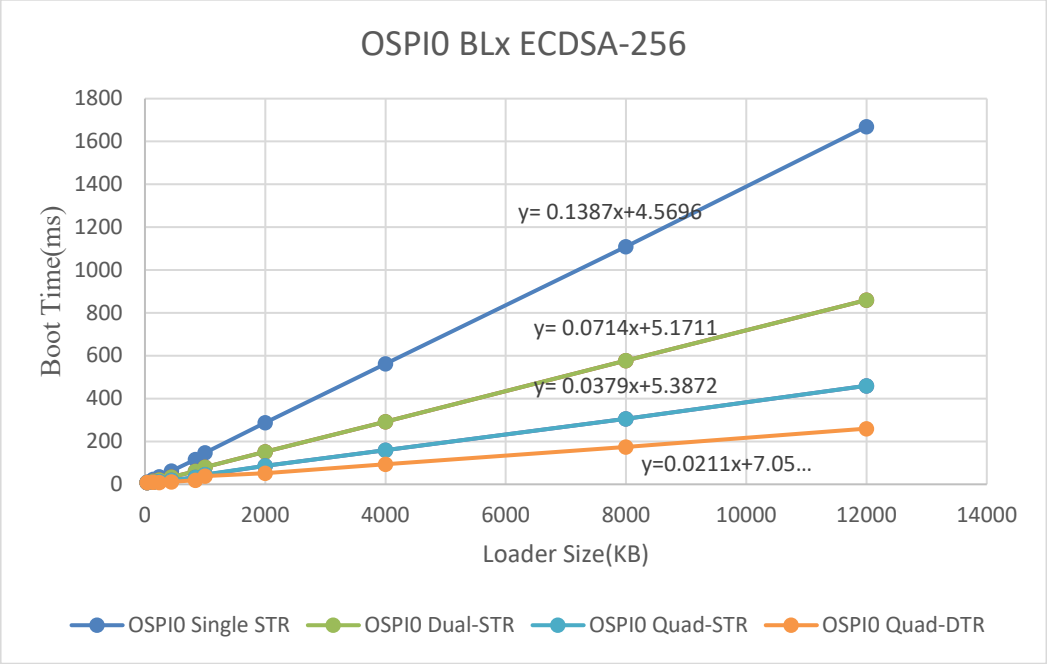
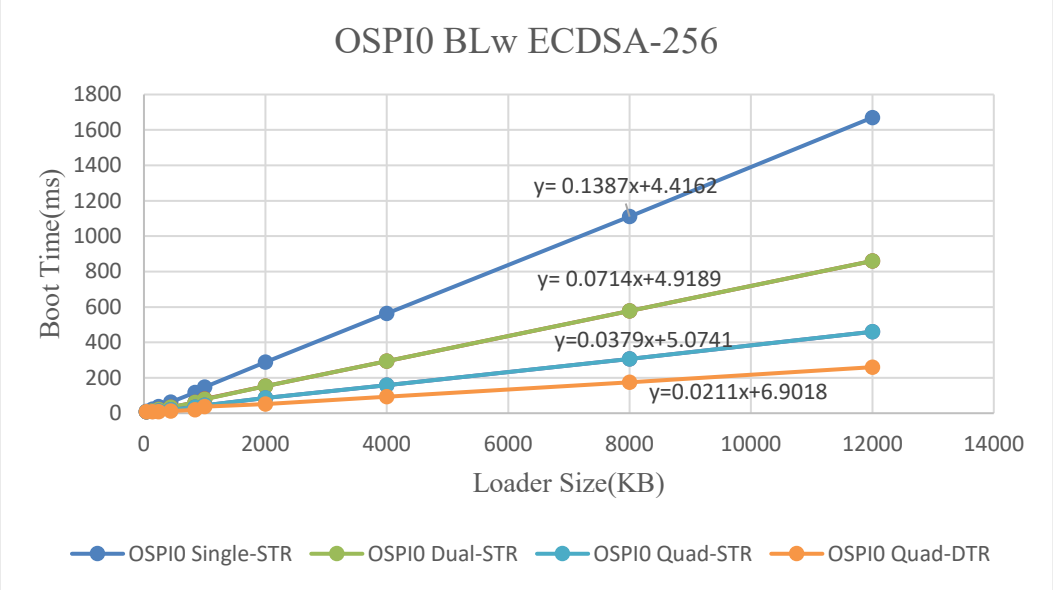


