



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](mailto:processor.support@analog.com) or [processor.tools.support@analog.com](mailto:processor.tools.support@analog.com) for technical support.

## Booting the ADSP-BF561 Blackfin® Processor

Contributed by Jayanti Addepalli

Rev 1 – May 15, 2007

### Introduction

This EE-Note discusses the booting process for ADSP-BF561 Blackfin® dual-core processors. The available boot modes for silicon revisions 0.2 and beyond are addressed. This document also describes the loader file structure with regard to the booting process. This document is supplemented by code examples verified using VisualDSP++® 4.5 (November update) on an ADSP-BF561 EZ-KIT Lite® evaluation platform.

### Booting Process

*Booting* refers to the process of loading application code and data into the internal and external memories of the Blackfin processor immediately after reset. The code and data are brought in from an external source, which could be a memory device or a host processor, depending on the boot mode configuration. The boot ROM, which occupies the lowest 2 Kbytes (Blackfin memory at address 0xEF000000 – 0xEF0003FF) of internal memory space, includes a bootstrap kernel that contains the configuration settings required for each boot mode.

The `BMODE[1:0]` pins configure the boot mode. These pins are sensed and latched into the system reset configuration register (`SICA_SYSCR`) when the processor is brought out of reset. The bootstrap code reads `SICA_SYSCR` to determine the value of the `BMODE[1:0]` pins. Depending on the selected boot mode, the appropriate code is executed from the boot ROM.

When the `/RESET` signal to the processor is released, core A executes the boot kernel code. The application is loaded from the source into internal and/or external memory. The external memory can be SRAM or SDRAM. Core B is held in the idle state during this time.

The application is expected to be in a defined format, referred to as the *boot stream*. A boot stream is composed of multiple blocks of data and commands. Each block contains header information that indicates what the block is supposed to be (e.g., a zero-fill block, an initialization block, or a final block).

The boot kernel processes the boot stream block-by-block until reaching a block flagged as the last block. The control then jumps to the start of core A instruction memory to begin code execution.

## Boot Modes (Silicon Revision 0.5)

Blackfin processors can boot from a non-volatile memory (flash, ROM, EPROM, etc.) via asynchronous memory bank 0 of the EBIU or from an SPI device (memory or host) via the SPI port. Figure 1 shows the data flow during the boot process. The numbered arrows indicate the sequence of events.

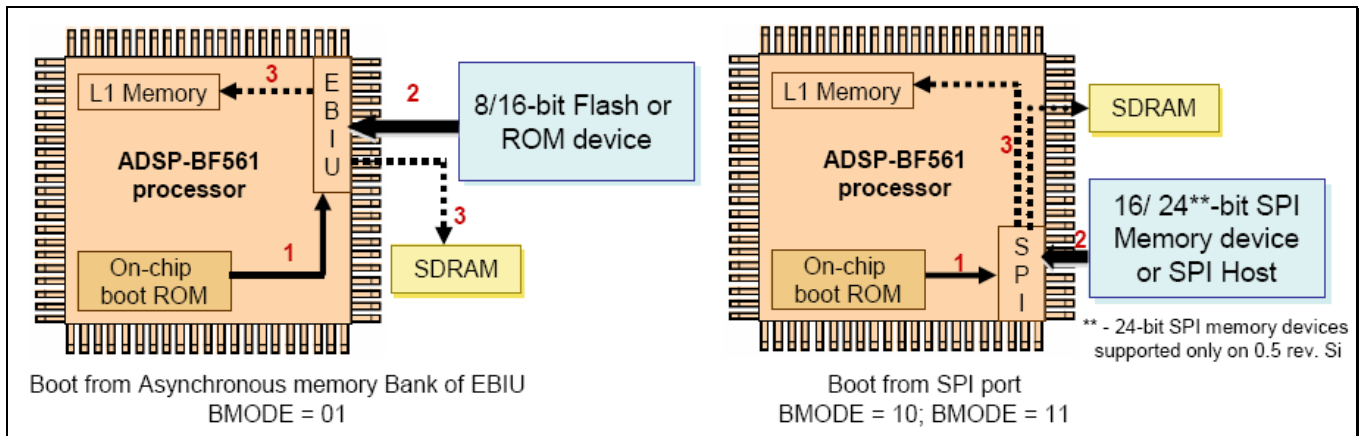


Figure 1. Data flow for the booting process on ADSP-BF561 processors

Table 1 lists the boot modes determined by the state of the BMODE[1:0] pins after reset. On the EZ-KIT Lite board, positions 1 and 2 of the SW3 switch set the boot mode. The boot modes for the silicon revisions before revision 0.5 are listed in Appendix 1 – Boot Modes on Older Silicon Revisions.

BMODE[1:0]	SW3 Position 2, Position 1	Description
00	ON, ON	No boot mode - Executes from external 16-bit memory connected to ASYNC Bank 0 (bypass boot ROM)
01	ON, OFF	Flash/ROM boot mode - Boots from 8/16-bit flash/PROM (default mode on an EZ-KIT Lite board)
10	OFF, ON	SPI slave mode - Boots from an SPI host
11	OFF, OFF	SPI master mode - Boots from a 16/24-bit addressable SPI memory

Table 1. ADSP-BF561 Blackfin processor boot modes (0.5 silicon)

## Loader File Structure

The loader utility (`elfloader.exe`) of VisualDSP++ parses input executable files (`.DXE`) to create a loader file (`.LDR`); refer to Figure 2. It segments the application into multiple blocks and creates header information for each block. Multiple `.dxe` files are combined into a single loader file (boot stream).

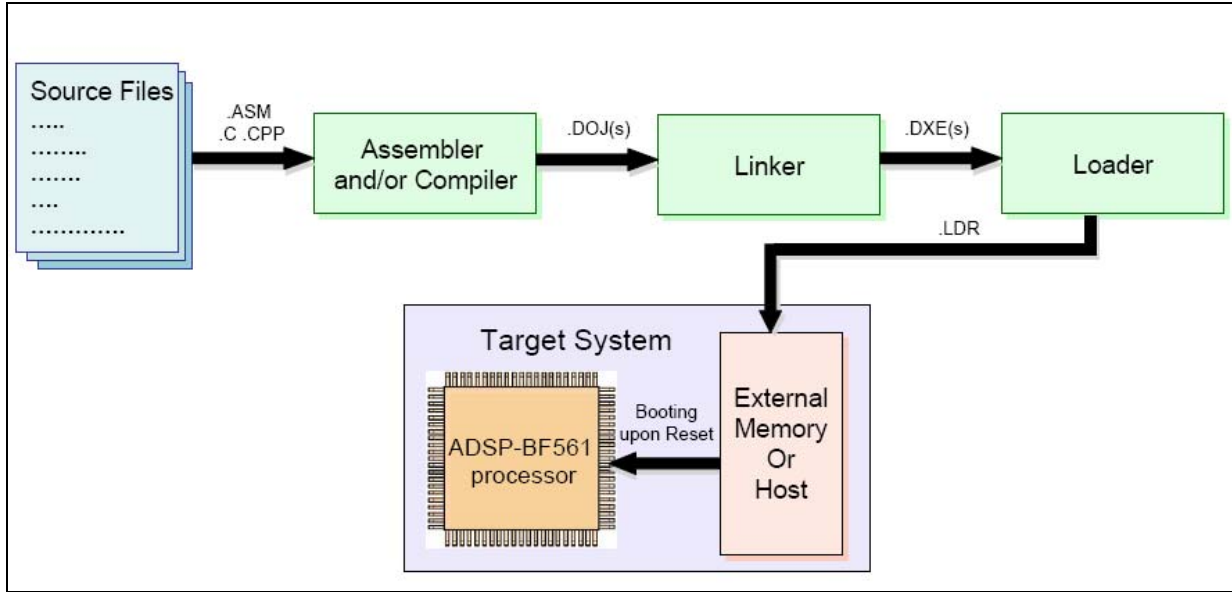


Figure 2. ADSP-BF561 stand-alone system

The ADSP-BF561 processor boot stream begins with a 4-byte global header. The global header contains a signature in the upper 4 bits that prevents the boot ROM from reading a boot stream from a blank device. Figure 3 provides a bit-by-bit description of the global header.

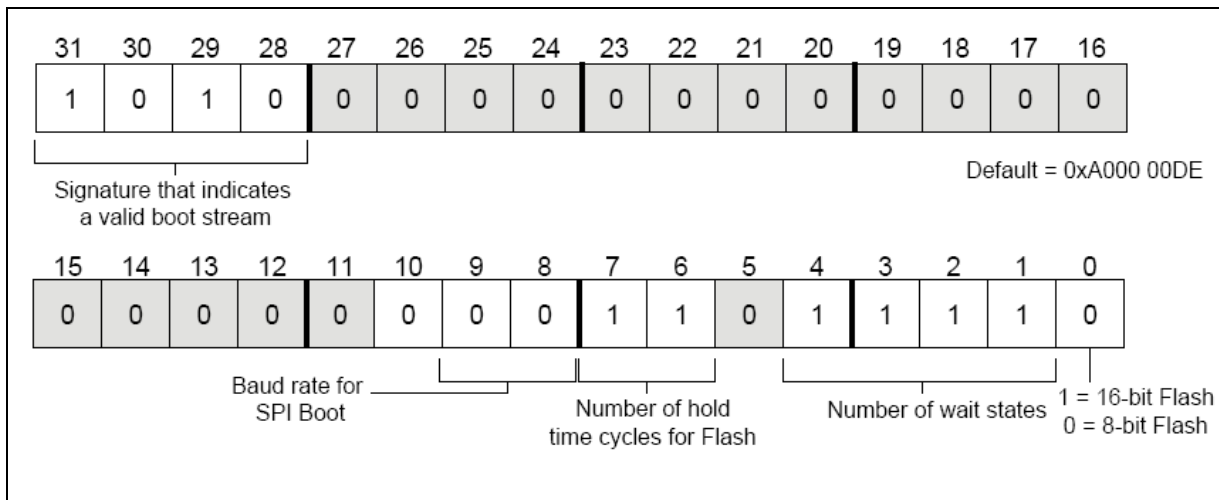


Figure 3. Global header structure

Each block in the loader file begins with a 10-byte block header. Each header contains a start address and count for the data block, followed by a flag word.

