Technical notes on using Analog Devices DSPs, processors and development tools
Contact our technical support at dsp.support@analog.com and at dsptools.support@analog.com
Or visit our on-line resources http://www.analog.com/ee-notes and http://www.analog.com/processors

Running Programs from Flash on ADSP-BF533 Blackfin® Processors

Contributed by Steve K *Rev 1 – May 22, 2004*

Introduction

This document describes the process of setting up an application to run on the ADSP-BF533 Blackfin® processor via flash memory, instead of being loaded from VisualDSP++® 3.5 via an emulator. It uses the ADSP-BF533 EZ-KIT Lite™ evaluation system as an example board.

Minimum requirements:

- VisualDSP++ 3.5 (base release or Update)
- ADSP-BF533 Blackfin processor, silicon revision 0.2 or higher

The EZ-KIT Lite example program "Blink" is used to demonstrate the procedures. It can be found in the `..\Blackfin\EZ-Kits\ADSP-BF533\Examples\Blink\C` directory within the VisualDSP++ installation.

The document considers two approaches:

- Running the program via the ADSP-BF533 Blackfin processor boot ROM
- Running the program directly from flash memory, via bypass mode

Initial Steps: Running Under the Emulator

Begin by coping the Blink example from the VisualDSP++ installation to a new working directory, and build it to verify that it works, as follows:

1. Copy the `..\Blackfin\EZ-Kits\ADSP-BF533\Examples\Blink\C` directory to a new location.
2. Start VisualDSP++.
3. Use `Project->Open` to open `BF533 Flags C.dpj` in the new location.
4. Rebuild all.
5. Load the program into the ADSP-BF533 EZ-KIT Lite board (via an in-circuit emulator or USB agent) and run it.
6. Observe the six LEDs, blinking on and off in sequence.

This program is running entirely from internal L1 memory, using L1 Instruction memory and L1 Data A and L1 Data B memory. No caches are enabled. SDRAM is not used.

In the `ISRs.c` file, `Timer0_ISR()`uses the code shown in Listing 1 to generate the blinking sequence.

```
    if(sLight_Move_Direction)
    {
         if((ucActive_LED = ucActive_LED >> 1) == 0x00) ucActive_LED = 0x20;
    }
    else
    {
         if((ucActive_LED = ucActive_LED << 1) == 0x40) ucActive_LED = 0x01;
    }
```

Listing 1. LED sequence in Blink application

While following the various stages of this document, you may find it helpful to modify this piece of code to produce different sequences. This will help verify that on power-up your EZ-KIT Lite board is now running a newly modified program. Listing 2 shows suggested alternative sequences.

```
    if (++ucActive_LED == 0x40)
         ucActive_LED = 1;    // alternative 1: count up

    if (ucActive_LED-- == 0)
         ucActive_LED = 0x40; // alternative 2: count down
```

Listing 2. Suggested alternative LED sequences

Booting from Flash Devices

ADSP-BF533 Blackfin processors feature an on-chip boot ROM that contains a boot kernel. If the BMODE pins are programmed in "boot from flash" mode, the kernel processes a boot stream stored in the flash memory and decomposes it into blocks of code and data. These blocks are copied to RAM areas within the memory space. The boot stream is created by the loader utility after building a `.DXE` file, and is contained within a loader file that must be programmed into the flash memory. After the boot kernel has copied the blocks to their respective destinations, the kernel jumps to the start of L1 Instruction memory and starts executing from there. Consequently, all code and data have been copied to their final destinations before the application code starts running.

Creating a Boot Stream

1. In `Project->Project Options`, on the `Project` tab, under `Target`, change `Type` to `DSP loader file`.
2. On the `Load` tab, set `Boot mode` to `Flash` and set `Output width` to `16-bit`.
3. On the `Load` tab under `Boot kernel options`, clear the `Use boot kernel` option.
4. On the `Load` tab under `ROM splitter options`, ensure that `Enable ROM splitter` is not selected.
5. Close the Project Options dialog box.
6. Save project changes via `Project->Save`.
7. Rebuild the project.

