



Managing Multiple DXEs on ADSP-BF561 Blackfin® Processors

Contributed by Manik Aryapadi

Rev 1 – July 11, 2005

Introduction

This application note describes how to manage the booting of multiple executables from external memory at run-time, thus providing the flexibility to dynamically switch among them.

This project was implemented on the ADSP-BF561 EZ-KIT Lite® evaluation system (ADDS-BF561-EZLITE, Rev 1.1) and applies to the ADSP-BF561 Blackfin® processors with silicon revision 0.2 and beyond.

The software code accompanying this project was tested using VisualDSP++® 4.0 tools.

Synopsis

This application note addresses the following topics:

- The ADSP-BF561 Blackfin processor booting process
- Multi-application (multi-DXE) management for the ADSP-BF561 Blackfin processors
- Reconfiguring the C/C++ run-time header
- Second-stage loader (SSL) memory management
- Booting multiple DXE files from 16-bit flash memory and swapping them at run time without having to reset

ADSP-BF561 Booting Mechanism

The ADSP-BF561 Blackfin processor consists of a multiprocessor configuration featuring two Blackfin cores. Since the core architectures of the ADSP-BF561 processor and the ADSP-BF533 processor are alike, there are no significant differences in the booting mechanism between the two devices.

The booting process of the ADSP-BF533 processor is documented in *ADSP-BF533 Blackfin Booting Process (EE-240)* ^[1]; therefore, only those details unique to the ADSP-BF561 processor are detailed in this application note.

Boot modes supported on various silicon revisions and the appropriate jumper and DIP switch settings for each are elucidated in the Appendix.

The ADSP-BF561 Blackfin processor has two cores: core A and core B. After reset, the on-chip Boot ROM, which is located at 0xEF00 0000, is executed.

The first step in creating the boot code involves compiling and linking the application code into an executable file. The elfloader utility then converts the executable into a boot stream file (.LDR), which is then burned into flash or another external memory device (e.g., PROM, EEPROM, etc.).

After reset, the Boot ROM reads the boot stream file from an external memory device, parsing the headers and moving blocks of data to specified memory locations. After the blocks are loaded,

the Boot ROM jumps to the start of core A's L1 instruction SRAM (0xFFA0 0000) and executes

the code. This procedure is summarized in Figure 1.

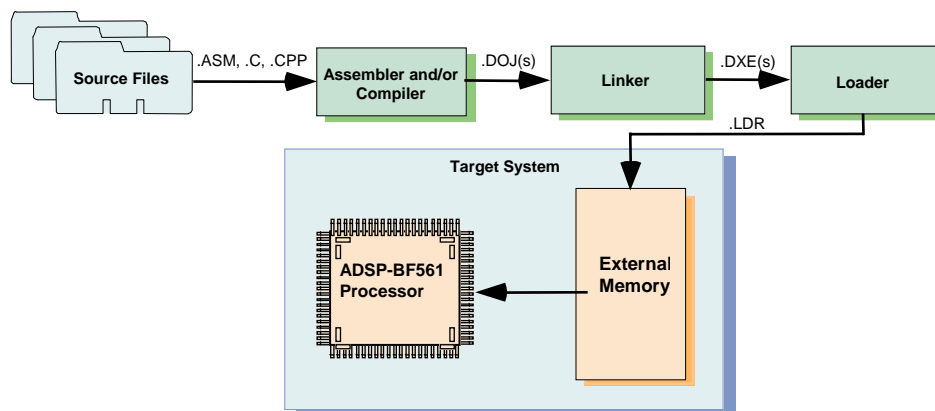


Figure 1. ADSP-BF561 Booting Sequence

Multi-Executable Management

The ADSP-BF561 processor application creates two executable files (p0.dxe and p1.dxe), one for core A and one for core B, respectively.

