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

# Boot Kernel Customization and Firmware Upgradeability on SHARC® Processors

*Contributed by Mitesh Moonat* *Rev 1 – October 5, 2009*

## Introduction

SHARC® processors are hardwired to read 256 instructions from either a non-volatile memory (master boot) or a host (slave boot) after reset. This set of 256 instructions is called the *boot kernel*. The boot kernel reads the rest of the boot stream and initializes the internal and external memory with the main application. Default boot kernels for all boot modes are provided with the VisualDSP++® development tools. Most of the time, the default boot kernel can be used as is to boot an application. Sometimes, a little modification of the boot kernel may be needed to add PLL (Phase Locked Loop) or external memory controller (AMI, SDRAM/DDR2DRAM) initialization code. Moreover, some scenarios may require boot kennel modifications to achieve results different from the normal booting procedure.

This EE-Note discusses some of these scenarios and explains how the default boot kernels can be modified to satisfy the application requirements in such cases. The document further discusses how the firmware on SHARC-based boards can be upgraded dynamically using in-circuit flash programming. To illustrate the two concepts (boot kernel modification and firmware upgradeability), a typical application is provided as an example. This application first upgrades the firmware via RS-232 communication with a graphical user interface (GUI) running on the PC and then uses a customized boot kernel to load the upgraded application. To better illustrate the approaches discussed, example code for the ADSP-21262, ADSP-21364, ADSP-21369, ADSP-21375, and ADSP-21469 EZ-KIT Lite® evaluation systems are provided with this application note in the associated `.ZIP` file.

## SHARC Boot Modes

The supported boot modes for the ADSP-2126x, ADSP-2136x, ADSP-2137x, and ADSP-2146x SHARC processors are:

1. SPI slave boot
2. SPI master boot
3. Parallel port (ADSP-2126x and ADSP-21366/5/4/3/2 processors only)
4. External port boot (ADSP-21369/8/7, ADSP-2137x, and ADSP-2146x processors only)
5. Link port boot (ADSP-2146x processors only)

The boot kernels for the above-mentioned boot modes share similar structure. The majority of differences in the boot kernels are due to the different peripherals used to read the boot stream. For details on the structure of SHARC boot kernels, refer to *Tips and Tricks on SHARC EPROM and Host Boot Loader (EE-56)*[1].The boot kernel modifications are mostly required for boot modes in which the SHARC processor is master (boot modes 2 and 3). This application note focuses mainly on boot modes 2 and 3 where the DSP boots from a parallel or serial EEPROM/flash. Figure 1 shows how the `.ldr` file `Blink.ldr` situated on the flash, which consists of following three sections:

1. **Boot Kernel:** The first 1536 bytes of the `.ldr` file corresponding to the 256 instructions of the boot kernel.
2. **Application excluding IVT:** The content following the boot kernel that contains the information regarding initialization of the internal (16-bit, 32-bit, 48-bit, 64-bit) and external memory content corresponding to the application being booted. This, however, does not include the interrupt vector table.
3. **Final Init Block:** This section contains the 256 instructions corresponding to the interrupt vector table of the application being booted. These instructions are initialized by a special self-modifying subroutine in the boot kernel.

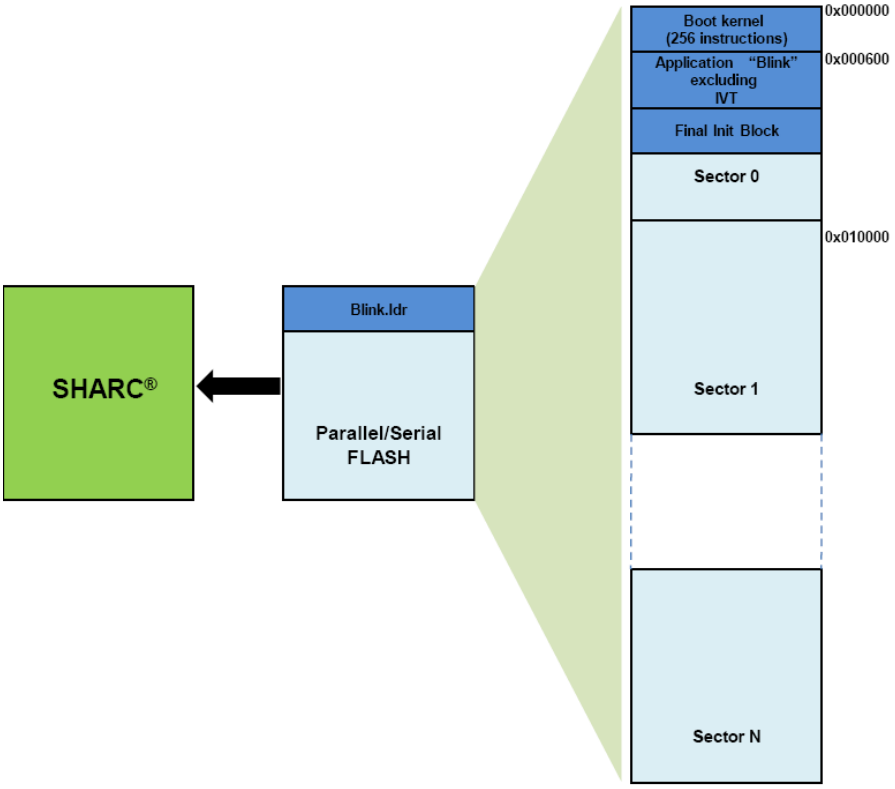


*Figure 1. SHARC booting via parallel/serial flash*

Note that Figure 1 assumes the length of all the sectors of the flash is same and equal to 0x10000 bytes. It could be different for different a flash devices.