When the project is rebuilt (after creating `Debug\BF533 Flags C.dxe`), VisualDSP++ will invoke the Elfloader utility to generate the boot stream from this `.DXE` file. The boot stream will be created in the loader file `Debug\BF533 Flags C.ldr`.

Programming the Boot Stream into the EZ-KIT Lite Board

1. Select `Tools->Flash Programmer`. The Flash Programmer window opens.
2. In the `Driver File` field, browse to the file `…\Blackfin\Flash Programmer Drivers\ADSP-BF533 EZ-kit Lite\BF533EZFLASH.DXE`.
3. Click on `Load Driver`. The `Status` circle will change from red to yellow and then to green.
4. In the `Data File` field, browse to the file `Debug\BF533 Flags C.ldr` in your project.
5. Click on `Load File`. The boot stream from the loader file will be programmed into the EZ-KIT Lite board's flash memory.
6. Exit the Flash Programmer.
7. Exit VisualDSP++.
8. Power-cycle the EZ-KIT Lite board. Observe the LEDs blinking on power-up.

Using the Boot Kernel and SDRAM

If any of the application is mapped to SDRAM – such as if the `USE_CACHE` option of the standard `.LDF` files is employed – the SDRAM will need to be initialized before the boot kernel can copy a block from the boot stream into SDRAM. This can be achieved through the use of initialization blocks.

The loader utility can be given an initialization file. This is a `.DXE` file built for the target using the `ADSP-BF533_ASM.ldf` file. The loader utility extracts the executable code section (`program` section), and inserts it into the boot stream with a special marker. The marker indicates to the boot kernel that it should execute the code within the block immediately after copying it. Thus, the block can be used to set up SDRAM before application components are copied into it.

The initialization object file must contain code that operates like a function that considers all registers to be callee-preserved; it must save and restore all the registers it uses. Since this initialization code is called like a function, it must terminate with an `RTS` instruction. The boot kernel will provide sufficient stack space within the L1 scratchpad memory.

For an example of an initialization file, refer to Listing 2-1 in *VisualDSP++ 3.5 Loader Manual for 16-bit Processors* [1].

Running from Flash via Bypass Mode

When the ADSP-BF533 Blackfin processor detects that it is being configured in bypass mode at reset, it does not process a boot stream. Instead, it bypasses the boot ROM, jumps to the start address of flash memory, and starts execution directly. This means that the application code is being executed from within flash, rather than being copied to RAM and then being executed from there.

In this mode, any code or data that should be in the L1 Instruction, L1 Data A, or L1 Data B memories has not yet been copied when control is passed to the application. Generally, the application will at least need to initialize data items to their starting values before modifying them during the course of the application.

Data sections can be initialized in an automated fashion 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. The table is described in the Memory Initializer Boot Stream section.

Processing the Data Stream in Assembly Applications

The run-time libraries contain a routine for processing this table that the assembly programmer can call to cause the initialization. The function is called `_mi_initialize` (one leading underscore, from assembly). It takes no parameters and returns an integer result. Zero means successful initialization, and a negative value means that the table was invalid.

Alternatively, the assembly programmer could process the table directly. The table is pointed to by the global variable `___inits` (three leading underscores). This symbol is a pointer, and is normally a `NULL` pointer (points to `0x0000 0000`). If the `.DXE` file has been processed for run-time initialization during linking, the `___inits` symbol will be modified to point to the start of the table. See the Memory Initializer Boot Stream section for details on the format of the table.

Processing the Data Stream in C/C++ Applications

The C/C++ developer does not need to call the run-time function `mi_initialize` (no leading underscores, from C/C++) explicitly; the start-up code for the C/C++ run-time calls `mi_initialize` automatically before invoking `main`.

Limitations of mi_initialize

At the time of this writing, the run-time initialization support has two limitations:

- It does not support writing to L1 Instruction memory. Therefore, it cannot be used to copy code from flash into L1 Instruction memory.
- It does not provide any means of invoking initialization code for SDRAM. Therefore, it cannot be used to copy code or data from flash into SDRAM, unless some other means is used to enable SDRAM before the memory initialization process runs.