- ADDRESS: Target address to which the block will be booted within memory (4 bytes)
- COUNT: Number of bytes in the block (4 bytes)
- FLAG: Block type and control commands (2 bytes)

The individual bits of the flag word are shown in [Figure 4](#). After reset, the headers are read and parsed by the on-chip boot ROM. The boot stream is processed block-by-block, and payload data is copied to destination addresses. The destination can be on-chip L1 or L2 memory or off-chip SRAM/SDRAM.

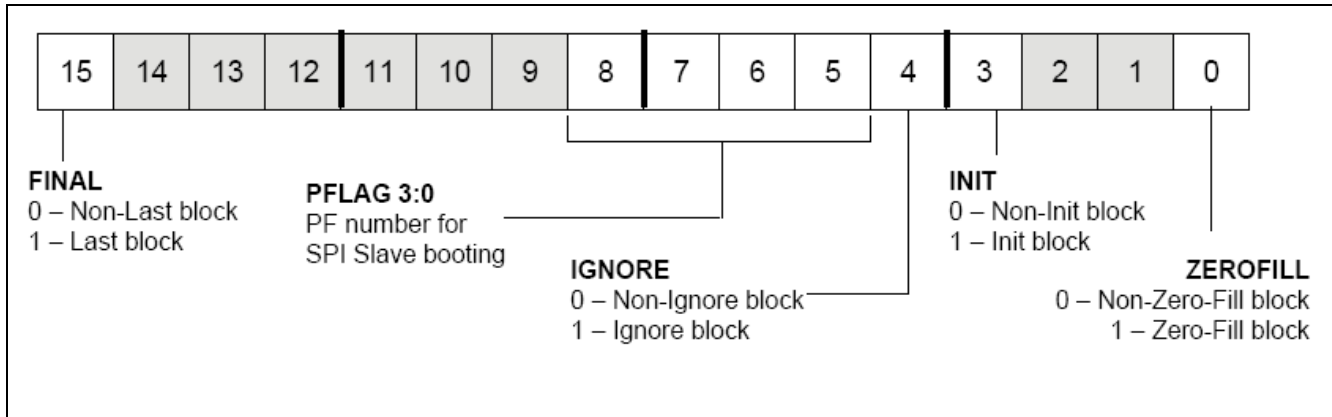


Figure 4. Flag word structure

A brief description of the different fields in the flag word follows:

- **ZEROFILL:** Specifies that the block is a buffer with zeros. ZEROFILL blocks have no payload data. The boot ROM fills COUNT bytes of memory with zeros, starting from the given ADDRESS. This helps to generate a condensed loader file for applications with large zero buffers.
- **INIT:** Specifies that the block is an initialization block. It executes before the actual application boots over it. When the on-chip boot ROM detects an init block, it boots the block into internal memory and makes a CALL to it. Refer to [Initialization Code](#) for details.
- **IGNORE:** Indicates a block that is not booted into memory. It instructs the boot ROM to skip COUNT bytes of the boot stream.
- **PFLAG:** Used for SPI slave boot mode (BMODE = 10). PFLAG indicates the PF<sub>X</sub> used for the host wait (HWAIT) signal from the Blackfin processor to the master SPI device. This value can be between 1 and 15 (0x1 – 0xF).
- **FINAL:** Indicates that the boot process is complete after this block is processed. The boot ROM jumps to the start of core A L1 instruction memory (0xFFA00000) after processing a FINAL block. The processor continues to remain in supervisor mode (servicing the lowest-priority interrupt - IVG15) when it jumps to internal memory for code execution.

A .dxe file count block follows the global header in the loader file. It contains a 32-bit byte count for the first .dxe file of the boot stream. The “ignore bit” is set in its flag word so that the boot ROM does not try to load this block into memory. It is encapsulated by a 10-byte block header like other blocks, though it contains only a byte count. The address field is irrelevant for this block since it is not copied into memory.

The other blocks of the first .dxe file and a similar structure for the second .dxe file follow the .dxe count block. In a dual-core project, two .dxe files are generated: p0.dxe for core A and p1.dxe for core B. The structure of the boot stream for a dual-core application is shown in [Figure 5](#).



The boot ROM does not support booting to core A (0xFFB00000 - 0xFFB00FFF) or core B (0xFF700000 - 0xFF700FFF) scratchpad memories. Data that must be initialized prior to runtime should not be placed in scratchpad memory. If booting to scratchpad memory is attempted, the processor hangs within the on-chip boot ROM.

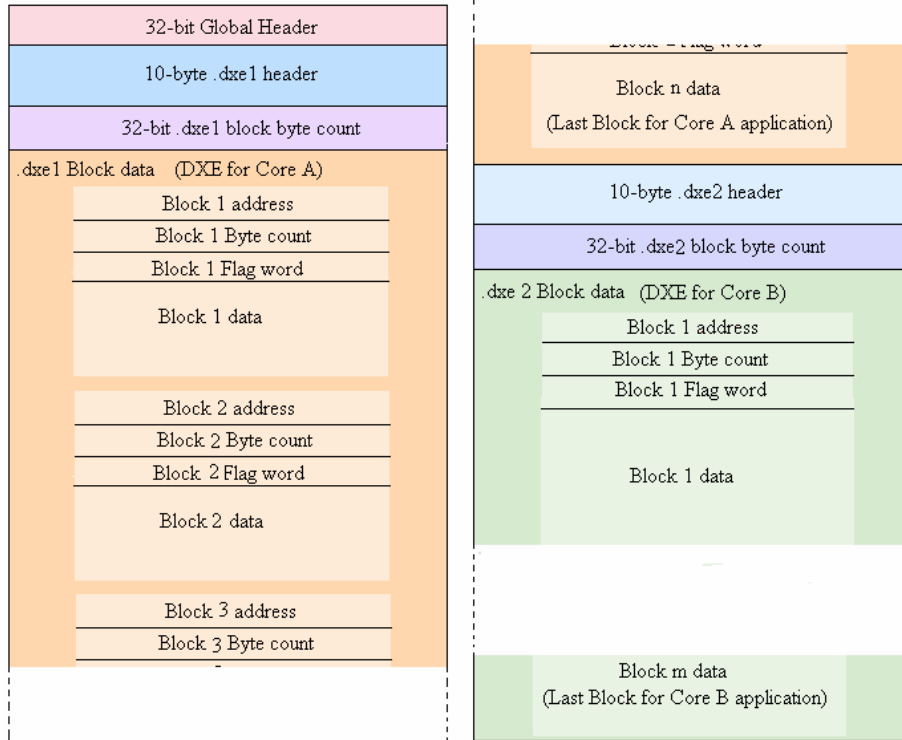


Figure 5. Boot stream structure for a dual-core application

## Booting to Both Cores

There are many nuances associated with the dual-core boot process, including specific loader switches, the need to unlock core B, the process for changing the PLL, and using the flash programming utility.

### Loader Switches: `-NoFinalTag`, `-nosecondstagekernel`

The loader can be configured to automatically append the core B boot stream to the end of the core A boot stream in such a way that the boot ROM recognizes it as a single boot stream. The `final` tag is present only in the final block of the core B boot stream. This can be done using the loader switch `-NoFinalTag`.

The `-nosecondstagekernel` switch ensures that VisualDSP++ does not include the second-stage loader (SSL) by default. The tool currently includes the SSL even when its use is not specified in the project options. The usage of these loader switches is shown in Figure 6.

These tags are supported beginning with the November update of VisualDSP++ 4.5.