## Boot Kernel Customization

This section discusses two cases where the modification required in the boot kernel is different from adding PLL and external memory interface initialization code.

### CASE 1: Selective Booting of One Application from Multiple Applications

Figure 2 shows a simple case where two applications "Blink1" and "Blink2" reside in the flash: one in sector 0, and the other in sector 1. The default boot kernel will load the application from sector 0 (Blink1). The requirement is to modify the boot kernel in a way that based on a condition (for example, the assertion of a flag input); it should load one of the two applications selectively. The customized boot kernel should be loaded after reset. Thus, it should be a part of the `.ldr` file corresponding to the application in sector 0 (Blink1). The `.ldr` file corresponding to the application in sector 1 (Blink2) can have the default boot kernel.

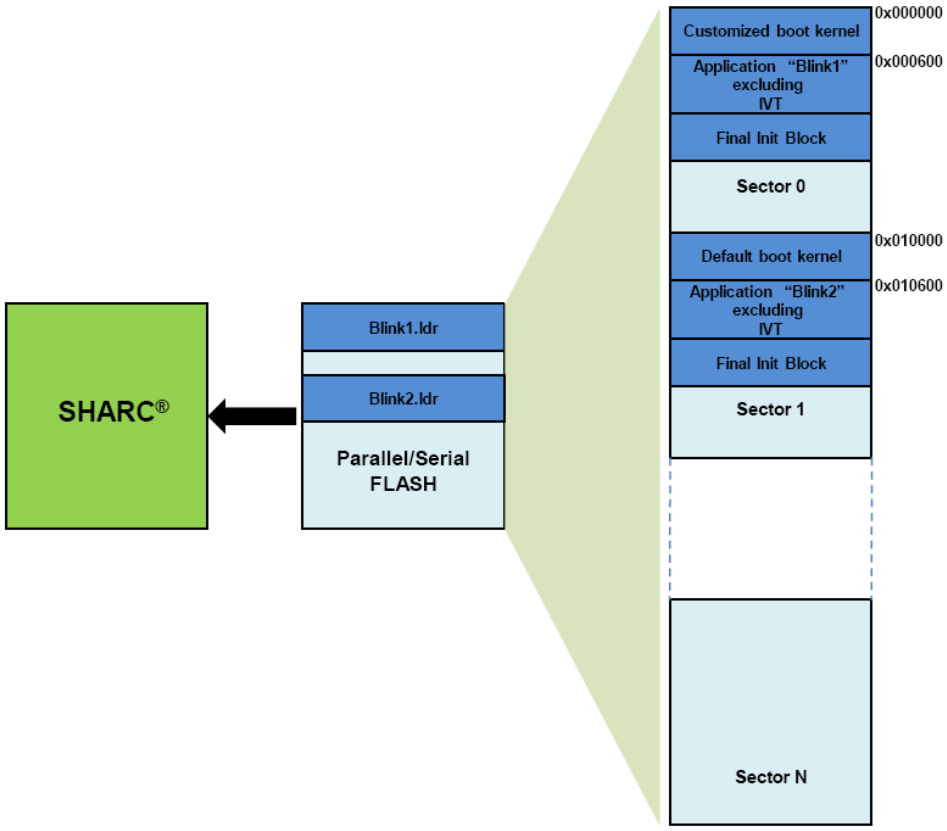


*Figure 2. Booting one of multiple applications from the flash*

Note that Figure 2 assumes the length of all the sectors of the flash is same and equal to 0x10000 bytes. It may be different for a different flash device.

This case can further be extended to have more than two applications. Consider the following two factors while making modifications in the boot kernel:

- The length of the customized boot kernel should not exceed 256 instructions.

- The interrupt vector location corresponding to the peripheral being used must have the `RTI` instruction intact. After the boot kernel is loaded, the execution branches to the interrupt vector location corresponding to the peripheral being used (SPI/external port/parallel port). This location should have an `RTI` instruction to return to the reset vector location and start executing the kernel. Moreover, some boot kernels also use this `RTI` instruction for further reads (interrupt based) from the flash.