An assembly developer can avoid these limitations by processing the data table directly. A C/C++ developer can avoid these limitations by providing an alternative implementation of `mi_initialize`.

The “Blink” application does not use SDRAM, so the second limitation does not pose a problem for the examples in this document. To deal with the first limitation within the scope of this document, the “Blink” application will be linked so that its code runs entirely from flash memory.

Creating a Flash-Based Application

Creating a flash-based application is a two-stage process. First, a customized `.LDF` file is needed to re-map the application code and arrange for data initialization. Then the project must be configured to generate a no-boot flash image.

Creating the Flash-Based .LDF File

The “Blink” application does not have an `.LDF` file of its own; it uses the default compiler `.LDF` files. Therefore, begin by creating an `.LDF` file for the project, as follows:

1. Select `Tools->Expert Linker->Create LDF…` Click on `Next`.
2. Select the `C` project type. Click `Next`.
3. This is a single-processor ADSP-BF533 session, so the defaults presented on the next dialog suffice. Click `Next`.
4. Click `Finish`.
5. The Expert Linker will open its graphical editor. Close it.
6. Right-click the `BF533 Flags C.ldf` file in the `Project` window and select `Open with Source Window`.

Placing Code Sections in Flash

The `.LDF` file has to be changed to reflect that code will be mapped into the flash areas. This requires the following changes:

- Map section `program_ram` to `MEM_ASYNC0` (instead of `MEM_L1_CODE`).
- Map section `l1_code` to `MEM_ASYNC0` (instead of `MEM_L1_CODE_CACHE`).

Because the code is mapped to flash, it will be executable directly from the flash memory.

Placing Data in Flash or RAM

Constant Data

For constant data, there are two options: the data can be mapped directly to flash memory because it is unchanging, or it can be mapped to RAM and initialized at run-time. The former saves space, but the latter runs faster. This is an application-level decision.

In this example, we consider the `constdata` input section to be constant data that nevertheless might be frequently accessed by the application (e.g., FIR coefficients). Therefore, we will map these input sections into RAM for initialization at runtime. But the compiler also generates a number of tables that are used for C++ constructor and exception support, and those are unlikely to be referenced often, so we will map them directly into flash memory.

This is a convenient decision for this example, because `constdata` is a misnomer for the output section in the default `.LDF` files: it also maps input sections `L1_data_a`, `cplb_data`, `data1`, and `voldata`, which are not constant. If we were to resolve `constdata` into flash memory, we would have to ensure that these other input sections are separately mapped elsewhere in RAM.

We separate the `constdata` *output* section into `constdata` and `cplustables`; the `constdata` input section is mapped into the former, in `MEM_L1_DATA_A`, and the sections `ctor`, `ctorl`, `.gdt`, `.gdtl`, `.edt`, `.cht`, `.frt`, and `.frtl` are mapped into the latter, in `MEM_ASYNC0`.

Furthermore, because the `constdata` section needs to be processed at runtime to initialize it, we mark it with a `RUNTIME_INIT` *initialization qualifier*. This is used during link-time, when the run-time initialization is configured, to indicate that the section has data that needs to be saved into ROM and copied into RAM at run-time.

The two modified sections from the `.LDF` file are shown in Listing 3.

```
constdata RUNTIME_INIT
        {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS($OBJECTS(L1_data_a) $LIBRARIES(L1_data_a))
        INPUT_SECTIONS($OBJECTS(cplb_data) $LIBRARIES(cplb_data))
        INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
        INPUT_SECTIONS($OBJECTS(voldata) $LIBRARIES(voldata))
        INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
        } >MEM_L1_DATA_A


cplustables
        {
        INPUT_SECTION_ALIGN(4)
               INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor) )
               INPUT_SECTIONS( $OBJECTS(ctorl) $LIBRARIES(ctorl) )
               INPUT_SECTION_ALIGN(4)
               INPUT_SECTIONS( $OBJECTS(.gdt) $LIBRARIES(.gdt) )
               INPUT_SECTION_ALIGN(4)
               INPUT_SECTIONS( $OBJECTS(.gdtl) $LIBRARIES(.gdtl) )
               INPUT_SECTION_ALIGN(4)
               INPUT_SECTIONS( $OBJECTS(.edt) $LIBRARIES(.edt) )
               INPUT_SECTION_ALIGN(4)
               INPUT_SECTIONS( $OBJECTS(.cht) $LIBRARIES(.cht) )
               INPUT_SECTION_ALIGN(4)
               INPUT_SECTIONS( $OBJECTS(.frt) $LIBRARIES(.frt) )
               INPUT_SECTION_ALIGN(4)
               INPUT_SECTIONS( $OBJECTS(.frtl) $LIBRARIES(.frtl) )
        } >MEM_ASYNC0
```