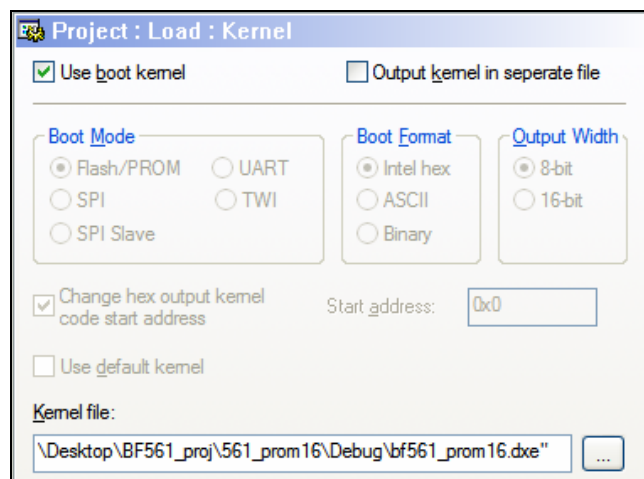


Figure 2. Kernel Options for ADSP-BF561 Processor

At any given instance, the Boot ROM can load only a single executable before it jumps to the start of core A's L1 instruction SRAM. In order to load two or more executables, a second-stage loader (SSL) must be incorporated. The SSL is used for pre-boot initialization and multi-DXE management. The default SSL, located in the \ldr directory of the VisualDSP++ 4.0 tools suite, can be selected. Alternatively, a customized kernel can be utilized by invoking

the SSL switch -l user kernel in the command line option or by changing the Load:Kernel page settings in the Project Options dialog box.

To facilitate multi-DXE loading, the executable files must be included in the Additional options box in the Load:Kernel page settings. The executables are loaded in the order specified in the command line.

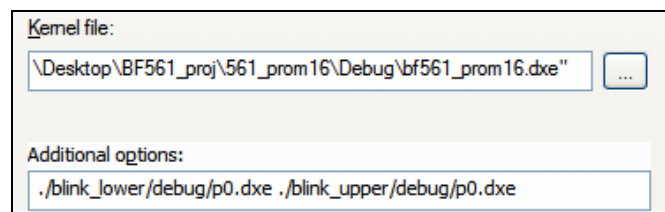





Figure 3. Additional Options in the Load:Kernel Page

-  Booting into core A scratchpad memory (0xFFB0 0000 - 0xFFB0 0FFF) and core B scratchpad memory (0xFF70 0000 - 0xFF70 0FFF) is not supported by the processor Boot ROM.
-  The .LDF file can be modified to combine both cores into a single executable. This is **not recommended** for projects that utilize shared memory or use the C/C++ run-time headers.

 The last 1024 bytes of L2 memory are allocated to the SSL by default. This part of memory **must be reserved** while in the booting mode.

Loader Operations

The loader utility converts the executable into a boot-loadable format (.LDR) that is readable by the processor Boot ROM. It also configures the output .LDR file according to a user-specified boot format (Intel hex-32, binary, ASCII) and output width (8- or 16-bit).

The loader utility is run by changing the project type from an executable to a loader file in the Project Options dialog box.

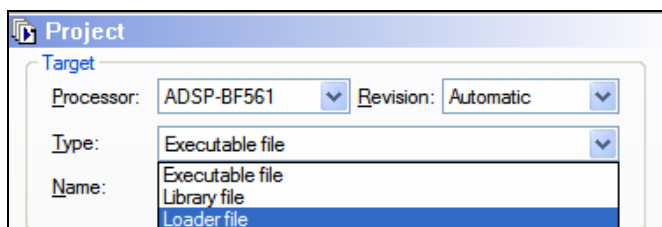


Figure 4. Project options for ADSP-BF561 processors

In order to ensure that the boot stream file (.LDR) incorporates the SSL as well as the individual executables, the “elfloader” command-line option in the Post-Build page of the Project Options dialog box must be used. For further instructions regarding how to use the elfloader command-line option, refer to the *VisualDSP++ 4.0 Loader Manual* ^[2].