*Approach 1: Modifying the Read Address in the USER_INIT Section*

The first 256 instructions are loaded, starting from the physical address 0x000000 of the parallel/serial flash. Thus, the external address settings for the parallel port, external port, and SPI master boot modes at reset are:

- `EIPP` = 0x000000
- `EIEP` = 0x4000000
- Read address sent to the SPI flash with the read command = 0x000000

The `EIPP` register increments by four after one 32-bit word (four bytes) is read and the `EIEP0` register increments by one after one 32-bit word (four bytes) are read. The SPI flash device increments its read address internally by one after one byte is read. Thus, after the 256 instructions are loaded:

- `EIPP` = 0x000600
- `EIEP` register = 0x4000180
- Address of the next byte to be received from SPI flash = 0x000600

In the default boot kernels, these addresses are not modified. To change the location where the boot kernel fetches the application, modify the `EIPP` and `EIEP` registers for parallel/external port boot or send a new read command to the SPI flash for SPI master boot. This modification has to be added in the `USER_INIT` section of the boot kernel before it starts reading the rest of the boot stream from the flash.

Listing 1 shows the code required to reinitialize the `EIPP` register for the parallel port boot to load "Blink2" when `FLAG0` = 1. The new `EIPP` register value = *Sector 1 start address (0x10000) + 8-bit offset to skip the default boot kernel (0x600) = 0x10600.*

```
If not FLAG0_IN jump (pc,3);
ustat1=0x10600;
dm (EIPP)=ustat1;
```

*Listing 1. Boot kernel modification for parallel port booting*

Listing 2 shows the code required to reinitialize the `EIPP` register for the parallel port boot to load "Blink2"when `FLAG0` = 1. The new `EIEP0` register value= *32-bit logical address for sector 1(0x404000) + 32-bit offset to skip the default boot kernel (0x180) =0x404180.*

```
If not FLAG0_IN jump (pc,3);
ustat1=0x404180;
dm (EIEP0)=ustat1;
```

*Listing 2. Boot kernel modification for external port booting*

Listing 3 shows the code required to change the read address for SPI master boot. Unlike parallel port and external port boot, it cannot be performed by changing the value of a register. Instead, it requires sending

a read command to the SPI flash with the new address. Similar to the parallel port boot, the new address for this case would be = *Sector 1 start address (0x10000) + 8-bit offset for the boot kernel (0x600) = 0x10600.*

```
    if NOT FLAG0_IN jump skip;

    r0=0x80;            //Flush the DMA FIFO
    dm(SPIDMAC)=r0;

    r0=TXFLSH|RXFLSH;   //FLUSH the SPI buffers
    dm(SPICTL)=r0;

    r0=0x64;            //Change SPI Baud (if required)
    dm(SPIBAUD)=r0;

    //Set up a receive DMA to a dummy location to send a read command to the FLASH
    r0=0xb8000;
    dm(IISPI)=r0;
    r0=1;
    dm(IMSPI)=r0;
    r0=1;
    dm(CSPI)=r0;

    r0=0xFF01;          //Keep the Chip Select high
    dm(SPIFLG)=r0;

    r0=SPIEN|SPIMS|WL32|SENDZ|CLKPL|CPHASE|TIMOD2;
    dm(SPICTL)=r0;

    r0=0x10000;         //Sector 1 Offset
    r1=0x03000600;      //Read command with 1536 bytes (boot kernel) offset
    r0=r0 OR r1;        //Combining the read command with the sector 1 start address
    i0=r0;
    bitrev(i0,0);       //Bit reversing the word as SPI FLASH expects MSB first
    r0=i0;
    dm(TXSPI)=r0;       //Putting the command and address in the transmit buffer

    r0=0xFE01;          //Now asserting the chip select (high to low transition)
    dm(SPIFLG)=r0;

    r0=0x00000007;      //Initiating a read DMA to send the word in TXSPI over MOSI
    dm(SPIDMAC)=r0;

    ustat1=dm(SPIDMAC);//Waiting for the DMA to finish
    bit tst ustat1 SPIDMAS;  // Check SPI DMA Status bit
    IF TF jump (pc,-2);// SPIDMAS = 1 when DMA in progress

    Skip:
    //Default kernel code……
```

*Listing 3. Boot kernel modification for SPI master boot*

### *Testing the EZ-KIT Lite Example Code*

Example code is provided for parallel/external boot and SPI master boot for ADSP-21262, ADSP-21364, ADSP-21369, ADSP-21375, and ADSP-21469 EZ-KIT Lite evaluation systems. The following steps

show how to test the modified boot kernels for CASE 1 on the ADSP-21369 EZ-KIT Lite board. Code for other EZ-KIT Lite systems can be tested similarly.

1. Build the modified kernel `369_prom` project to generate the `369_prom.dxe` file.
2. Build the project "Blink1" to generate `Blink1.ldr` for external port boot in ASCII format with loader options, as shown in Figure 3. Ensure that it uses the modified kernel `369_prom.dxe`.
3. Build the project "Blink2" to generate `Blink2.ldr` for external port boot in ASCII format with loader options, as shown in Figure 4. This project may use default boot kernel as its boot kernel will not be used.