Listing 3. Constant Data, Placed in Flash.

Variable Data

For variable data, there are also two options: the data may be initialized to zero, or it may be initialized to a non-zero constant. Both options are handled in approximately the same way; the difference between the two is the initialization qualifier.

The output sections (`data`, `l1_data_a` and `l1_data_b`) contain data that may be initialized to arbitrary constants. Each of these sections is marked with the `RUNTIME_INIT` qualifier to indicate that the contents must be copied from flash to the appropriate RAM areas (`MEM_L1_DATA_B`, `MEM_L1_DATA_A_CACHE`, and `MEM_L1_DATA_B_CACHE`, respectively). They map the input sections `data1`, `voldata`, `L1_data_a`, `L1_data_b`, and `cplb_data`, all of which will contain some initial non-zero values.

The output section `bsz` maps the `bsz` input sections, which only contain data that must be zero-filled on start-up. This output section is already marked with the `ZERO_INIT` qualifier, so the run-time initialization will just write zeroes to the section instead of copying bytes from flash. This section needs no modification.

Heap and Stack

These sections, which must be in RAM, do not need initialization qualifiers, as they are initialized explicitly by the run-time library. You may find it convenient to mark these sections as `ZERO_INIT`, to clear their contents first, though this is not necessary.

Initialization Tables

The sections `bsz_init` and `.meminit` must be mapped into flash memory. The `bsz_init` section, which is a very small, will contain 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 RAM areas.

Listing 4 shows how these sections are mapped in the `.LDF` file.

```
bsz_init
        {
         INPUT_SECTION_ALIGN(4)
             INPUT_SECTIONS( $OBJECTS(bsz_init) $LIBRARIES(bsz_init))
        } >MEM_ASYNC0
        .meminit {} >MEM_ASYNC0
```

Listing 4. Memory Initialization Tables in Flash Memory

RAM Versus ROM

The `.LDF` file declares `MEM_ASYNC0-3` as type `RAM`. This must be changed to be type `ROM`. Otherwise, the ROM splitter utility used to generate the Intel hex file will ignore the sections, and nothing will be programmed into the flash memory.

Start Address

The `.LDF` file explicitly resolves the `start` symbol to be the address of the first instruction executed. Normally, this is the start of L1 Instruction memory, but for an application that bypasses booting and goes directly to flash memory, it must be the start of flash, as shown in Listing 5.

```
    RESOLVE(start,0x20000000)
```

Listing 5. Resolving the Starting Address to the Beginning of Flash Memory

Enabling Memory Initialization

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

- If you are building your application within VisualDSP++, enable memory initialization by typing `-meminit` in the `Additional options` field of the `Link` tab on the Project Options dialog box.
- If you are building your application from the command line, enable memory initialization by adding the `-mem` switch to the `ccblkfn` command.

If run-time memory initialization is selected via one of these methods, the linker will perform an additional pass after producing the `.DXE` file, which has the following effects:

- Output sections marked as `ZERO_INIT` are recorded in an initialization table.
- Output sections marked as `RUNTIME_INIT` are recorded in the same table. In addition, the contents of these sections are added to the table.
- The initialization table is written into the `.meminit` output section.
- A pointer to the start of the table is written into the `bsz_init` output section.