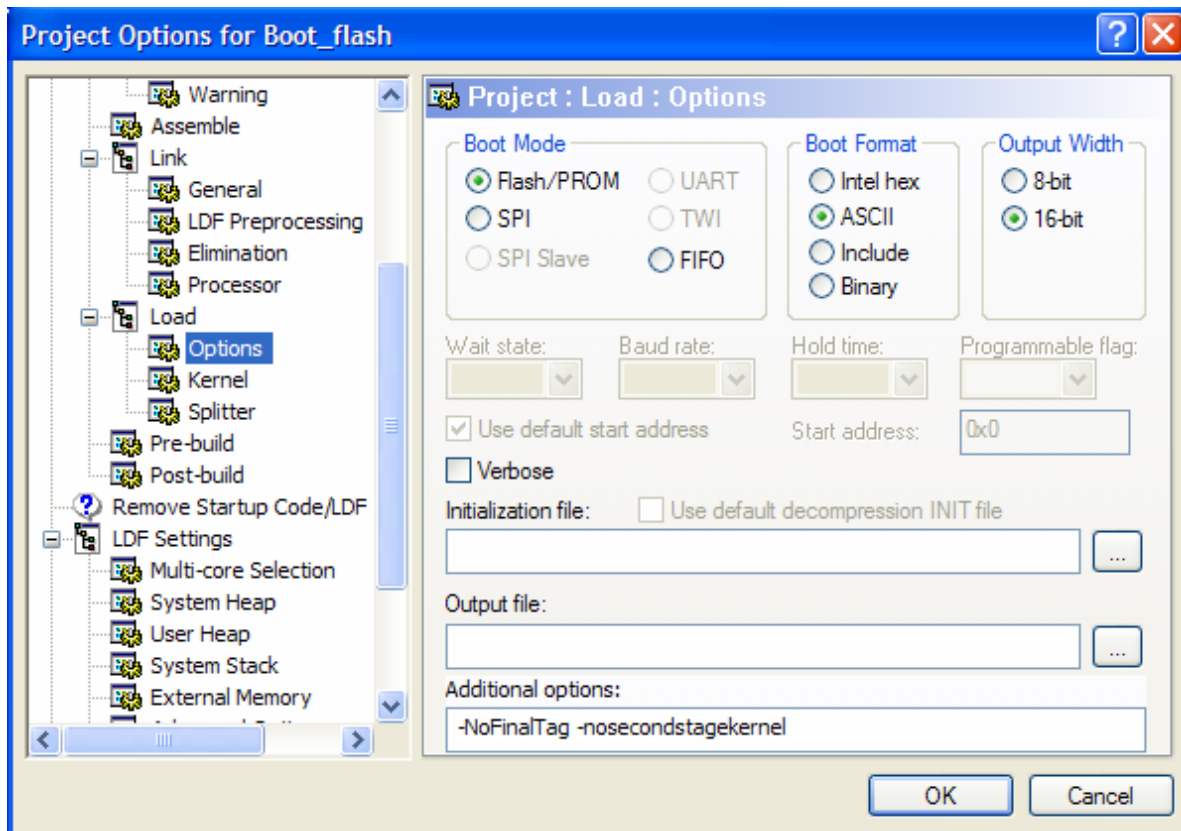


Figure 6. Using loader switches to boot a dual-core application without the SSL

With separate executables for each core, older versions of VisualDSP++ (prior to the November update of VisualDSP++ 4.5) required the SSL to boot both cores. This is discussed in [Appendix 2 – Booting Using the Second-Stage Loader](#). The SSL is not required starting from the November Update of VisualDSP++ 4.5. Init code can be used for initialization functions. Refer to [Initialization Code](#) for details.

## Unlocking Core B

By default, core A executes the boot kernel code; core B is in the idle state during this time. Core B remains in the idle state until the user code executed by core A clears the `COREB_SRAM_INIT` bit in `SICA_SYSCR`. When set, this bit prevents all core B interrupts and wake-up events from being serviced; thus, core B cannot come out of idle. When this bit is cleared by core A, core B is released from idle and begins executing from the start of its L1 instruction memory space (`0xFF600000`). Note that once `COREB_SRAM_INIT` is cleared, setting it again has no effect. That is, you cannot lock core B by setting this bit once it has been unlocked. While in a VisualDSP++ debug session, it is not required to clear this bit since the in-circuit emulator (ICE) unlocks core B automatically through the JTAG port.



When booting a dual-core application, the core A application code must unlock core B by clearing the `COREB_SRAM_INIT` bit in the `SICA_SYSCR` register.

## Changing PLL Ratios

In order to change the PLL settings, both cores are required to be in the idle state. Since core B is in the idle state immediately after booting, core A can change the PLL settings for the desired core and system clock frequencies and execute an `IDLE` instruction. After the PLL is locked to the new frequency, it can wake up core A, which in turn can unlock core B.

## Using the Flash Programmer Utility

The Flash Programmer utility that comes with VisualDSP++ is core-specific (core A) and must be loaded into core A in order to operate correctly. The Flash Programmer relies on the user to set the correct core focus.



To set up the correct core, select core A in the Multiprocessor window before opening the Flash Programmer user interface. If core B is selected, the Flash Programmer will not load properly, and an error will be generated.



When booting a single-core application, specify the `-nosecondstagekernel` tag in the Additional Options field (Loader page) of the Project Options dialog box.

## Initialization Code

In most applications, certain tasks must be performed before the application starts utilizing off-chip memory interfaces. These tasks are done inside an init block in the loader file. As an init block, the code contained within is loaded into core A L1 instruction memory and is then executed immediately to take care of these tasks before carrying on with the boot sequence to move code and data from the boot source and into external memory.

For example, if the boot image consumes external memory, the SDRAM controller must be initialized before the boot ROM attempts to load code and data over the EBIU into external memory. While in a VisualDSP++ debug session, the SDRAM controller can be configured automatically by selecting the `Use XML reset values` option of the Target Options dialog box (Settings -> Target Options). This is

the default setting in VisualDSP++, so SDRAM is always properly configured for use on an EZ-KIT Lite board. When booting the processor, there is no such provision. It therefore becomes essential to initialize the SDRAM memory controller before any instructions or data are loaded into SDRAM.

If the time taken by the boot process is critical to the system, the PLL register settings may have to be changed before the application starts booting. Also, when booting from a flash/ROM device, the process is faster if only the required wait states were set in the EBIU configuration registers. By default, the slowest configuration is used by the boot ROM. If the PLL is changed in the init block, this will have impact on how the SDRAM controller is configured as well. Similarly, the SPI baud rate settings for booting via the SPI interface can be altered to speed up the processor boot time.

In both the PLL and SDRAM examples previously mentioned, init code or an SSL must be used for the tasks that must be executed prior to the application code itself being loaded. The *elfloader* utility converts the init code into an init block and prepends it to the application executable(s) in the loader file. The init blocks are identified by a bit in the flag word of the 10-byte block header. When the boot ROM encounters this block in the boot stream, it loads the block and executes it immediately. An `RTS` instruction must be executed at the end of the init code to return control to the boot ROM. The initialization blocks must save and restore registers and return to the boot ROM so that the boot ROM can load the rest of the blocks in the boot stream. The init block code in L1 instruction memory is then overwritten by the application code when it is subsequently loaded.

The init block can also be used to force the boot ROM to load a specific `.dxe` file from the external memory device. The initialization block can manipulate the `R0` or `R3` registers, which the boot ROM uses as the external memory pointers for flash/PROM or SPI memory boot, respectively. After the processor returns from the execution of the init block, the boot ROM continues to load blocks from the location specified in the `R0` or `R3` register, which can be any `.dxe` file in the boot stream.

An example project for booting from flash memory that uses init code to configure the SDRAM controller registers to their default values can be found in the software that accompanies this EE-note. The use of the second stage loader is documented in *Managing Multiple DXEs on ADSP-BF561 Blackfin Processors (EE-272)*<sup>[5]</sup>.

## Boot Modes

This section discusses the details of boot modes, which are determined by the `BMODE[1:0]` pins. In addition to the three boot modes, the processor also supports the No-Boot mode (`BMODE = 00`), in which the boot ROM is bypassed and the application runs directly from flash memory bank 0 (at address `0x20000000`).

### 8/16-Bit Flash/PROM Boot (`BMODE[1:0] = 01`)

Flash memory is connected to asynchronous memory bank 0 of the EBIU. Since the EBIU is 16 bits wide, an 8-bit flash/PROM device will occupy only the lower 8 bits of the data bus. (`D[7:0]`). The pin-to-pin connections between the Blackfin processor and a 16-bit flash device are shown in [Figure 7](#), and the connections for an 8-bit device are shown in [Figure 8](#).



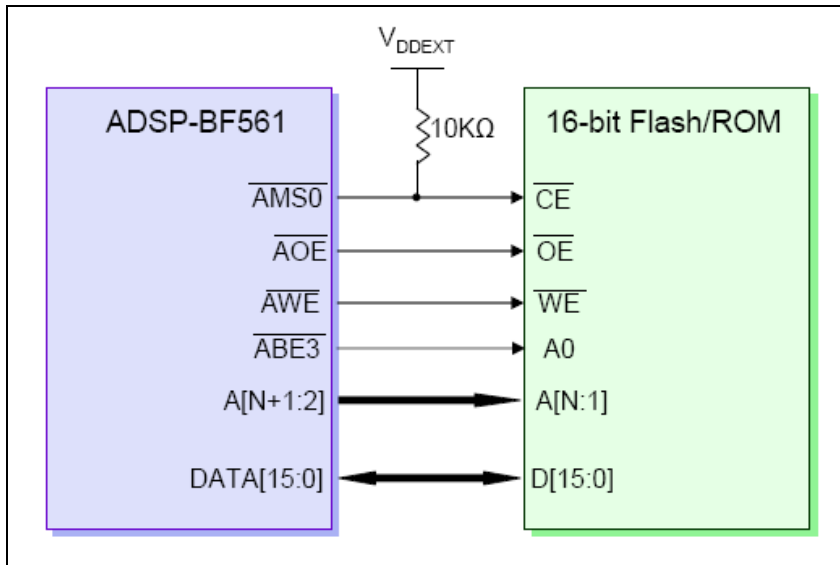


Figure 7. Interfacing an ADSP-BF561 Blackfin processor to a 16-bit flash/PROM device