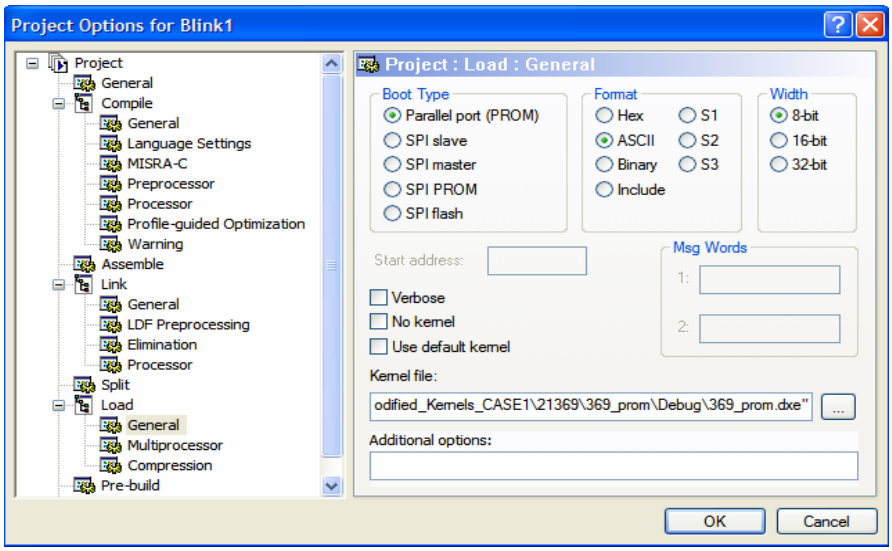


*Figure 3. Loader options for "Blink1"*

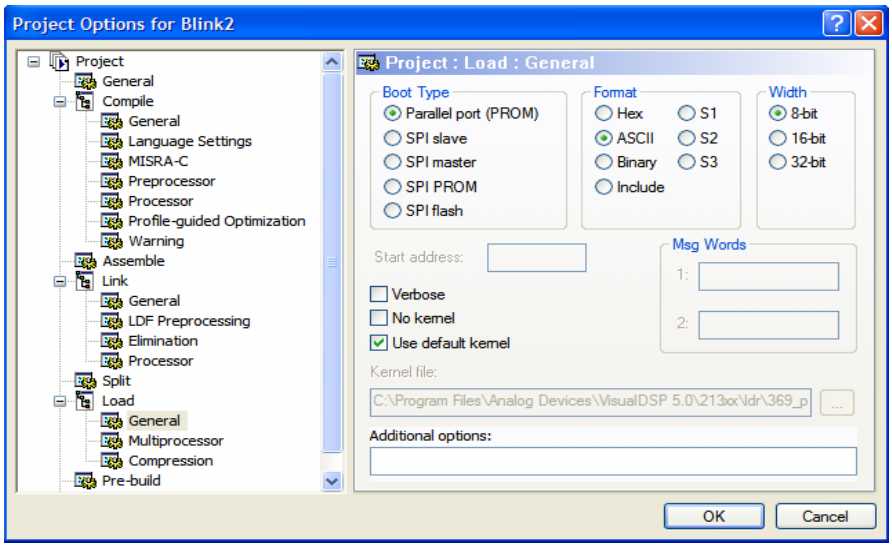


*Figure 4. Loader options for "Blink2"*

4. Load `Blink1.ldr` into sector 0 and load `Blink2.ldr` into sector 1 of the parallel flash using the VisualDSP++ Flash Programmer utility. Figure 5 and Figure 6 show the Flash Programmer options for loading `Blink1.ldr` and `Blink2.ldr`, respectively.