When the application starts running, the following will occur:

- The run-time library will follow the pointer to the initialization table.
- For each section recorded in the initialization table, the run-time library zero-fills the section or copies starting values from the initialization table to the section, depending on the section's initialization qualifier.

Therefore, as long as the `.meminit` and `bsz_init` sections are mapped to flash (as described previously) and the sections mapped to volatile memory are marked with a suitable initialization qualifier, the run-time memory initialization will ensure that the volatile data sections are appropriately configured when the application starts running.

Creating the No-Boot Flash Image

The loader file is created in a similar manner to the boot ROM-based loader field. Since raw executable code is placed into the flash memory instead of a boot stream, the ROM splitter option is used.

In the Load tab of the Project Options dialog box:

1. Select the `ROM splitter options` category, and select `Enable ROM splitter`. Because the ROM splitter is enabled, options under the `Loader options` and `Boot kernel options` categories are ignored, so they do not need to be changed.
2. Set `Format` to `Hex`.
3. Set the `Mask address` to `29`.
4. In `Output file`, specify an output filename.
5. Click `OK`.

Now save the changes to the project, and rebuild it.

When the project is rebuilt, after creating `Debug\BF533 Flags C.dxe`, VisualDSP++ will invoke the Elfloader utility to generate the loader file (named in step 4) from this `.DXE` file.

Some versions of VisualDSP++ may generate linker warning li2131 – "Input sections of incompatible init qualifier detected in the output section" – during the linking phase. These warnings are triggered when an input section's initialization qualifier does not exactly match the output section's initialization qualifier. The run-time library's `data1` section is built without qualifiers, so mapping it into an output section with the `RUNTIME_INIT` qualifier will cause the linker to issue this warning, which can be safely ignored in this case.

Ensure that input sections are not mapped into output sections with initialization qualifiers that are incompatible for the input section's contents. For example, input sections with non-zero contents cannot be safely mapped to an output section with a `ZERO_INIT` qualifier, nor can an input section with a `ZERO_INIT` qualifier be safely mapped to an output section with a `NO_INIT` qualifier. Both of these combinations would mean that the contents would be initialized incorrectly.

Loading the Loader File onto the EZ-KIT Lite Board

1. Select `Tools->Flash Programmer…`. The Flash Programmer window opens.
2. In the `Driver File` field, browse to the file `…\Blackfin\Flash Programmer Drivers\ADSP-BF533 EZ-kit Lite\BF533EZFLASH.DXE`.
3. Click on `Load Driver`. The `Status` circle will change from red to yellow and then to green.
4. In the `Data File` field, browse to the output file previously specified via the `ROM splitter` category (on the `Load` page).
5. Click on `Load File`. The loader file will be programmed into the EZ-KIT Lite board's flash memory.
6. Exit the Flash Programmer.
7. Exit VisualDSP++.
8. Power-cycle the EZ-KIT Lite board.

On power-up, the boot ROM will attempt to interpret the flash contents as a boot stream. Since the flash contents are not a boot stream, the booting process will fail. Instead, the flash contents must be executed directly, as described below.

Running the Program in Flash Memory

For diagnostic purposes, the program in flash memory can be invoked under emulator control, using the program shown in Listing 6.

```
int main(void)
{
    void (*program)(void) = (void (*)(void))(0x20000000);
    (*program)();
}
```

Listing 6. Running a Program in Flash Memory.

To execute on power-up, however, the processor must be reset in bypass mode. This is done on a Revision 1.2 EZ-KIT Lite board by resetting with R11 shorted out. Recent revisions of EZ-KIT Lite boards provide a jumper.

R11 is located to the left of the Blackfin processor (moving left from the Blackfin processor, you will find C198, R11, and R10, in order). R11 is just two contacts, with no connector between them. To boot the processor in bypass mode, short the gap between the two contacts while power-cycling or pressing the Reset switch (top left corner of the EZ-KIT Lite board). Observe that the lights start blinking when the program starts running.