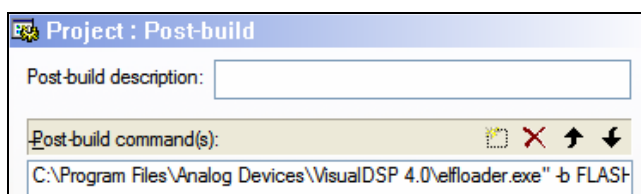


Figure 5. ADSP-BF561 Post-build Page

The file name and destination of the loader file must also be specified in the Load:Options page settings.

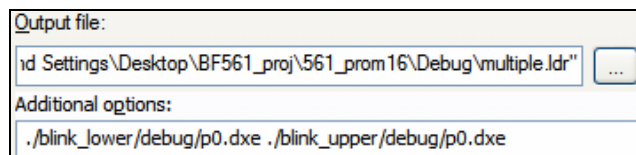



Figure 6. Output File on the Load:Options Page

ADSP-BF561 processors and ADSP-BF533 processors share the same loader file structure. Refer to *EE-240* for a detailed description of the loader file structure.

 The ADSP-BF561 boot stream begins with a 4-byte global header that contains information about the external memory device and a signature that prevents the Boot ROM from reading a blank device. The ADSP-BF533 processor’s boot stream **does not** contain a global header.

Flash Programmer Utility

After the loader file is built, it can then be programmed into flash memory using the flash programmer utility. Perform the following steps to burn the loader file into flash:

- 1) In VisualDSP++ 4.0, choose `Tools->Flash Programmer` to activate the flash programmer utility.
- 2) Select the flash programmer driver from the default location:


```
C:\Program Files\Analog Devices\VisualDSP 4.0\Blackfin\Flash Programmer Drivers\ADSP-BF561 EZ-Kit Lite\BF561EzFlash.DXE
```
- 3) After the driver has been loaded (status indicator turns green), the loader file present at the specified location is loaded into flash and verified.

For more details on flash-based applications, please refer to *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)* ^[3].

The remainder of the application note discusses how to switch between two or more executables at runtime. Tasks include modifying the C/C++ run-time header and placing memory sections

into specific locations using the Expert Linker. The software code accompanying the project

consists of the SSL and two programs that scroll different sets of LEDs.