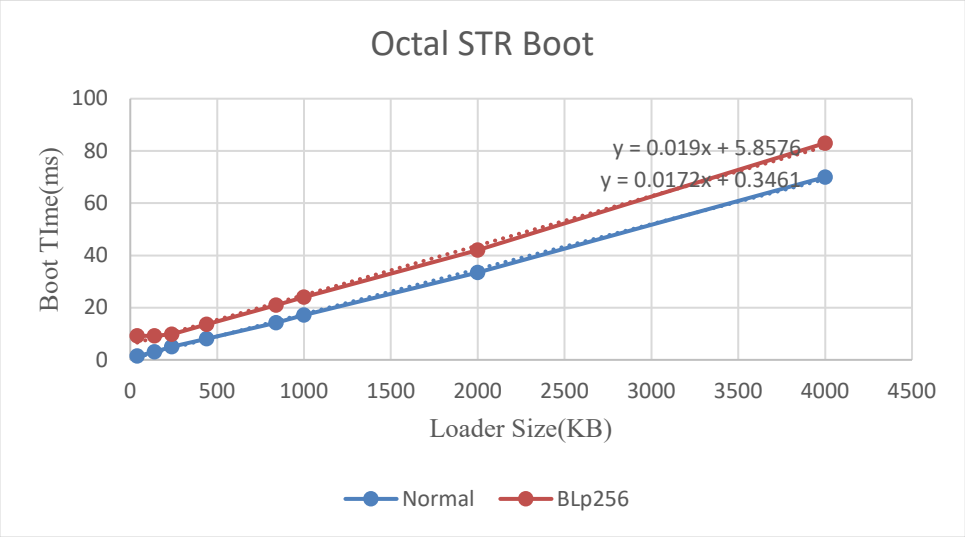
Figure 8: OSPI BLw-256 Boot at 62.5 MHz



These is an extra overhead present in DMA transfer in the SPI secure boot, where the MSIZE setting always remains *two*. The MDMA channels operate on the 2-byte aligned page addresses because of the presence of the 212-byte secure header. This adds an extra delay with each 32-bit DMA transfer. This delay is why an SPI secure boot time is sometimes greater than an OSPI secure boot.

The OSPI Controller for an ADSP-2156x SHARC+ processor does not support the Octal boot by default. The Octal boot can be accomplished using a secondary stage boot, which further improves the boot time. This can be achieved by using a custom boot implementation or a hook function. The octal boot example code in the `7. Boot Time Measurement` folder is included with *EE447Associated Zip File (TTP_BootROM.zip)*[9].