Performance When Executing From Flash Memory

The flash memory described in this EE-Note is connected directly to the External Bus Interface Unit (EBIU). At reset, the settings for this Asynchronous Memory bank default to the largest number of wait states and hold states to account for the possibility that a slow flash memory will be connected. At these maximum settings, the maximum rate of processor execution is one 16-bit instruction every 15 System

Clock (SCLK) cycles. At reset, the SCLK value on the ADSP-BF533 EZ-KIT Lite board is 54 MHz (27 MHz * 10 MSEL/5 SSEL).

In the Blackfin Instruction Set Architecture[1], the most frequently used instructions are encoded as 16-bits. However, the instruction set also contains 32-bit and 64-bit instructions. Each instruction fetch from flash is 16 bits in size, so a 64-bit instruction will take four 16-bit fetches. Because flash is an "asynchronous" memory, turning instruction cache on and making the flash "cacheable" will not improve performance the first time code is executed. If, however, the code from flash is re-executed once it has been brought into cache, subsequent execution will enjoy single-cycle throughput.

If the flash connected in a system supports fewer wait states and the hold time is lower, the maximum instruction fetch rate from 16-bit flash is one 16-bit fetch every two SCLKs. This can be accomplished by using the init booting function described in the *VisualDSP++ 3.5 Loader Manual for 16-bit Processors* [1] to configure the EBIU appropriately.

Another way to improve performance is to use overlays. With this method, code segments are transferred into internal memory using DMA before the actual code is executed. The memory DMA controller is used to bring code into one memory bank while the processor executes code from another memory bank.

For further information on the EBIU, refer to the *ADSP-BF533 Blackfin Processor Hardware Reference* [2].

Memory Initializer Boot Stream

The Memory Initializer utility is invoked at link-time, when the compiler's `-mem` switch is used or when the linker's `-meminit` switch is used. The Memory Initializer utility processes the linked application and builds a boot stream within the `.meminit` section. The format of this stream is described by the structures defined in the `<meminit.h>` header file, and is explained here.

The `___inits` Pointer

The start of the boot stream is the global pointer `___inits` (in assembly, with three leading underscores) or `__inits` (in C, with two leading underscores). In an application that has not been processed by the Memory Initializer utility, this pointer has the value 0, indicating that there is no boot stream to process. When the Memory Initializer utility processes the application, it creates the boot stream and then updates the value of this global pointer to point to the start of the boot stream.

The pointer's C declaration is shown in Listing 7.

```
typedef struct mi_table_header_t mi_table_header;
extern mi_table_header *__inits;
```

Listing 7. C Declaration of __inits Pointer.

The Table Header

The table header indicates the format and size of the following boot stream. It is defined using the structures listed in Listing 8.

[1] See the *Blackfin Processor Instruction Set Reference* [3].

```
struct mi_magic_t {
  unsigned char version;
  unsigned char reserved;
  unsigned short magic;
};

struct mi_table_header_t {
  struct mi_magic_t magic;
  unsigned long num_blocks;
  struct mi_block_header_t blocks[1];    /* arbitrary number of blocks */
};

typedef struct mi_magic_t mi_magic;
```

Listing 8. Memory Initializer Header Structures

The `magic` field identifies the format of the boot stream. For Blackfin processors, the only allowed values for the `mi_magic_t` structure's fields are:

Field	***Value***
`version`	0
`reserved`	Reserved for future use
`magic`	`MI_MAGIC_BLACKFIN` (`0xFFFF`)

Table 1. mi_magic_t Fields

The `blocks[]` field is declared as an array that is one block in length. In reality, it indicates the position of the first block in the stream. `num_blocks` indicates the total number of blocks in the stream, and therefore gives the true size of the `blocks[]` field.

Block Structure

Each block in the boot stream defines a portion of memory to be initialized, either by copying data (including code) from the boot stream into the area of memory or by zero-filling the area of memory.