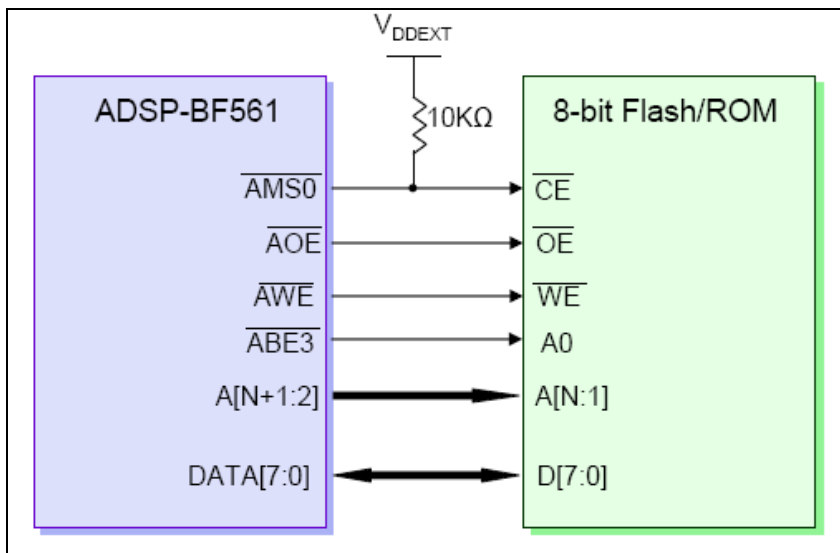


Figure 8. Interfacing an ADSP-BF561 Blackfin processor to an 8-bit flash/PROM device

The processor performs a 16-bit fetch regardless whether an 8-bit device or a 16-bit device is connected. Therefore, when an 8-bit device is connected, the upper eight bits that are received by the Blackfin processor are ignored. The lower eight bits are then placed consecutively in internal memory.

There are no multiplexers inside the processor that read the lower eight bits (connected to external memory) and move them to the upper eight bits of the 16-bit word. Therefore, packing into 16-bit words is not supported when connecting 8-bit devices to the external bus.

Figure 9 shows a description of the first few bytes of the loader file. This .LDR file was created in ASCII format for booting from a 16-bit flash device.

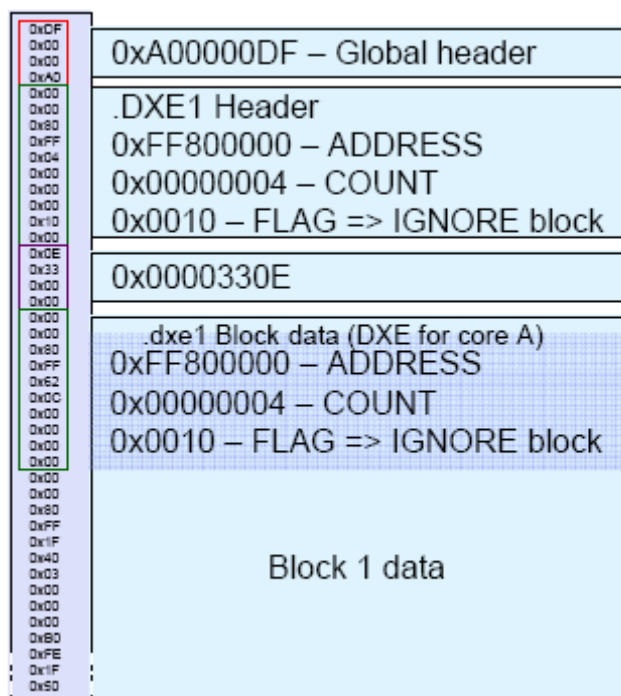


Figure 9. Example ASCII loader file

The first four bytes of the loader file constitute the global header. The information about whether it is an 8-bit flash boot or a 16-bit flash boot is contained here. With the exception of the global header, the loader file created for an 8-bit flash looks exactly the same as that in Figure 9. The global header for an 8-bit flash device would be 0xA00000DE. The on-chip boot ROM interrogates bit 0 of this header to determine whether an 8-bit or a 16-bit device is connected.

The 8/16-bit flash boot routine located in the boot ROM memory space has all configuration settings set for the slowest device possible (4-cycle setup time; 15-cycle R/W access times; 3-cycle hold time). The boot kernel assumes that asynchronous memory bank 0 is enabled with 16-bit packing.

The example code in the associated ZIP file demonstrates this boot method. Core A blinks one row of LEDs on the EZ-KIT Lite board (rev.2.0), while core B blinks the other row of LEDs. On the board, the 8 MB of flash memory is organized as 4M x 16-bit and is mapped into the processor's ASYNC bank 0.

#### SPI Slave Boot by a Master Host (BMODE[1:0] = 10)



This boot mode is NOT supported on ADSP-BF561 processor silicon prior to revision 0.5.

The processor is configured as a slave device in this mode. A host is used to transmit the loader file. The host does not require knowledge of the details of the file. It must only be configured to transmit one byte at a time from the loader file (ASCII format). Figure 10 shows the pin connections required for this boot mode.

A user-defined programmable flag pin is an output on the Blackfin processor and an input to the SPI host device. This flag allows the processor to hold off the host device from sending data during certain sections of the boot process. When this flag is de-asserted (low), the host can continue to send bytes to the processor.

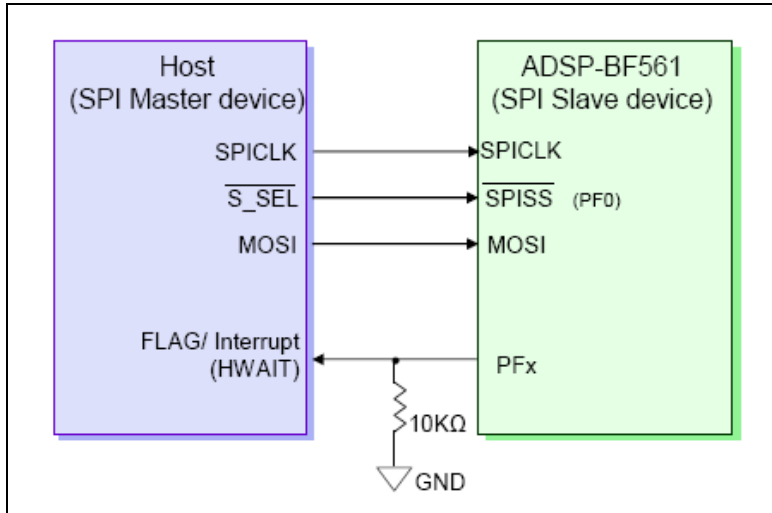



Figure 10. Interfacing an ADSP-BF561 processor to an SPI host device (SPI slave boot mode)

In the above setup,  $PF_x$  is the host wait ( $HWAIT$ ) signal from the Blackfin processor to the master device. This is used to “hold off” the host during certain times in the boot process (i.e., during init code execution and zero-fill blocks). When the  $PF_x$  signal is asserted (high), the host device will stop sending bytes to the Blackfin processor. When de-asserted (low), it will resume sending bytes from where it stopped.

A pull-down resistor on the  $PF_x$  pin is necessary to ensure that the master will continue to send bytes until the first block is processed by the slave.

 The host must ensure that the Blackfin processor is out of reset before transmitting the loader file. All bytes sent before the processor comes out of reset will be lost, and the boot sequence will fail.

The VisualDSP++ 4.5 (November update) user interface does not support this boot mode (later updates will). Therefore, the loader file must be created using the command line. One method of specifying the command-line argument is via the Post-build page of the Project Options dialog box as shown in Figure 11.

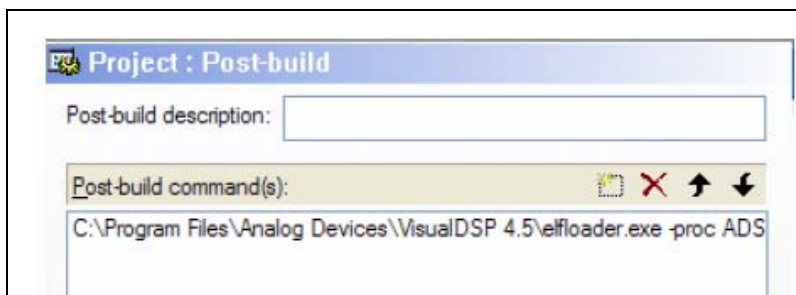


Figure 11. Using the post-build capability in VisualDSP++ to specify a command-line argument

An example of the command to be used is shown in Listing 1.

```
C:\Program Files\Analog Devices\VisualDSP 4.5\elfloader.exe -proc ADSP-BF561
.\Debug\p0.dxe .\Debug\p1.dxe -pflag 4 -b spislave -include -Width 8 -o
.\Debug\SPI_Slave.ldr -si-revision 0.5
```

Listing 1. Command to generate a loader file for SPI slave boot mode

The above command generates the output loader file, `SPI_Slave.ldr`, in the `Debug` folder of the project.