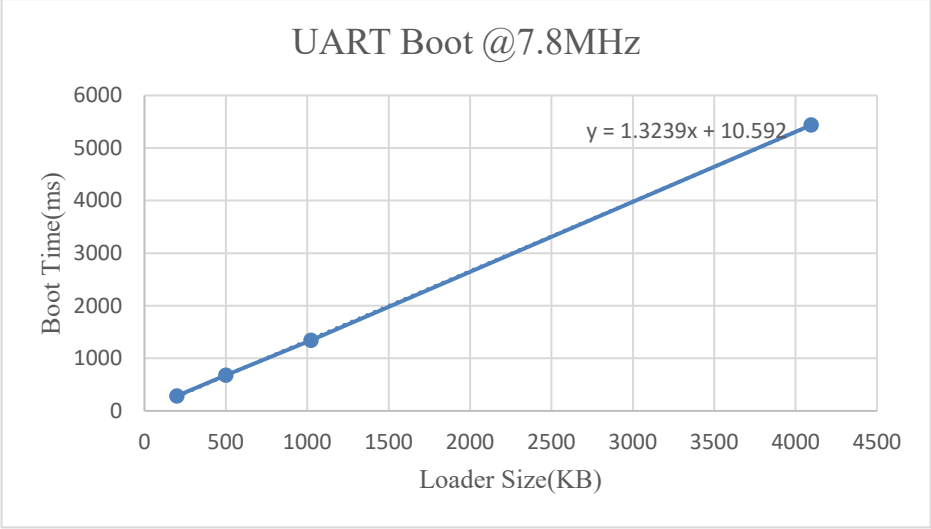
Figure 9: OSPI Octal-STR at 62.5 MHz



UART Host Boot

The UART Host Boot (`SYS_BMODE` 3) for the ADSP-2156x SHARC+ processors is a host boot mode, where the processor receives boot data from a UART host connected to its interface. UART0 is the default booting peripheral. The maximum UART clock speed for the UART boot is 7.8 MHz, achieved by using init code or OTP to configure the CGU for the maximum clock speed (CCLK 1 GHz, SCLK0 125 MHz, and SYSCLK 500 MHz). Because the UART is a slow speed peripheral, boot time for both normal and secure boot is identical. Boot time is primarily dependent on the loading time of the peripheral.

Figure 10: UART Target Boot at 62.5 MHz

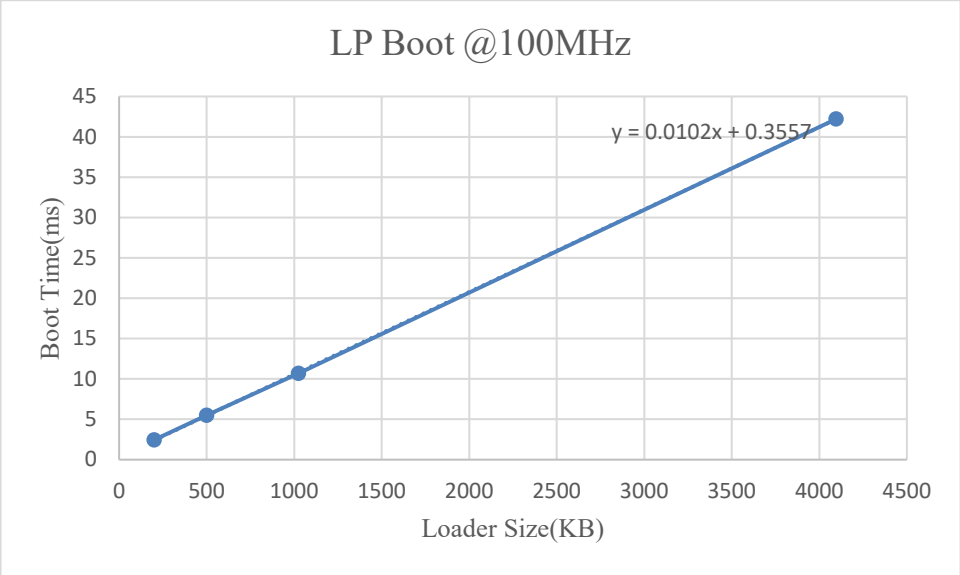


Linkport Host Boot

The Link port boot (`SYS_BMODE` 4) is a host boot mode where the processor receives boot data from an external link port host through link port 0. The DMA controls all transfers from the link port to memory. The maximum operating frequency of the link port is 125 MHz for which the host boot source is responsible for driving with the clock frequency. This is achieved by using init code or OTP to configure the CGU for maximum clock speed (CCLK 1 GHz, SYSCLK 500 MHz, and SCLK0 125 MHz). The link port receiver operates at an asynchronous frequency up to the maximum supported operating frequency.

Figure 11 shows the linear boot time equation for Linkport normal boot at 100 MHz LP CLK is provided.

Figure 11: Linkport Target Normal Boot at 62.5 MHz



Pre-boot Time

As mentioned in earlier section, pre-boot time accounts for the configuration of all system resources prior to starting the boot operation and includes:

- Core initialization
- SPU and SMPU configuration
- Secure debug key processing
- CGU configuration
- DMC configuration
- Fault Configuration
- L1 memory initialization

The pre-boot execution time varies based on the features enabled through OTP programming. See the *ADSP-2156x SHARC+ Processor Hardware Reference*[4] for more information about pre-boot operations. Default pre-boot time for the ADSP-2156x processor without OTP programming is 2.1 ms.