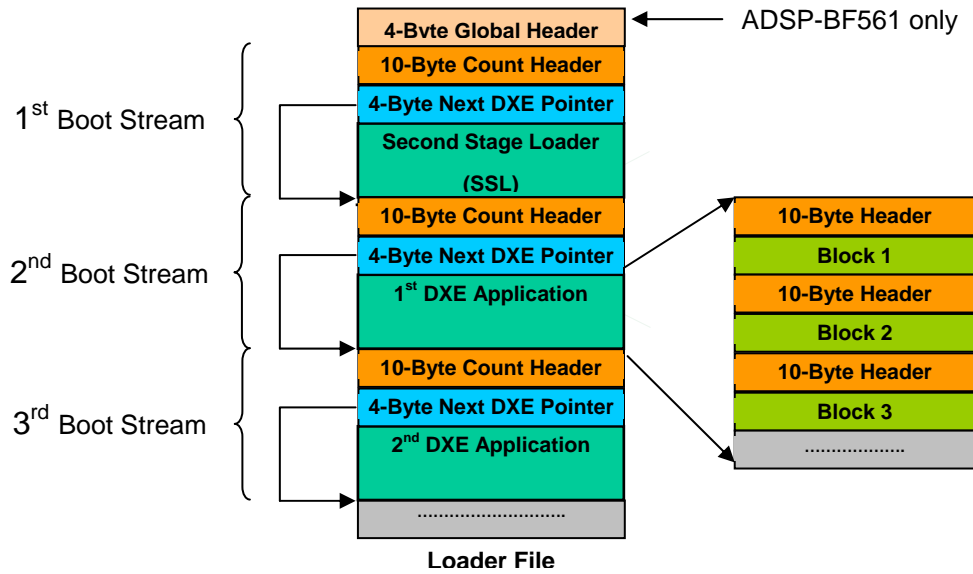


Figure 7. ADSP-BF561 Multi-DXE Loader File Structure

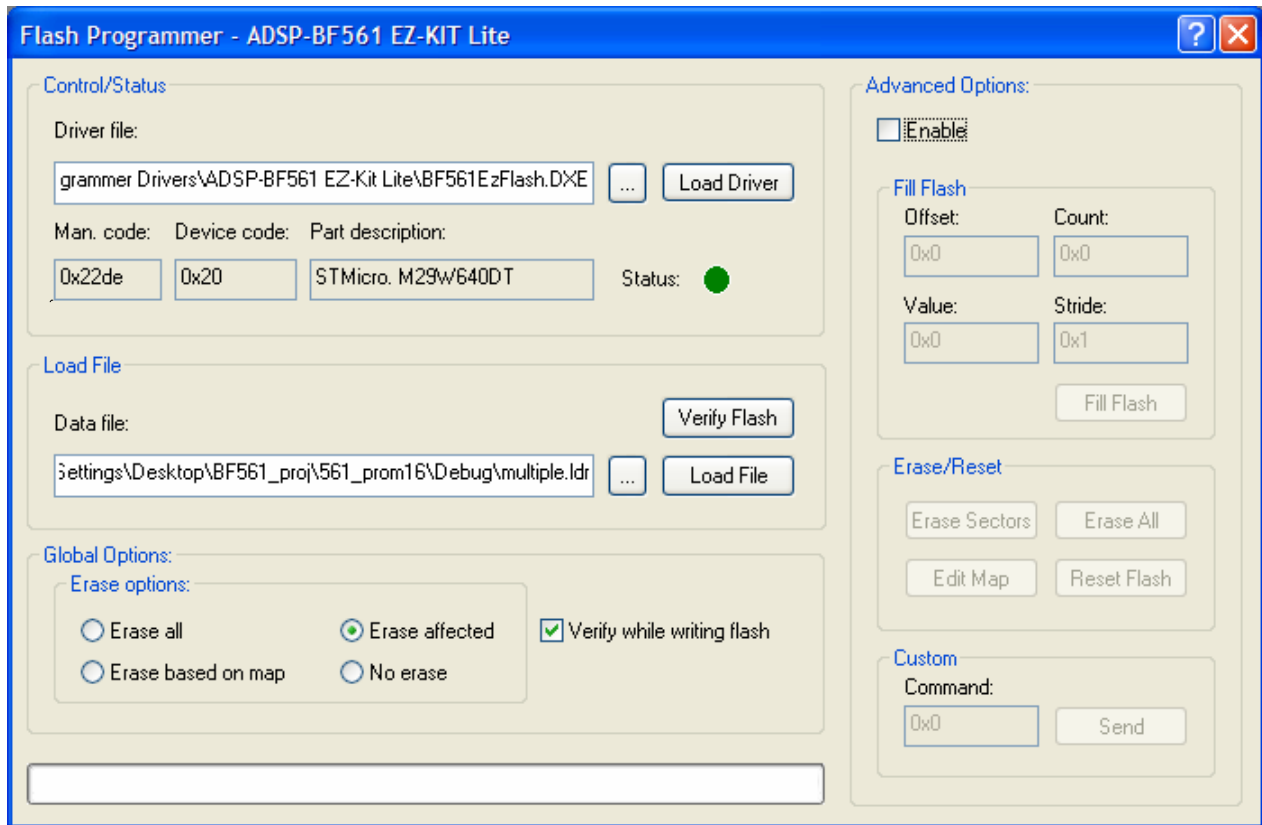


Figure 8. Flash Programmer Utility

Reconfiguring the C/C++ Run-Time Header

Attached to this EE-Note is a multi-DXE boot example (see associated ZIP file). The ZIP file contains an SSL project and two LED scroll application projects. Upon RESET, the on-chip Boot ROM boots in the SSL. The SSL then waits until PF5 (SW6 push button) or PF6 (SW7 push button) are asserted. If PF5 is asserted, the first executable that scrolls LEDs 13-20 is booted in. If PF6 is asserted, the SSL skips the first executable and boots in the second executable, which scrolls on LEDs 5-12. Either of these executables can then be booted into the device, without resetting, by pressing the push-button switches.