The `PFx` pin used as the `HWAIT` signal is selected via the `-pflag` switch in the command line, and it is embedded within the loader file. In the command shown above, `PF4` is being used. The `elfloader` utility embeds this number in the `PFLAG` bit field (bits [8:5]) of the `FLAG` word in each 10-byte header.



If the `-pflag` switch is not used in the command line, the default value of the `PFLAG` field in the `FLAG` word will be 0, indicating that `PF0` is assumed as the `HWAIT` signal to the host. Since `PF0` is the `/SPISS` pin, which is needed for a successful SPI slave boot, the `-pflag` switch must always be used and must specify a value other than 0.

Figure 12 shows the transmission of the first few words of the loader file. An ADSP-BF533 Blackfin processor was used as the host to send the loader file, which is included in the associated file.

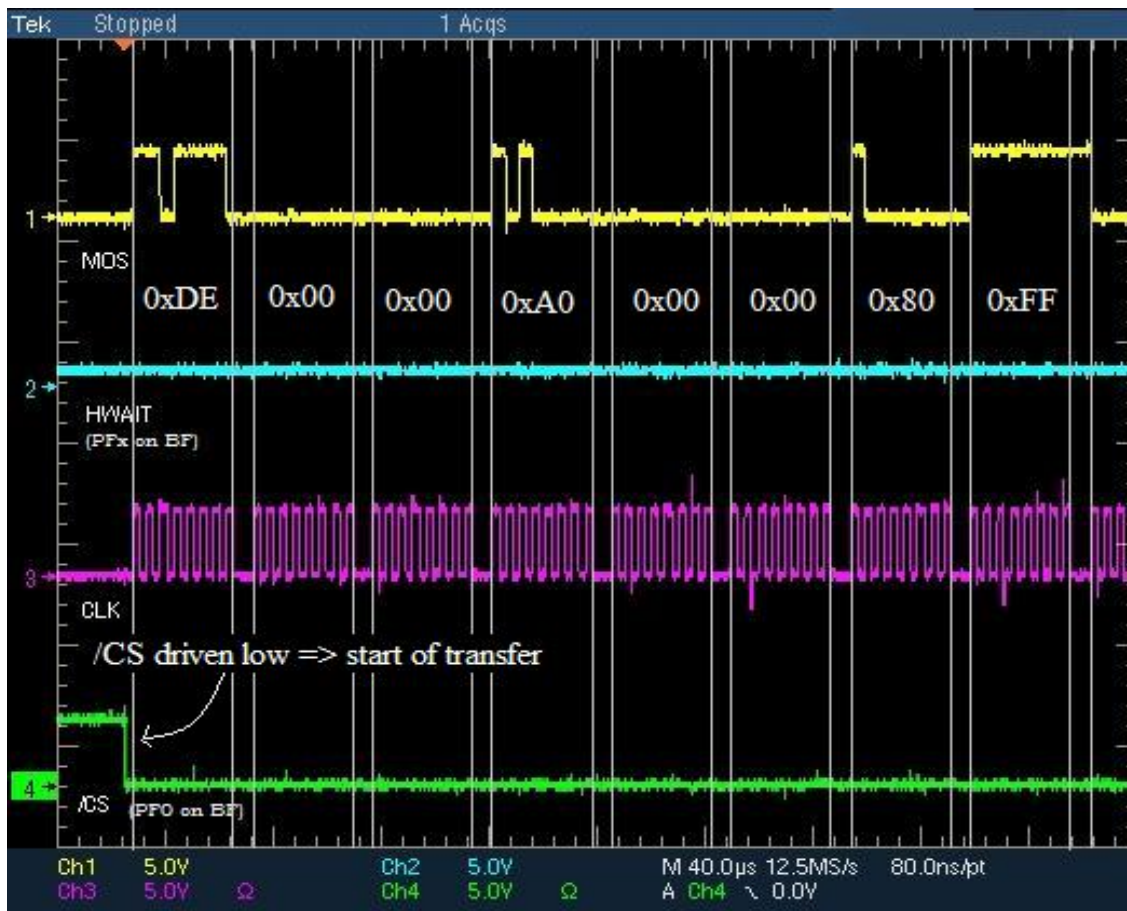


Figure 12. Beginning of SPI slave boot sequence

The HWAIT signal is driven high to signal the host to pause the process of sending bytes from the loader file. This is done when the init code is being executed or when zero-fill blocks are being processed. An example of the HWAIT signal being driven high during the processing of a zero-fill block is shown in Figure 13.

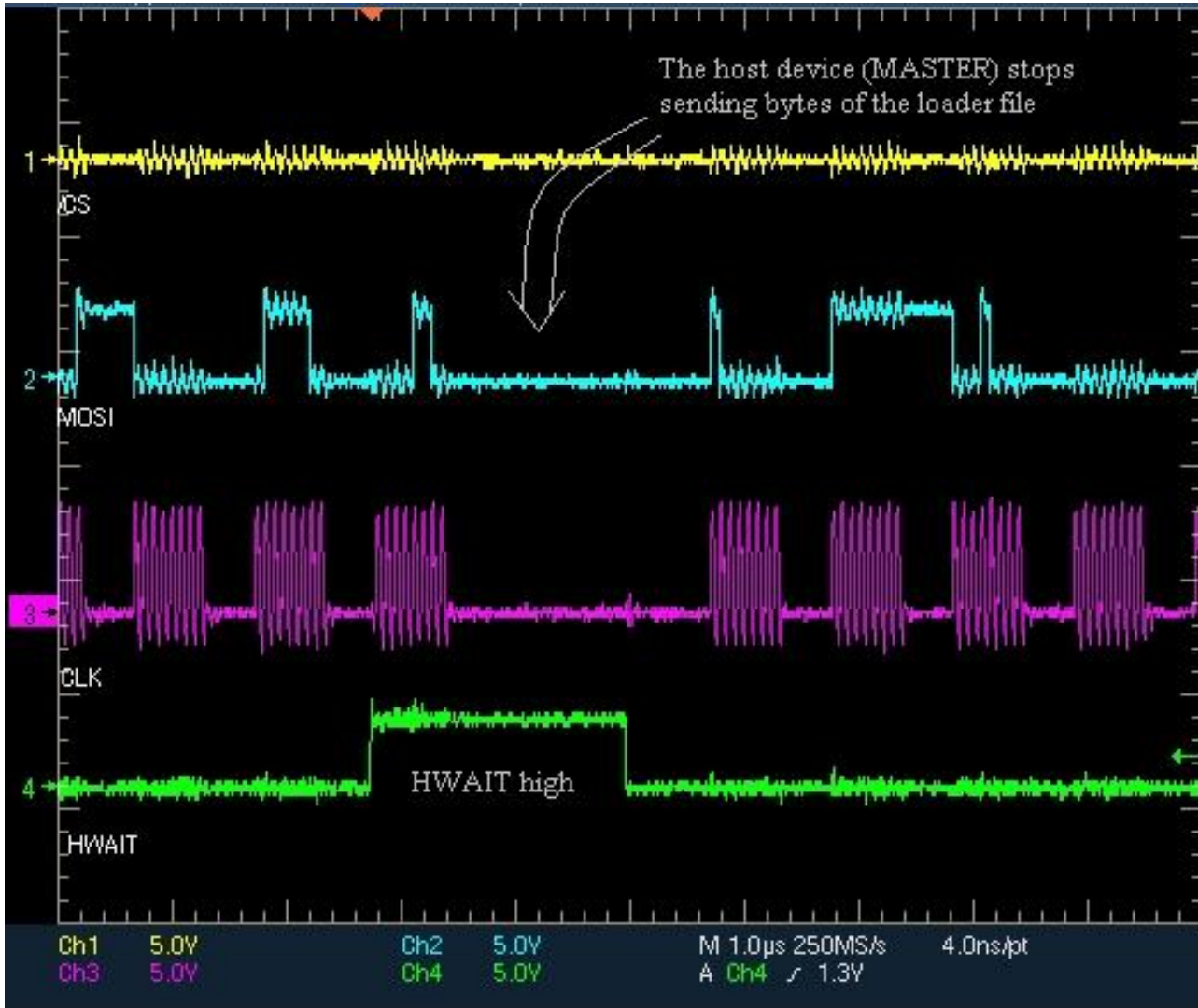


Figure 13. HWAIT signal driven high during the processing of a zero-fill block

### SPI Master Boot from a Memory Device (BMODE[1:0] = 11)

In this mode, the processor is configured as an SPI master, and an SPI memory device is used as the slave. Figure 14 shows the interface between an ADSP-BF561 Blackfin processor and an SPI memory device.

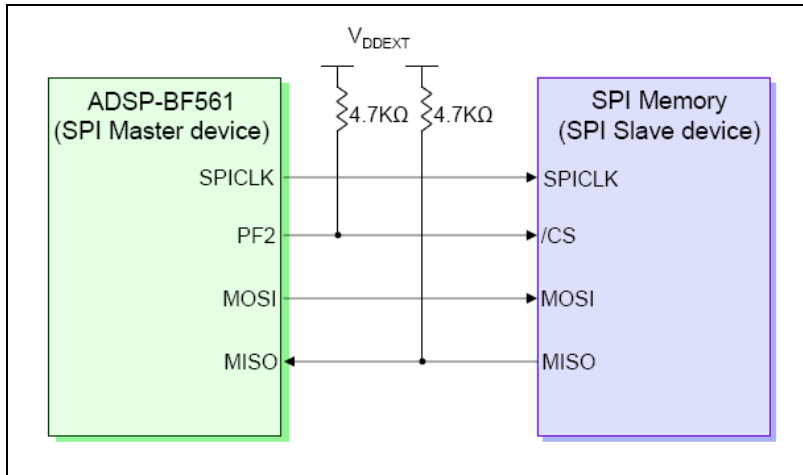


Figure 14. Interfacing an ADSP-BF561 processor host to an SPI slave memory device (SPI master boot mode)



A pull-up resistor on MISO is required for this boot mode to work properly when booting from a 24-bit addressable device. This is because the boot ROM expects a 0xFF on the MISO pin if the device does not respond to the command sent by the processor.