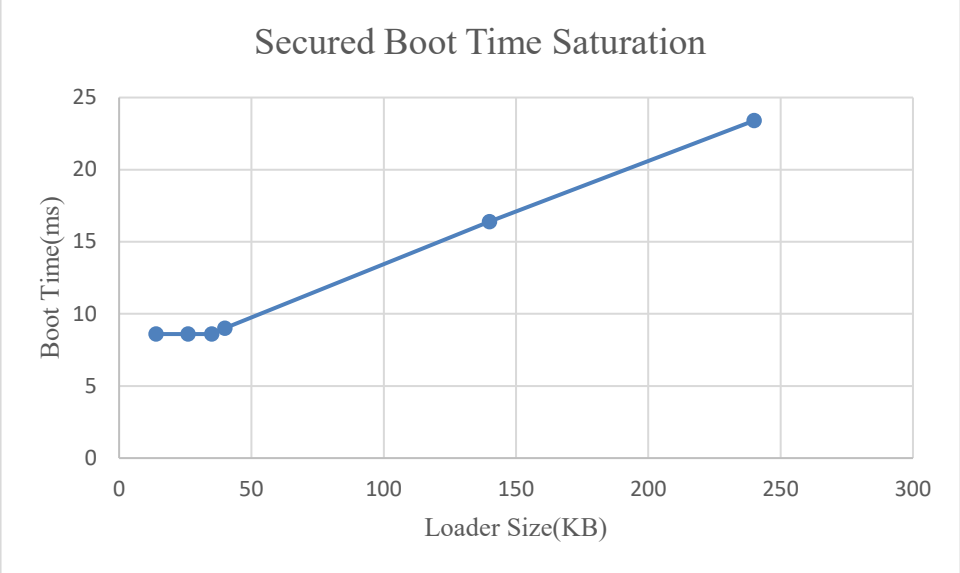
Initcode execution time (when present) is based on the features enabled inside the initblock application and affects the total boot time. Fill blocks can also affect the boot time.

Saturation in Secured Boot Time

The authentication routine consumes a considerable amount of time in the Crypto Engine and plays an key role in determining the total secure boot time. For the ADSP-2156x SHARC+ processor, the Boot ROM must wait for the entire loader stream to load before the Crypto engine can generate the signature from the computed SHA hash digest. The authentication routine needs the signature to complete. Because of the sequential process, the authentication routine becomes somewhat of a “bottleneck” for smaller boot images, which affects the total secure boot time. For a smaller boot image, secure boot time reaches a saturation point (the boot time remains constant for smaller sizes).

Figure 12 shows the SPI2 dual secure boot reaching the saturation point at 40 KB, when booting at clock speeds (CCLK=1 GHz, SYSCLK=500 MHz, and SPI CLK=62.5M Hz). As the crypto engine runs on the SYSCLK clock, the saturation point varies with other clock frequencies and boot modes.

Figure 12: Boot Time Saturation in SPI2 BLp-256 Duel-STR Boot at 62.5 MHZ



References

[1] *ADSP-21566/21567/21569 SHARC+ Single Core High Performance DSP (up to 1 GHz) Datasheet Rev 0, March 2020. Analog Devices, Inc.*

[2] *ADSP-21591/21593/21594/ADSP-SC591/SC592/SC594: SHARC+ Dual-Core DSP with Arm Cortex-A5 Preliminary Data Sheet (Rev. PrD)*

[3] *EE384 Associated Zip File (EE384.zip). Rev 1, September, 2021. Analog Devices, Inc.*

[4] *ADSP-2156x SHARC+ Processor Hardware Reference. Revision 0.3, March 2020. Analog Devices, Inc.*

[5] *ADSP-SC59x/ADSP-2159x SHARC+ Processor Hardware Reference.* Rev 0.0. Analog Devices, Inc.

[6] *CrossCore® Embedded Studio 2.9.0 Software > SHARC® Development Tools Documentation > Loader and Utilities Manual > Loader for ADSP-SC5xx/ADSP-215xx Processors*

[7] *EE336: Secure Booting Guide for ADSP-BF70x Blackfin+ Processors. Rev 1, November 2014. Analog Devices, Inc*

[8] *EE432: Boot Time Estimation for ADSP-SC59x/ADSP-2159x SHARC+ Processors. Rev 1, August 23, 2021*

[9] *EE447Associated Zip File (TTP_BootROM.zip), Rev 1, May 11, 2023.*

Document History

Revision	Description
Rev 1 – May 11, 2023 *by Juganta Saikia &* *Madhumadhi Srinivasan*	Initial Release