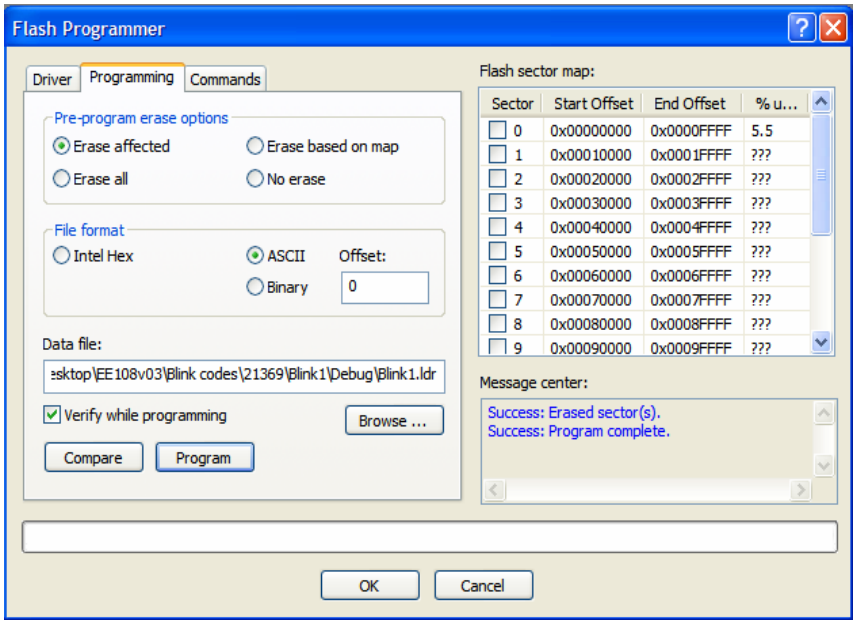


*Figure 5. Loading "Blink1" in sector 0*

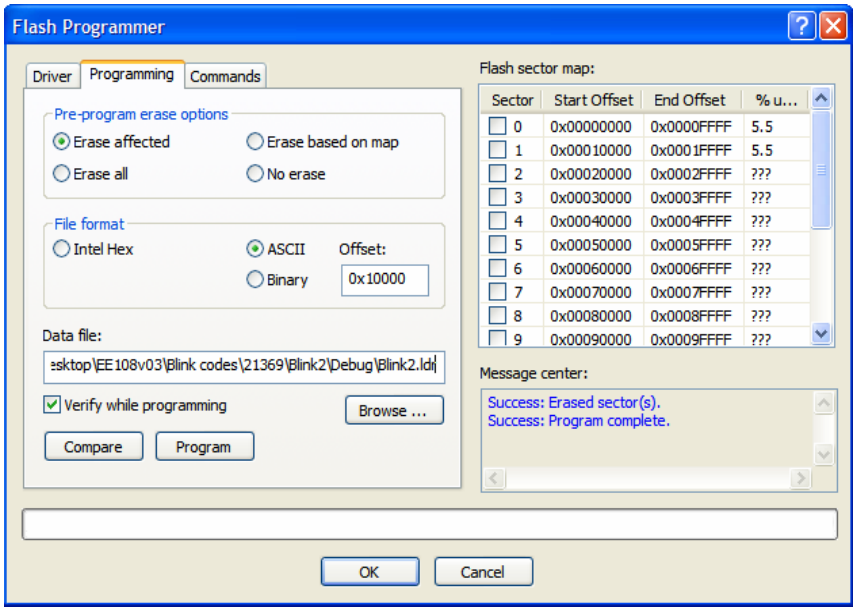


*Figure 6. Loading "Blink2" in sector 1*

5. Close the VisualDSP++ session, remove the emulator, and press the `RESET` switch (or power-cycle the EZ-KIT Lite board). Ensure that the `BOOT_CFGx` pins are set to use the external port boot mode. After reset, the customized boot kernel loads and checks the status of the `FLAG0` pin (connected to `SW8`). If the switch was pressed (`FLAG0` = 1), the kernel loads application "Blink2"; otherwise, if `FLAG0` = 0, it loads the application "Blink1".

*Approach 2: Using Multiprocessor Booting Feature*

Some SHARC parts (especially older models) such as ADSP-2106x, ADSP-2116x, and ADSP-21368 processors can support multiple SHARC devices sharing the same external bus. For these processors, the VisualDSP++ loader supports a booting scheme whereby a cluster of up to six processors can be booted from a single EPROM/flash device. The generated hexadecimal file contains the boot loader and the boot images for the processors. Furthermore, an included table contains the EPROM/flash offset address of the executables. After reset, the identical boot loader will be loaded into all the SHARC processors of the cluster. The original boot loader first determines the ID of the processor on which it is running. From the offset table, it obtains the address where the corresponding data is stored in the EPROM/flash. Finally, the boot loader boots the application as in a single-processor scheme.

This approach takes advantage of this feature and use the `Multiprocessor input file(s)` box (on the `Load : Multiprocessor` tab) to group multiple executables in an EPROM/flash. Now something else (rather than the processor ID) determines the application to be loaded. So the original boot loader has to be modified slightly.

First, create a new project (let's call it `multi_loader`) to rebuild the boot loader. Then, copy the original source file (`060_prom.asm` or `065L_prom.asm`) and the corresponding Linker Description File (`060_ldr.ldf` or `065L_ldr.ldf`) from `<install_path>\21k\ldr` into your project directory. Use the "060" files for all ADSP-2106x processors except ADSP-21065L processors. The original `060_ldr.ldf` file selects the ADSP-21062 processor. Feel free to change the type.

Search in the original source file for the instructions that determines the processor's ID.

```
R0=DM(SYSTAT);
R0=FEXT R0 BY 8:3;
```

*Listing 4. Instruction identifying processor ID*

As shown in Listing 4, the two code lines determine the ID stored in `R0`. The content of `R0` determines the executable to be booted. Now, these two instructions will be replaced by any others. For example, `R0` might be controlled by a jumper connected to the flag pins. Two issues must be taken into account:

- `R0` must have a value between 1 and 6
- The maximal length of the boot loader must not exceed 256 instructions

Listing 5 is an example of the modifications required for the ADSP-21061 EZ-KIT Lite board. Normally, this loader will boot program #1, but if the `FLAG1` button is pushed during reset, program #2 will be loaded instead.

```
R0=1;
IF NOT FLAG1_IN R0=R0+1;
```

*Listing 5. Boot kernel modification for ADSP-21061 EZ-KIT Lite*

Since you may modify `R0` any way you like, many scenarios become possible. Instead of simply checking the flag pins, you can set up the SPORT or a Link Port to receive the number of the application to be booted. Additionally, this technique can also be used to load the same executable into more (or all) SHARC processors in a cluster. Of course, you can do that using the original boot loader, but then you have to store the same application multiple times in the EPROM/flash. Just use the instruction "`R0=1;`" or

"R0=0;" when the Multiprocessor input file(s) box is not used. Furthermore, you can load program #1 into processors ID2, ID4, and ID6 and load program #2 into processors ID1, ID3, and ID5. You might even combine the ID with a jumper value.