A pull-up resistor on the /CS line is required because a high-to-low transition on the /CS pin is necessary for most SPI memory devices to start a valid instruction after power on.

The EZ-KIT Lite board has a pull-down resistor on the MISO pin. This creates a voltage-divider circuit if the MISO pin is pulled high for booting a 24-bit addressable device in this mode. Remove this resistor if booting from a 24-bit addressable SPI device.

Standard SPI memory devices that are 16-bit addressable are supported by this interface for all revisions of the ADSP-BF561 processor. Standard 24-bit addressable devices are supported only on the 0.5 revision processor.



24-bit addressable SPI memory devices are NOT supported on ADSP-BF561 processor silicon revisions 0.3 and older.

Atmel SPI DataFlash devices are also supported by this boot mode. These Atmel DataFlash devices are available in A-, B-, C-, and D-series. SPI master booting can be accomplished on the D-series of these devices only when they operate in the “power of 2” mode of addressing. This mode is not the default, so the device must be programmed by sending a command word to a one-time programmable bit of a register in the memory device. This should be done in a predefined programming sequence followed by a power cycle, details for which are available in the device’s data sheet. Once the programming is completed, the device can be used like any other SPI device for booting purposes.



Atmel D-type DataFlash devices must be programmed to the “power-of-2” addressing mode before they can be used to boot Blackfin processors.

Standard 16/24-bit addressable SPI memories are those that take in a read command byte (0x03) followed by two address bytes (for 16-bit addressable SPI memories) or three address bytes (for 24-bit addressable SPI memories). The general-purpose flag pin 2 (PF2) is used as the SPI chip select signal. After the correct read command and address are sent, data stored in the memory at the selected address is shifted in on the MISO pin. Data is sequentially read from that address with continuing clock pulses.

#### *SPI Memory Detection Routine*

Since this boot mode supports various SPI memories, the boot ROM must detect the memory device that is connected by following a defined sequence to address the SPI device. Because the SPI memory device will not respond until it is properly addressed, the following sequence achieves this goal:

1. A read command (0x03) is sent on MOSI, and a dummy read of the data from MISO is issued.
2. An address byte (0x00) is sent on MOSI, and a dummy read of the data from MISO is issued.
3. A second address byte (0x00) is sent on MOSI, and a dummy read of the data from MISO is issued.
4. A third byte (0x00) is sent (address byte if a 24-bit device is connected or dummy byte if a 16-bit device is connected) on MOSI, and the incoming byte on MISO is checked for anything other than 0xFF. An incoming byte other than 0xFF means that the SPI memory has responded after two address bytes, meaning that a 16-bit addressable SPI memory device is connected.
5. If the incoming byte is 0xFF, the on-chip boot ROM sends a dummy byte (0x00) on MOSI and checks whether the incoming byte on MISO is anything other than 0xFF. An incoming byte other than 0xFF means that the SPI memory has responded after three address bytes and a 24-bit addressable SPI memory device is connected.

If the incoming byte is 0xFF (meaning no devices have responded), the on-chip boot ROM assumes that one of the supported Atmel DataFlash devices are connected. These DataFlash devices have a different read sequence than the one described above for standard SPI memories. The on-chip boot ROM determines which of the Atmel DataFlash memories are connected by reading the status register. AT45DB041, AT45DB081, AT45DB161, AT45DB642 and AT45DB1282 devices are recognized by the boot ROM. The main difference between these DataFlash devices is the number of bytes per page. AT45DB642 and AT45DB1282 devices have 1024 bytes/page; AT45DB161 has 528 bytes/page; AT45DB041 and AT45DB081 devices have 256 bytes/page.

The interface between a 24-bit addressable Micron M25P80 SPI device and the ADSP-BF561 processor is shown in [Figure 15](#).

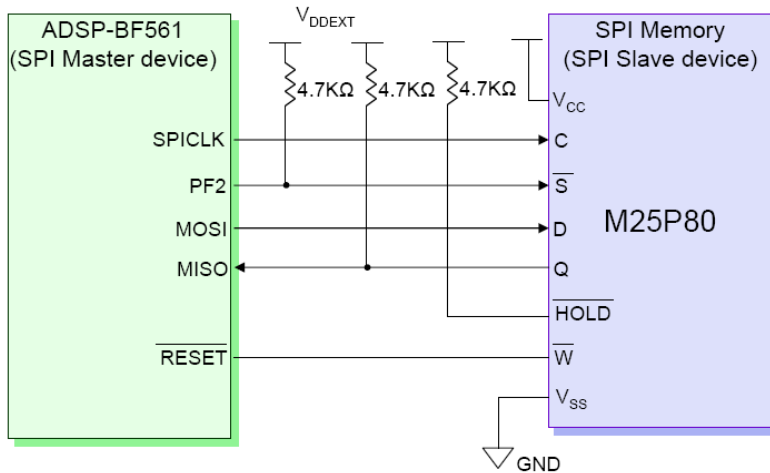


Figure 15. Interfacing an ADSP-BF561 Blackfin processor to a 24-bit addressable M25P80 device

The SPI baud rate register is set to 133, which results in a 203 kHz baud rate when the system clock is running at 54 MHz (as governed by the baud rate equation:  $54 \text{ MHz} / (2 * 133) = 203 \text{ kHz}$ ). Figure 16 shows the boot sequence for the SPI master boot mode using a Micron M25P80 device.

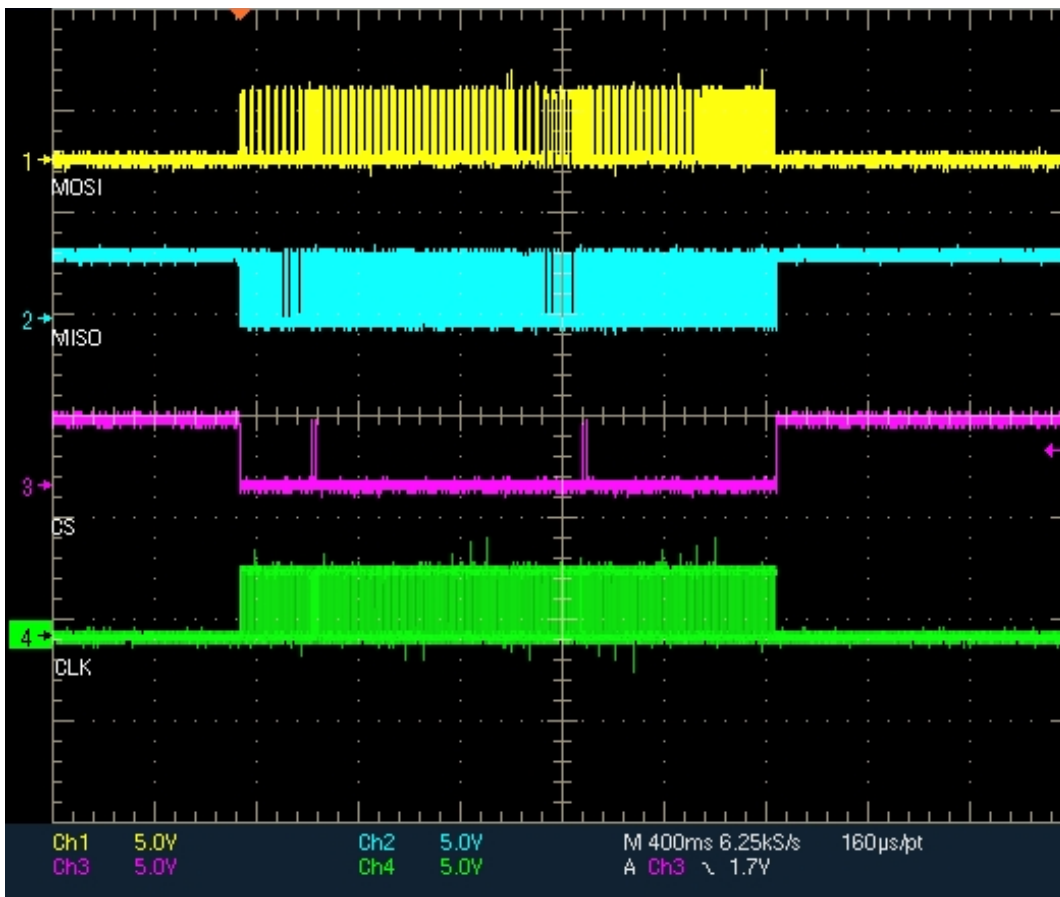


Figure 16. Timing diagram for SPI master boot mode



Initially, the on-chip boot ROM determines the type of memory connected. Figure 17 shows the detection routine for a 24-bit addressable device.