The 561_prom16 directory (in the ZIP file) consists of two sub-directories, blink_lower and blink_upper, which contain the sub-projects. Before the SSL is built, each of these sub-projects must be built.

These sub-directories contain project groups BF561_Blink1.dpg (to scroll LEDs 13-20) and BF561_Blink2.dpg (to scroll LEDs 5-12). When the project groups are compiled and built, they yield four executables (p0.dxe and p1.dxe for blink_lower and p0.dxe and p1.dxe for blink_upper).

The C/C++ run-time header (CRT) has been included in each of these projects. The CRT is used in projects that are coded in the C/C++ programming language and, among other tasks, initializes standard libraries such as `stdio` (standard input/output). Another of the primary functions of the CRT is to call `_main` upon completion of execution. In addition, the CRT also sets up default event handlers, enables interrupts, and sets reserved registers to known values. For more details on the CRT, refer to *Configuring the C/C++ Run-Time Header for Blackfin Processors-(EE-238)*^[4].

The CRT must be included in each of the sub-projects (that generate the executables) as part of core A and core B, as shown in Figure 9.

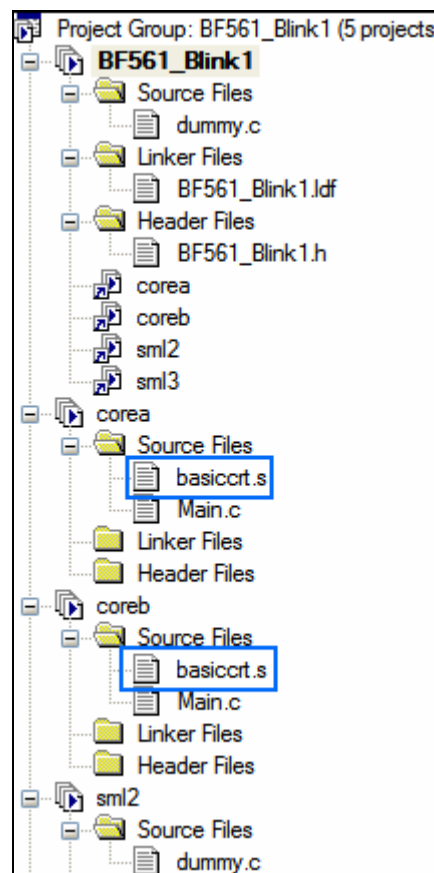


Figure 9. CRT File Inclusion in the Sub-Project

The Reconfiguration Process

By default, the CRT sets all the interrupt vector addresses (except the supervisor mode interrupt, IVG15) located in the Event Vector Table (EVT) to a generic handler's address. For the purposes of booting executables dynamically, this behavior compromises the vector address mapped to the flag interrupt used to switch applications. If the CRT overwrites the value assigned to the selected interrupt (IVG11) with the generic handler's address, it results in subsequent button-pushes vectoring to the default handler instead of the handler installed in the original application. To work around this, the `basicrt.s` file must be added to each of the sub-projects and modified.

```

// Initialize the Event Vector table.
P0.H = IVBh;
P0.L = IVBl;

// Install __unknown_exception_occurred in EVT so that
// there is defined behavior.
P0 += 2*4; // Skip Emulation and Reset
// P1 = 13;
R1.L = __unknown_exception_occurred;
R1.H = __unknown_exception_occurred;

.ivt2: [P0++] = R1;
.ivt3: [P0++] = R1;
.ivt4: [P0++] = R1;
.ivt5: [P0++] = R1;
.ivt6: [P0++] = R1;
.ivt7: [P0++] = R1;
.ivt8: [P0++] = R1;
.ivt9: [P0++] = R1;
.ivt10: [P0++] = R1;
// .ivt11: [P0++] = R1; // Not needed for maintaining IVG 11
// P0+=4; // Skipping IVG 11
.ivt12: [P0++] = R1;
.ivt13: [P0++] = R1;
.ivt14: [P0++] = R1;