The block structure is defined as shown in Listing 9.

```
struct mi_block_header_t {
  char *addr;
  unsigned long byte_count;
  unsigned long flags;
  unsigned long pattern_bytes;
};
```

Listing 9. Memory Initializer Block Structure

The `flags` field is divided into several sub-fields:

Bits	Use
0-1	Memory type: `MI_MEM_US` (`0x02`)
2-5	Word Size: `MI_WS_8BITS` (`0x00`)
6-8	Block kind
9-31	Reserved for future use

Table 2. Flags Sub-Fields

The *memory type* indicates the kind of memory the block is initializing. For Blackfin, this is always set to `MI_MEM_US` (`0x02`), which indicates that the memory is not of a fixed type, (e.g., Program Memory or Data Memory).

The *word size* indicates the addressable units for the memory being initialized. Since Blackfin is a byte-addressed architecture, this field is always `MI_WS_8BITS` (`0x00`).

The *block kind* indicates how the block should be interpreted to initialize the memory area.

Block Kinds

The block kind indicates what kind of initialization is required for the memory area. There are several possible values:

Value	**Meaning**
`MI_BT_RAW (0x00)`	The memory area will be initialized by copying data verbatim from the boot stream to the memory area.
`MI_BT_ZERO (0x01)`	The memory area will be initialized by zero-filling the memory area.
`MI_BT_REP (0x03)`	The memory area will be initialized by filling it with a repeating pattern from the boot stream.

Table 3: Block kinds

Note that value `0x02` is not used.

Raw Block Kind

For a raw block, the significant fields are `addr` and `byte_count`. Within the boot stream, the block will be immediately followed by `byte_count` bytes of data which are to be copied to the address `addr`. If there are any more blocks within the boot stream, the next block will follow the data for the raw block.

Zero Block Kind

For a zero block, the significant fields are `addr` and `byte_count`. A zero block indicates that the memory area starting at address `addr` is `byte_count` bytes in size and should be zero-filled. The block has no following data and, if there are more blocks in the boot stream, the next block will immediately follow this zero block.

Repeating Block Kind

For a repeating block, the significant fields are `addr`, `byte_count`, and `pattern_bytes`. The block will be followed by `pattern_bytes` bytes of data, which should be copied to the `byte_count` bytes of memory starting at address `addr`. If there are more blocks in the boot stream, the next block will follow after the repeated-pattern data. For VisualDSP++3.5, the repeated patterns are restricted to the normal data type sizes: 1, 2, or 4 bytes in length. For future versions of VisualDSP++, the Memory Initializer utility may be able to detect longer sequences of repeated patterns. There is no guarantee that `byte_count` will be a multiple of `pattern_bytes`; the pattern should be copied repeatedly to the memory area until `byte_count` bytes of data have been copied.

Following Blocks

For Blackfin processors, a block is always aligned on a 4-byte boundary within the flash device. This is achieved as follows:

- The first block is part of the table header, which is always aligned. Therefore, the first block is aligned.
- For zero blocks, there is no data between the zero block and the following block, so the following block is always aligned.
- For raw blocks and repeating blocks, there is data between the block and the following block, which may not be of a suitable length to correctly align the following block. If `byte_count` (for raw blocks) or `pattern_bytes` (for repeating blocks) is not a multiple of 4, then 1-3 bytes of padding will be inserted between the data and the start of the following block to ensure the following block's alignment.

Initializing L1 Instruction Memory

If the `.LDF` file sections that map code to L1 Instruction Memory are marked as `RUNTIME_INIT`, the Memory Initializer utility will create blocks in the boot stream that initialize these sections as well. At the time of this writing, the `mi_initialize` run-time library function does not support this, as L1 Instruction Memory may not be written to as normal data. If an application developer needs to initialize L1 Instruction Memory, the boot stream will have to be processed explicitly, using the cache control registers to write data into L1 Instruction Memory.

References

[1] *VisualDSP++ 3.5 Loader Manual for 16-bit Processors.* Rev 1.0, October 2003. Analog Devices, Inc.

[2] *ASDP-BF533 Blackfin Processor Hardware Reference.* Rev 1.0, December 2003. Analog Devices, Inc.

[3] *Blackfin Processor Instruction Set Reference, Rev 1.0.* March 2002. Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – May 22, 2004 by Steve K.	Initial Release