### *ADSP-21161N Example Code*

The example code provided for this approach can be tested on an ADSP-21161N EZ-KIT Lite evaluation system. "Blink 1" toggles FLAGS4-6, and "Blink2" toggles FLAGS7-9 on the board. The PROM boot kernel is modified to boot the application, based on the FLAG2 switch on the board. During reset, if the FLAG2 switch is pressed, blink example 1 is booted; otherwise, example 2 is booted.

Perform the following procedure to test this approach:

1. **Test the applications.** Build the modified PROM boot kernel project file, 161_prom, included with this EE-Note. Individually test both blink LED examples on the ADSP-21161N EZ-KIT Lite board.
2. **Create the loader file.** Create the combined loader file for both applications with the loader options, as shown in Figure 7. Program the loader file in to the flash using the VisualDSP++ Flash Programmer utility.
3. **Boot the applications.** After programming the flash, close the VisualDSP++ debug session and press the RESET button on the EZ-KIT Lite board. By default, blink example 2 is booted; when the FLAG2 button is pressed during reset, blink example 1 is booted. Note that the boot configuration on the ADSP-21161N EZ-KIT Lite board must be configured for EPROM boot.

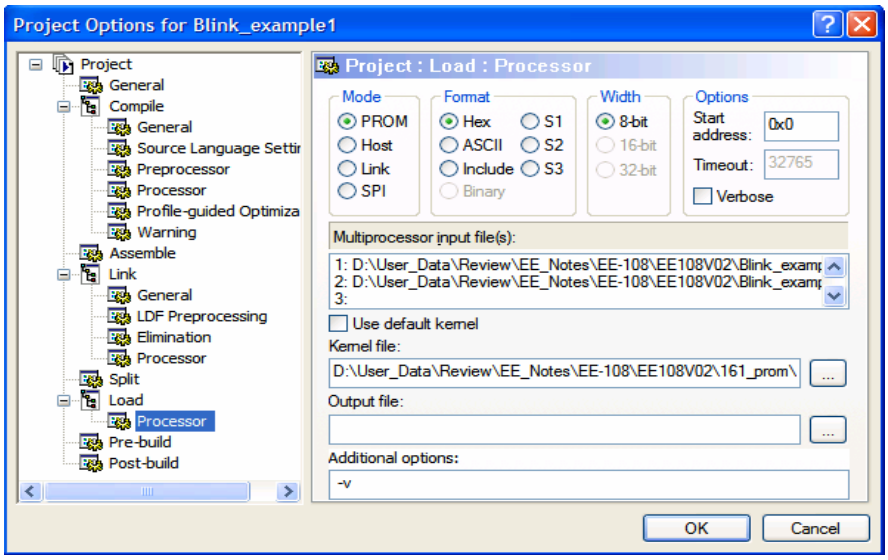


*Figure 7 Loader options for using the multiprocessor boot feature*

## CASE 2: Using a Boot Kernel for a Second-Stage Loader

Some applications may require more initialization other than initializing PLL and external memory interface. It may be initialization of the signal routing unit (SRU) or any other user-defined routines. Most have to be done only at the start of the application. Moreover, in some cases, the processor might be required to communicate with a host enquiring whether a firmware upgrade is available. If so, it has to

receive the new firmware, burn the upgraded application into the flash, and then boot the processor with the upgraded application.

The cases above require that code is executed by the processor before it loads the actual application. It may not always be possible to include this code in the `USER_INIT` section of the boot kernel for the following two reasons:

- The modified boot kernel's size cannot exceed 256 instructions.
- The default boot kernel sits in the interrupt vector table. Thus, the code cannot use the interrupts.

An alternative way could be to boot a small application first with the help of the default boot kernel. This small application should perform the user-defined initialization tasks and then jump to a section of the code that overwrites this small application with the actual application. This section of the code can be called a *second-stage* loader. It can be obtained by few modifications to the default boot kernel. Unlike the default boot kernel, it can be placed in any memory location.

This following section discuss the important factors that need to be addressed when modifying the boot kernel this way.

### *Interrupt Vector Table*

The interrupt vector table of the application using the modified boot kernel should still have an `RTI` instruction at the corresponding interrupt vector location (parallel port/external port/SPI). This can be done for easily for assembly code. For C code, you can modify the corresponding <processor family>`_hdr.asm` file (e.g., `26x_hdr.asm`, `36x_hdr.asm`, `46x_hdr.asm`). Listing 6 shows the `RTI` instruction added at the external port (`EP0`) interrupt vector location.

```
Default "36x.hdr" code……
___lib_P9I:          rti;nop;nop;nop;       // Peripheral interrupt 9
Default "36x.hdr" code…..
```