```

Listing 1. EVT Modification in the basiccrt.s File

As noted in [Listing 1](#), commenting out the code that overwrites IVG11 and, instead, skipping to IVG12 solves this problem and maintains the original vector address for IVG11. Additionally, as shown in [Listing 2](#), the IMASK register must also be manually preserved. By default, the CRT enables only the Supervisor mode interrupt (IVG15). For this application, the IVG11 interrupt must also remain enabled before calling `_main`

(application program). Since no reset event is taking place, the System Interrupt Controller register's context will be preserved through the executable switch. The core's IMASK register and the EVT_x registers, on the other hand, are written by the CRT, and thus, must be preserved. The steps shown in [Listing 1](#) and [Listing 2](#) guarantee that the processor's Core Event Controller register context is also preserved.

```

// At long last, call the application program.

cli r0; //disable interrupts
bitset (r0,11); //copying the contents of IVG11 into r0 (user specified register)
sti r0; //restoring the previous state of the interrupt system

CALL.X _main;

```

Listing 2. Disabling and Re-enabling Interrupts in the basiccrt.s File

SSL Memory Management

The 561_prom directory contains the project group bf561_prom16.dpg, which consists of

bf561_prom16.dsp (containing the main SSL code) and cmds.c.

`cmds.c` is used to pass an argument to the SSL. Since a C file is included in the project, the C Application Binary Interface (ABI) must be adhered to and global data must be declared and initialized according to C/C++ standards.

`cmds.c` enables a particular `.DXE` file to be loaded without specifying its address. The SSL will index into the boot stream to find the specified `.DXE` file. The `cmds.c` module also gives the user the flexibility to load, execute, and switch `.DXE` files in a particular order. The next step involves building the SSL and creating the loader file with the multiple executables (built in the previous step). This procedure is described on page 2.

Before building the SSL, the various sections need to be placed into appropriate memory locations. None of the executables should have access to these locations and, at the same time, cores A and B should be able to access this part of memory. This can be accomplished by reconfiguring the Linker Description File (`.LDF`) of each of the executables by using the Expert Linker to ensure that there is no memory conflict with the SSL. As noted in Listing 3, L2 shared memory fulfills both these criteria and, therefore, the interrupt sections in the `cmds.c` file were placed into this memory space.

```
#include <cDefBF561.h>
#include <sysreg.h>
#include <ccblkfn.h>
#include <sys/exception.h>

void loader_commands(void);
extern int SECOND_STAGE_LOADER(int, int);

void Init_Interrupts_A(void);
EX_INTERRUPT_HANDLER(A_ISR);

/* placing the Interrupts in L2 Shared memory */
/* please note that the following programs are running on a single core-core A */

section ("l2_shared") EX_INTERRUPT_HANDLER(A_ISR)
{
    if(*pFIO0_FLAG_C & 0x0020) //SW 5 is pressed
    {
        SECOND_STAGE_LOADER(0, 1); //(0-> indicates load executable, 1-> DXE number)
        SECOND_STAGE_LOADER(1, 0); //(1-> execute the DXE)
    }
    else if(*pFIO0_FLAG_C & 0x0040) //SW 6 is pressed
    {
        SECOND_STAGE_LOADER(0, 2); //load the 2nd DXE in the boot stream
        SECOND_STAGE_LOADER(1, 0); //execute the 2nd DXE
    }
} // end

section ("l2_shared") void loader_commands()
{
    *pFIO0_DIR = 0x0000;
    *pFIO0_INEN = 0x01E0;
    *pFIO0_MASKA_D = 0x01E0;
    Init_Interrupts_A();
}
```

```

    while(1);
}

section ("l2_shared") void Init_Interrupts_A(void)
{
    *pSICA_IMASK1 |= SIC_MASK(15);
    register_handler(ik_ivg11, A_ISR);
}

```

Listing 3. *cmds.c* Source File

Reconfiguring the SSL .LDF File

After the sections of the *cmds.c* module have been placed into L2 SRAM, the same must be done for the *bf561_prom16.dsp* file. This file contains the main SSL code, consisting of two main sections (see Listing 4).