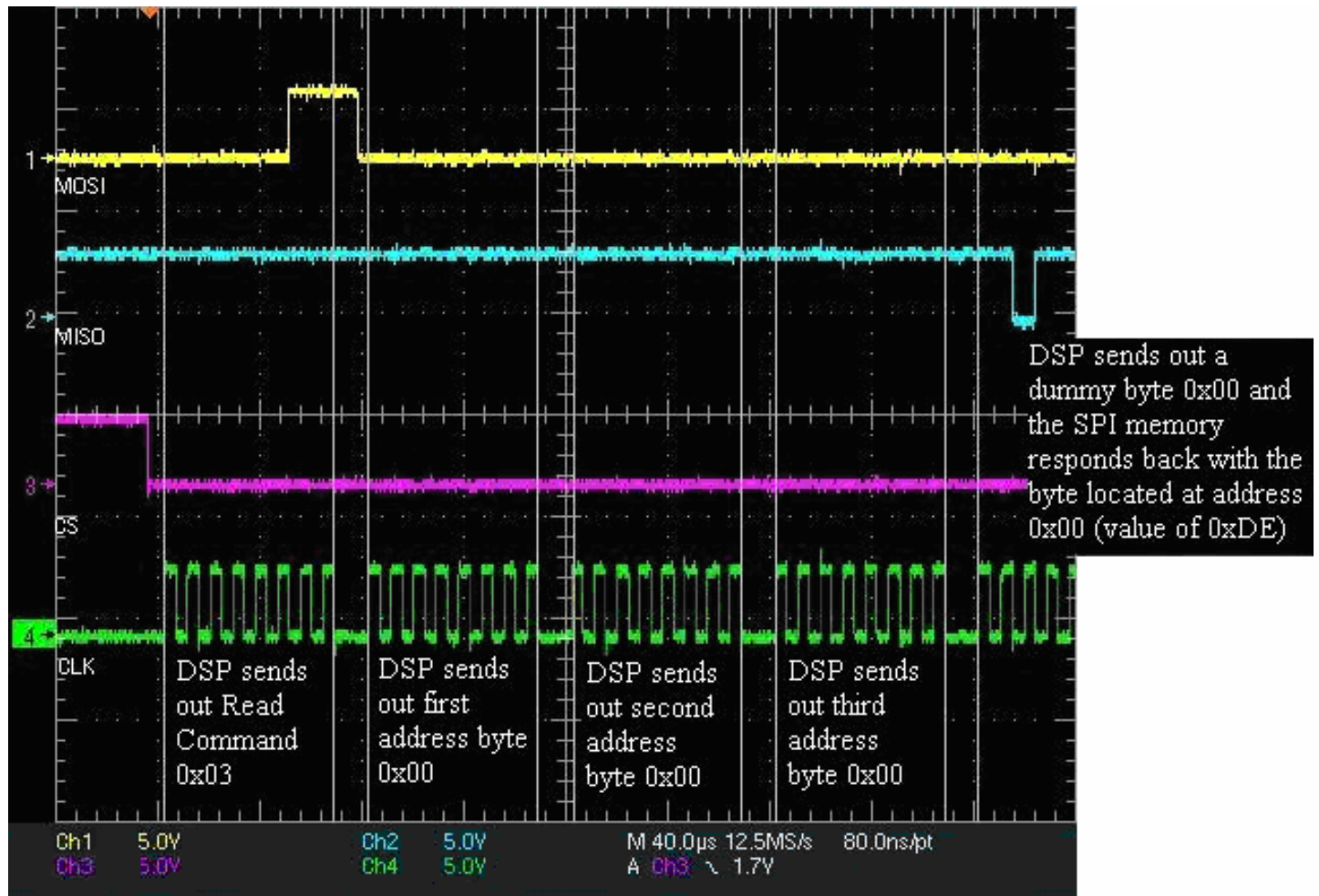


Figure 17. Memory detection routine for a 24-bit addressable device

Figure 18 shows the memory detection routine for a 16-bit addressable device (25LC640 from Microchip).

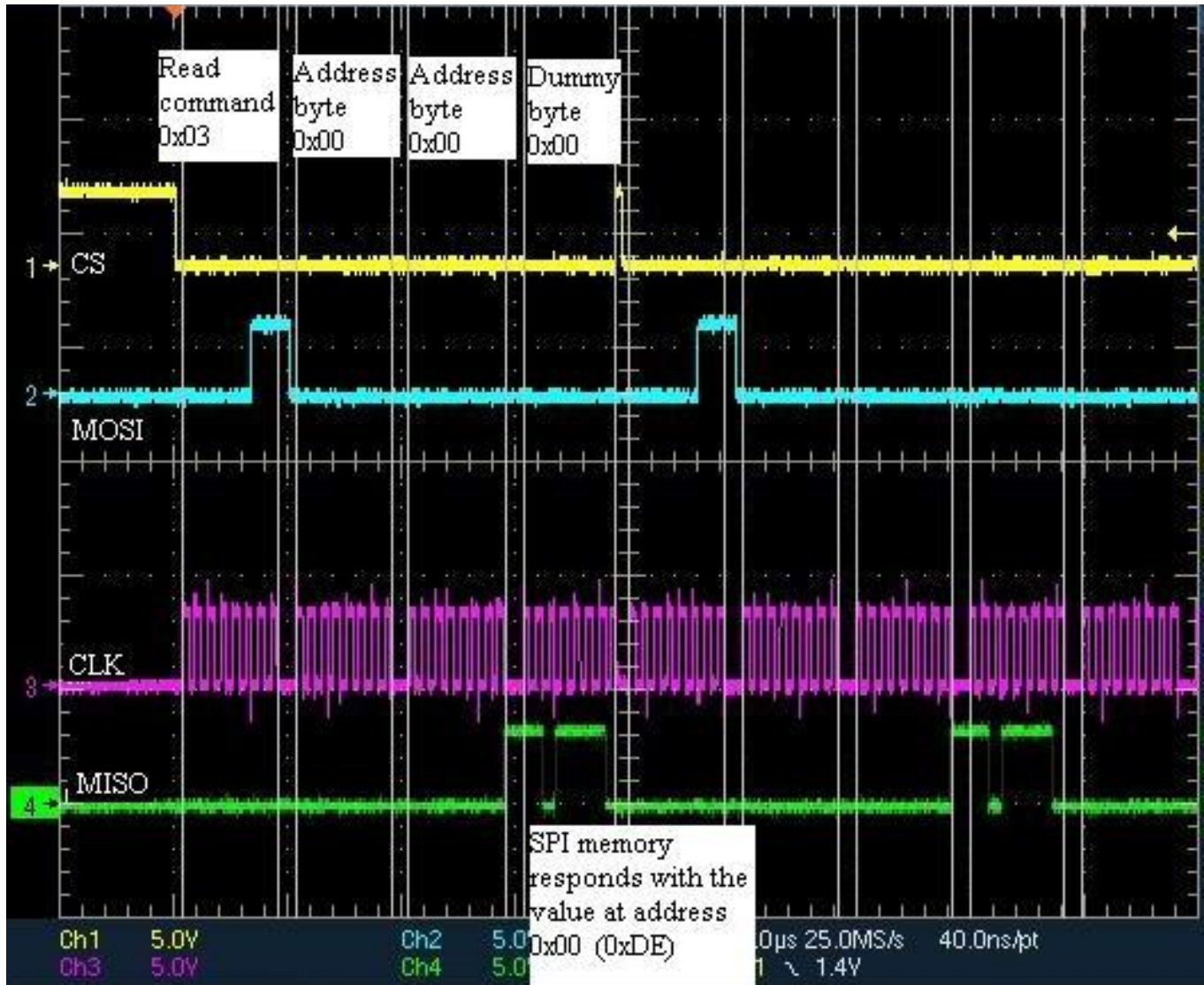


Figure 18. SPI memory detection routine for a 16-bit addressable device

### No-Boot or Direct Execution from External ASYNC Memory (BMODE = 00)

In this mode, when /RESET releases, the processor bypasses the boot ROM and starts fetching and executing instructions directly from address 0x20000000 in the off-chip asynchronous memory bank 0.

Only core A can execute directly from asynchronous memory. If core B also needs to execute an application in this mode, core A must place the required code and data into core B's memory using memory DMA. The procedure for getting an application working in this mode is similar to what is described in *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)*<sup>[4]</sup>.

When properly invoked, the loader utility (`elfloader.exe`) functions as a ROM splitter utility and is used to generate the `.LDR` file for this mode. Unlike the loader function of the utility, the splitter function does not format the application data when transforming a `.dxe` file into an `.ldr` file. It generates a file consisting of processor instructions (op-codes) that can be burned into an EPROM or flash memory device that connects to the target processor's system bus. The processor can directly fetch and execute these instructions rather than copy it to internal/external memories and execute it from there. The ROM splitter utility is invoked (Load : Splitter page of the Project Options dialog box) as shown in Figure 19. The Project Options should be set to generate a loader file.