*Listing 6 .Modifying "36x_hdr.asm"*

### *"final_init" Section*

The `final_init` section uses labels such as *"reset"* and *"final_init_loop"*. Since the modified boot kernel will not be placed in the IVT, these labels will not hold the same address as in the case of default boot kernel. Thus, to ensure that these addresses still remain the same, modifications are required in the instructions that use these labels. Listing 7 shows the definition of these labels at the start of the default boot kernel (`369_prom.asm`) and the instructions using them in the `final_init` section. Listing 8 shows the start of the modified `369_prom.asm` file and the modified instructions in the `final_init` section. The `_lib_RSTI` label is defined in `36x_hdr.asm` and is equal to 0x90005. The `reset` label is defined as "0x90003" which is the same as in the default boot kernel.

```
.SECTION/PM     seg_ldr;
    nop;nop;nop;
reset:
    nop;
final_init_loop:
    nop;

    call USER_INIT;
```

```
    LIRPTL = 0;
    IMASK = 0;
    IRPTL = 0;
    BIT SET MODE1 IRPTEN;
    ……
    ……
    DO final_init_loop UNTIL EQ;
    PCSTK=reset;

    ustat1 = dm(DMAC0);
    bit set ustat1 DEN;
    dm(DMAC0) = ustat1;

    JUMP reset (DB);
    IDLE;
    LIRPTL=0;
```

*Listing 7. Default "369_prom" boot kernel*

```
.global _load_application_parallel;
.extern _app_offset_parallel;
#define reset 0x90003
.extern ___lib_RSTI;

.SECTION/PM     seg_ldr;

_load_application_parallel:

    call USER_INIT;
    LIRPTL = 0;
    IMASK = 0;
    IRPTL = 0;
    BIT SET MODE1 IRPTEN;
    ……
    ……
    DO ___lib_RSTI UNTIL EQ;
    PCSTK=reset;

    ustat1 = dm(DMAC0);
    bit set ustat1 DEN;
    dm(DMAC0) = ustat1;

    JUMP reset (DB);
    IDLE;
    LIRPTL=0;
```

*Listing 8. Modified "369_prom" boot kernel*

### *Peripheral Initialization*

The default boot kernels need not initialize the corresponding peripheral as it is already initialized at reset. For example, in case of external port boot for the ADSP-21369 processor, the `AMICTLx` and `EPCTL` registers are initialized to 0xF4 and 0x5C1, respectively, after reset. However, for the modified boot kernel, the peripheral will have to be explicitly initialized. It can be done in the `USER_INIT` section. Listing 9 shows the initialization code in the modified `369_prom.asm` file.

```
USER_INIT:
        ustat1 = 0xF4;
        dm(EPCTL)=ustat1;
        ustat1=0x5C1;
        dm(AMICTL1)=ustat1;
        r0=dm(_app_offset_parallel);
        r0=lshift r0 by -2;
        r1=0x4000180;
        r0=r0 or r1;
        dm(EIEP0)=r0;
```

*Listing 9. Modified "369_prom" boot kernel*

### *Memory Usage Restrictions*

Ensure that the memory range (256 instructions) being used by the modified boot kernel subroutine is reserved and not used by the application being loaded. This is because the modified boot kernel no longer resides in the IVT. Thus, it may be overwritten at any time (even before the `final_init` section starts executing).

Unlike the default booting, the registers are not brought back to their reset state when loading an application using the modified boot kernel. If required, the application being loaded later might have to explicitly reset some registers which were modified by the previous application.

### *Testing the EZ-KIT Lite Example Code*

Example code is provided for ADSP-21262, ADSP-21364, ADSP-21369, ADSP-21375, and ADSP-21469 EZ-KIT Lite systems to show how an application can jump to the modified boot kernel code to overwrite itself with a new application. Listing 10 shows the `369_SSL` application using the modified boot kernels. To test this, first burn the `.ldr` file corresponding to any blink code at a particular flash offset and initialize `app_offset_parallel` or `app_offset_serial` accordingly. Then, uncomment `#define PARALLEL` to load the application from parallel flash or `#define SERIAL` to load the application from serial flash.

```
#define PARALLEL
//#define SERIAL

int app_offset_parallel = 0x10000;
int app_offset_serial=0;//x10000;
extern load_application_parallel();
extern load_application_serial();

int main()
{
    #ifdef PARALLEL
        asm("jump _load_application_parallel;");
    #endif
    #ifdef SERIAL
        asm("jump _load_application_serial;");
    #endif
    return 0;
}
```

*Listing 10. "369_SSL" using the modified boot kernels*

## Firmware Upgrade

For modern systems, upgrading the existing firmware with new firmware corrects and enhances the supported features and functionalities. Flash devices placed in a socket in a system can be removed easily, upgraded, and replaced. But this approach cannot be used for systems with soldered flash devices. Also, it has to be done manually and thus may not be used for dynamic upgrade. In such cases, *in-circuit flash programming* is a technique by which flash devices can be programmed with means of software running on the processor itself. For details on *in-circuit flash programming* for SHARC processors, refer to [2], [3], [4].