The SSL must always start in L1 memory of core A (0xFFA0 0000), hence a jump to the main program is placed at 0xFFA0 0000 to execute the code from another destination in memory.

Therefore, the `JMP_LDR` section needs to be placed at the top of core A's L1 memory.

The `SEG_LDR` section is also placed at a location in L2 SRAM for reasons outlined above. The Expert Linker can be used to view and modify the .LDF source file to place the appropriate sections in memory.

```

#define HeaderBuffer      GPStorage -12

.extern _loader_commands;

// The 2nd stage boot loader must start in Core A L1
// A jump to the program is placed at FFA00000, so we may execute in L2.

.section JMP_LDR;

    P0.L = START_OF_LOADER;
    P0.H = START_OF_LOADER;
    JUMP (P0);

////////////////////////////////////
.section SEG_LDR;
MEM_DMA:

// R0 = source address
// R1 = destination address
// R2 = count
// R3 = source config
// R4 = dest config
// modify registers have already been set up

```

Listing 4. *bf561_prom16.dsp* Source File

The Expert Linker is a powerful utility that allows you to place sections into memory graphically by dragging and dropping the

specified section. The `SECTIONS {}` command can also be used to place the specified sections by manually modifying the source file. For more

details on the syntax, refer to the description of LDF commands in the *VisualDSP++4.0 Linker and Utilities Manual* ^[5]. After incorporating the

changes detailed above, compile and execute the project group after loading the desired executables.

Appendix: Boot Modes Supported on Silicon Revisions

Silicon Revision 0.2

Silicon revision 0.2 supports true 16-bit flash/PROM mode.

BMODE[2:0]	Description
000	Reserved. Executes from external 16-bit memory connected to ASYNC Bank0 (“Bypass mode”)
001	Boot from 8/16-bit flash/PROM
010	Boot from 8-bit addressable SPI0 serial EEPROM in SPI Master mode
011	Boot from 16-bit addressable SPI0 serial EEPROM in SPI Master mode

Table 1. Silicon Revision 0.2 Boot Modes

Silicon Revision 0.3

Silicon revision 0.3 introduced SPI slave booting.

BMODE[2:0]	Description
000	Bypass
001	Boot from 8/16-bit flash/PROM
010	Boot from an SPI host in SPI Slave mode ¹
011	Boot from 16-bit addressable SPI0 serial EEPROM in SPI Master mode

Table 2. Silicon Revision 0.3 Boot Modes

¹ In silicon revision 0.3, the 8-bit SPI was replaced with slave SPI mode, but it was **non-functional** due to an anomaly; therefore, only silicon revision 0.3 supports 16-bit SPI and 8/16 flash.

Silicon Revision 0.4

BMODE[2:0]	Description
000	Bypass
001	Boot from 8/16-bit flash/PROM
010	Boot from an SPI host in SPI Slave mode ²
011	Boot from 16-bit addressable SPI0 serial EEPROM in SPI Master mode

Table 3. Silicon Revision 0.4 Boot Modes

² Silicon revision 0.4 includes support for slave SPI booting, and the bug that was found in rev 0.3 was fixed; therefore, silicon revision 0.4 supports slave SPI booting, 16-bit SPI and 8/16 flash apart from “bypass” mode.

Appendix: Jumper and DIP Switch Settings

This section describes jumper and DIP switch settings on the ADSP-BF561 EZ-KIT Lite evaluation system. The software code accompanying this application note will run successfully only if these settings are implemented.