Figure 19. Specifying ROM splitter options

The linker description file (`.LDF`) defines sections for placing the code and data into different memory segments inside the processor's memory map. Whether data and/or instruction segments are processed by the loader or the splitter utility depends upon the `.LDF` file's `TYPE()` command for the particular section. Segments declared with `TYPE(RAM)` are consumed by the loader function of the utility, and segments declared by `TYPE(ROM)` are consumed by the splitter function of the utility. The declaration of the PROGRAM memory segment as `TYPE(ROM)` is shown in Listing 2.

```
PROGRAM { TYPE(ROM) START(0x2000000) END(0x200FFFF) WIDTH(8) }
```

Listing 2. Declaring a ROM type memory segment

Data sections can be initialized automatically by using initialization qualifiers within the `.LDF` file and by enabling memory-initialization during linking. When this is done, the final `.Dxe` file will contain a data table similar to the boot stream produced by the loader utility for booting via the boot ROM. Listing 3 shows the use of initialization qualifiers for a data section. The `D_MEM` label is a section in L1 data memory.

```
data1 RUNTIME_INIT
{
    INPUT_SECTION_ALIGN(4)
    INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1) )
} > D_MEM
```

Listing 3. Data section with initialization qualifiers

The `bsz_init` and `.meminit` sections must be mapped to flash memory. The `bsz_init` section, which is very small, contains a pointer to the start of initialization data. The `.meminit` section is a special section that is populated after linking with the tables needed to perform the run-time initialization. In effect, it is similar to a boot stream: it contains blocks of data that are copied from flash memory to areas of RAM. Listing 4 shows the mapping of the memory initialization tables in flash memory.

```
bsz_init
{
    INPUT_SECTION_ALIGN(4)
    INPUT_SECTIONS( $OBJECTS(bsz_init) $LIBRARIES(bsz_init))
} >PROGRAM

.meminit
{
    ALIGN(4)
} >PROGRAM
```

*Listing 4. Memory initialization tables*

Run-time memory initialization is enabled by adding an extra flag during the link stage. There are two ways of doing this:

- If using the VisualDSP++ GUI, enable memory initialization by adding the `-meminit` switch in the Additional options field of the Link tab of the Project Options dialog box.
- When building the application from the command line, enable memory initialization by adding the `-mem` switch to the `ccblkfn` command.

An alternative to using the run-time initialization described above is to directly map the data sections to flash memory and declaring them as `TYPE(ROM)`.

A 16-bit memory device must be connected to the `/AMS0` strobe. To support reads from this memory region, the external bus interface unit (EBIU) uses the default external memory configuration that results from hardware reset. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).

The associated ZIP file contains an example project in which core A blinks all the LEDs on the EZ-KIT Lite board by running directly from the flash device on the board.

## Appendix 1 – Boot Modes on Older Silicon Revisions

### Silicon Revision 0.3

BMODE[1:0]	Description
00	Executes from external 16-bit memory connected to ASYNC Bank0 (bypass Boot ROM)
01	Boots from 8/16-bit flash/PROM
10	Not functional
11	Boots from a 16-bit addressable SPI memory in SPI master mode

### Silicon Revision 0.2

BMODE[1:0]	Description
00	Reserved. Executes from external 16-bit memory connected to ASYNC Bank0 (bypass Boot ROM)
01	Boots from 8/16-bit flash/PROM
10	Boots from an 8-bit addressable SPI memory in SPI master mode
11	Boots from a 16-bit addressable SPI memory in SPI master mode

*Note: Silicon Revision 0.4 of the ADSP-BF561 processor was never released.*

## Appendix 2 – Booting Using the Second-Stage Loader

This section applies only to older versions of VisualDSP++.

The boot ROM loads only one executable before it jumps to the start of core A instruction SRAM (0xFFA00000), though the loader file contains blocks for both .DXE files. Use a second-stage loader (SSL) when two .DXE files are to be loaded. The SSL must start at 0xFFA00000. The boot ROM loads and executes the SSL.

A default SSL is provided for each boot mode and can be customized by the user. Unlike the initialization blocks, it takes full control over the boot process and never returns to the boot ROM. It can use the .dxe byte count blocks to find specific .dxe files in external memory if a loader file includes code and data from multiple .dxe files. The default SSL uses the last 1024 bytes of L2 memory. This area must be reserved during booting but can be reallocated at runtime.

The SSL project must first be created and built. It should then be included in the Project Options as the Kernel File. The Use boot kernel option should be selected. The IDDE will only include p1.dxe on the elfloader command line, so you must enter ./debug/p0.dxe in the Additional options field, as shown in Figure 20.

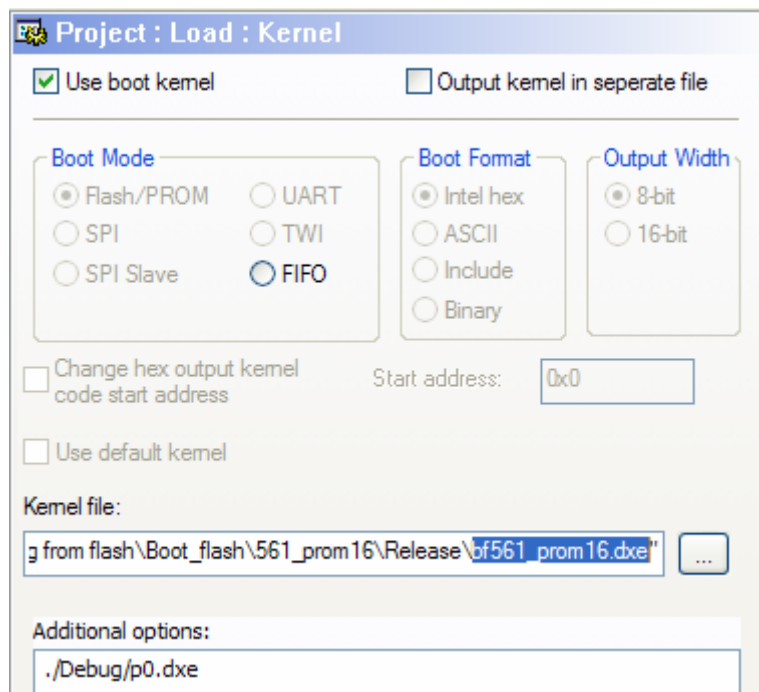


Figure 20. Specifying the second-stage loader

## Appendix 3 – Booting Checklist

If the processor does not boot, begin debugging with a simple application that toggles a programmable flag pin. Use an ICE (in-circuit emulator) to check whether the code was booted into the processor. Connect the ICE to the target and open an emulator session in VisualDSP++. Either clear the `Reset processor(s) before load` option on the `General` page of the `Preferences` dialog box (`Settings -> Preferences`), or ensure that the `Initial connection options` option in the `Device Properties` dialog box is set to `Halt` or `Do not disturb` (access the `Device Properties` dialog box via the `VisualDSP++ Configurator`). This ensures that the state of the processor is not disturbed when the emulator software starts up. Disconnect from the target and reset the board to boot the processor. Then reconnect to the target and view the `Disassembly` window. If you do not see the expected code, there is a boot failure. [Table 2](#) provides booting debug tips.

Booting Checklist	
<input type="checkbox"/>	BMODE pins: Ensure that the pins are configured properly (observe silicon revision restrictions!)
<input type="checkbox"/>	Unlock core B: Ensure that core B is unlocked by the core A application in a dual-core project
<input type="checkbox"/>	Configure SDC: Use init code to configure the SDC (if SDRAM is used)
<input type="checkbox"/>	Flash boot: Ensure that the 8/16 bit flash option is chosen appropriately
<input type="checkbox"/>	SPI slave boot: Pull down <code>HWAIT (PFx)</code> ; <code>PF0</code> should not be used for <code>HWAIT</code> ; the processor should be out of reset before the host starts sending data
<input type="checkbox"/>	SPI master boot: Pull up <code>/CS (PF2)</code> ; pull up <code>MISO</code> when booting from 24-bit SPI devices (applicable to 0.5 silicon only); Atmel D-type DataFlash devices are supported in “power-of-2” mode only
<input type="checkbox"/>	<code>/TRST</code> pin: Pull-down necessary to boot the processor; do not leave this pin floating (see EE-68 <sup>[6]</sup> )
<input type="checkbox"/>	<code>/BR</code> pin: Terminate appropriately; pull up if not used
<input type="checkbox"/>	Power to the processor: Ensure correct voltages and power-on reset timings (see data sheet)
<input type="checkbox"/>	Clock: Check <code>CLKIN</code> , <code>CLKOUT</code> ; PLL ratios should ensure that <code>CCLK</code> and <code>SCLK</code> are within their specified limits

Table 2. Booting debug tips

## References

- [1] *ADSP-BF561 Blackfin Processor Hardware Reference*. Rev 1.0, July 2005. Analog Devices, Inc.
- [2] *ADSP-BF533 Blackfin Booting Process Application Note (EE-240)*. Rev 3, January 11, 2005. Analog Devices, Inc.
- [3] *Visual DSP++ 4.5 Loader and Utilities Manual* Rev 1.0, April 2006 Analog Devices, Inc.
- [4] *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)*. Rev 1, May 22, 2004. Analog Devices, Inc.
- [5] *Managing Multiple DXEs on ADSP-BF561 Blackfin Processors (EE-272)*. Rev 1, July 11, 2005. Analog Devices, Inc.
- [6] *Analog Devices JTAG Emulation Technical Reference (EE-68)*. Rev 9, October 18, 2004. Analog Devices, Inc.

## Document History

Revision	Description
<i>Rev 1 – May 15, 2007 by Jayanti Addepalli</i>	Initial release.