### Testing the EZ-KIT Lite Example Code

To illustrate the concept of a firmware upgrade, example code is provided for ADSP-21262, ADSP-21364, ADSP-21369, ADSP-21375, and ADSP-21469 EZ-KIT Lite evaluation systems. The example code communicates with a graphical user interface running on a PC via RS-232 to receive the `.ldr` file of the firmware to be loaded. Figure 8 shows a snapshot of the GUI. The SPORT is used to emulate UART protocol for processors that do not have an on-chip UART (ADSP-21364 and ADSP-21262 processors).

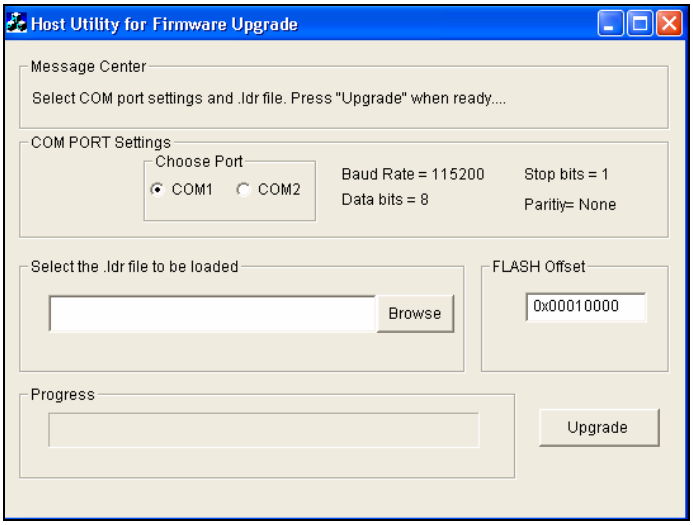


*Figure 8. GUI for firmware upgrade*

Before opening the `Firmware_Upgrade.exe` executable, ensure that the `.dll` files (`MFC42D.DLL`, `MFCO42D.DLL`, and `MSVCRTD.DLL`) provided in the associated `.ZIP` file are copied to the path <install_path>`\WINDOWS\system32`. To test the example code on any of the EZ-KIT Lite evaluation systems, perform the following steps:

1. Connect the RS-232 port of the EZ-KIT Lite board to COM1/COM2 port of the PC. For ADSP-21262 or ADSP-21364 EZ-KIT Lite systems, use an external RS-232 transceiver chip (the one provided with ADSP-21369, ADSP-21375, and ADSP-21469 EZ-KIT Lite systems can also be used). Connect `DAI_PB02` to UART TX (from the processor), and `DAI_PB04` to UART RX (to the processor).
2. Load the example code `xxx_SSL_GUI` (where xxx = 21262/21364/21369/21375/21469) and run. This code blinks a particular LED (mentioned in the respective code) on the EZ-KIT Lite board five times and then waits until a particular push button (mentioned in the respective code) on the EZ-KIT Lite board is pressed.

3. Open the GUI, select the COM port, select the `.ldr` file to be sent, and click `Upgrade`.
4. Press the corresponding push button (based on the flag pin used in the code) on the EZ-KIT Lite board.
5. The firmware upgrade procedure starts and performs the following steps:
    1. The processor signals the PC that it is ready to communicate.
    2. The PC sends four bytes corresponding to the 32-bit integer indicating the length of the `.ldr` file. The processor sends acknowledgement byte "0x03" for each byte received.
    3. The PC sends four bytes corresponding to the 32-bit flash offset where the `.ldr` file needs to be burnt. The processor sends acknowledgement byte "0x04" for each byte received.
    4. The PC sends the first byte of the `.ldr` file.
    5. The processor receives the byte sent by the PC, writes it into the flash, and sends the acknowledgement byte "0x02" to the PC.
    6. Steps (d) and (e) are repeated until all bytes of the `.ldr` file are sent and programmed into the flash. The flash now contains the upgraded boot image.
6. The processor loads the upgraded firmware with the help of modified boot kernel with the approach mentioned in CASE 2.

In the protocol mentioned here, the processor first receives one byte from the PC, programs it into the flash, then receives the next byte, and so on. To speed up the communication further, use DMA transfers to bring in the next block of data in parallel, while the core is programming the current block of data into the flash. A CRC error check can also be added to ensure data integrity while receiving data from the PC.

## References


[1] *Tips and Tricks on SHARC EPROM and Host Boot Loader (EE-56).* Rev 3, March, 2007, Analog Devices Inc.

[2] *In-Circuit Flash Programming on SHARC Processors (EE-223).* Rev 2, February, 2007, Analog Devices Inc.

[3] *In-Circuit Programming of an SPI Flash with SHARC Processors (EE-231).* Rev 2, August, 2007, Analog Devices Inc.

[4] *In-Circuit Flash Programming on ADSP-2106x SHARC Processors (EE-280).* Rev 2, March 2007, Analog Devices Inc.


## Document History

| Revision | Description |
|---|---|
| *Rev 1 – October 5, 2009*<br>*by Mitesh Moonat* | Initial release.<br><br>Note: contents and code examples from EE-108, "*Managing Multiple Applications in a Single EPROM for SHARC Processors*", have been merged into this EE-Note. Therefore, EE-108 has been made obsolete. |