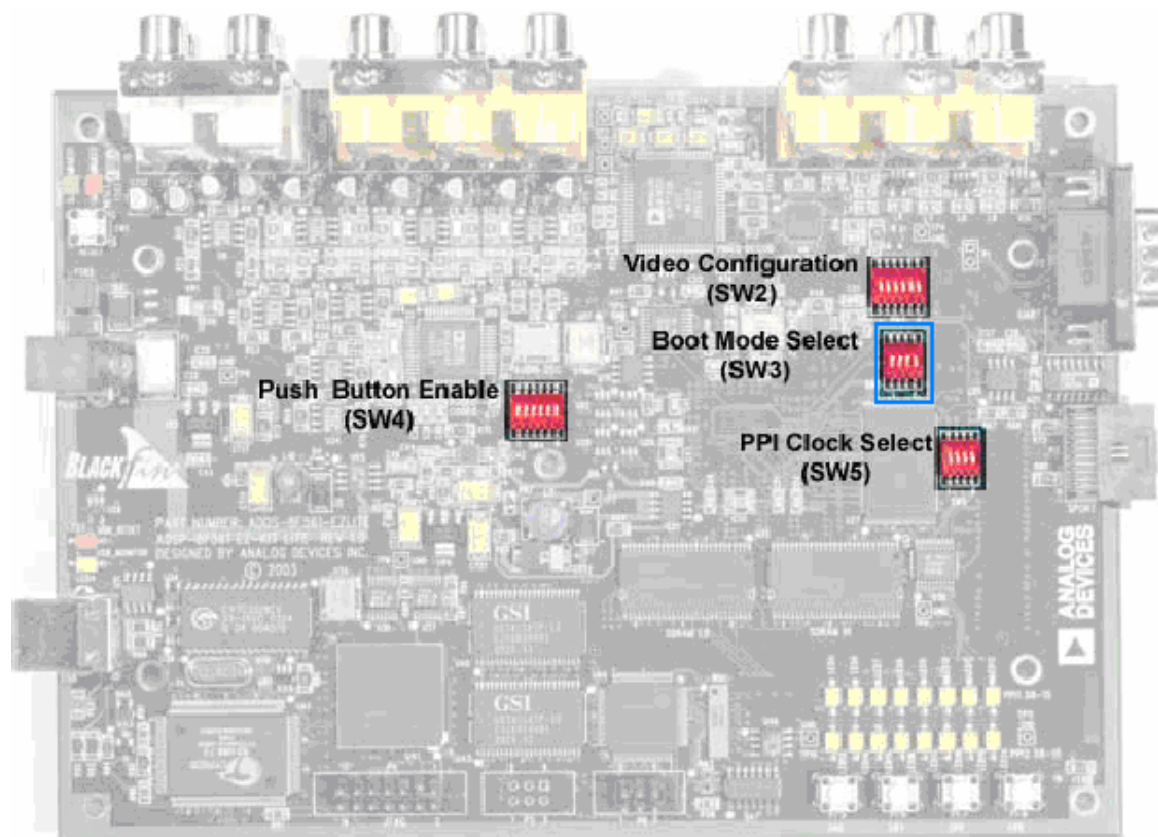


Figure 10. DIP Switch Locations on the ADSP-BF561 EZ-KIT Lite Evaluation System

As observed in Figure 10, the switch settings control different modes of operation for push buttons, video configurations, and PPI clock selections. There is no need to change any of the default settings for these switches; however, you must modify SW3, which controls the boot mode settings to accommodate this project.

Boot Mode	Position
Reserved	OFF
Flash memory	ON
8-bit SPI PROM	ON
16-bit SPI PROM	ON or OFF

Table 4. Boot Mode Select Switches for Current Project.

Positions 1 and 2 set the boot mode, whereas position 3 sets the processor's PLL in bypass mode, which is essential for this project. Position 4 can be ON or OFF, as it does not affect the functionality of the software.

References

- [1] *ADSP-BF533 Blackfin Booting Process (EE-240)* Rev 3.0, January 2005. Analog Devices, Inc.
- [2] *VisualDSP++ 4.0 Loader Manual*. Rev 1.0, January 2005. Analog Devices, Inc.
- [3] *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)*. Rev 1, May 2004. Analog Devices Inc.
- [4] *Configuring the C/C++ Run-time header for Blackfin Processors (EE-238)*. Rev 1, May 2004. Analog Devices Inc.
- [5] *VisualDSP++ 4.0 Linker and Utilities Manual*. Rev 1.0, January 2005. Analog Devices, Inc.

Document History

Revision	Description
<i>Rev 1 – July 11, 2005 by M.Aryapadi and J. Beauchemin</i>	Initial Release