VISUALDSP++™ 3.5 Loader Manual for 16-Bit Processors

Revision 1.0, October 2003

Part Number
82-000035-04

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

CONTENTS

PREFACE

Thank you for purchasing Analog Devices development software for digital signal processor (DSP) applications.

Purpose of This Manual

The *VisualDSP++ 3.5 Loader Manual for 16-Bit Processors* contains information on how to use the loader/splitter to convert executable files into boot-loadable (or non-bootable) files for 16-bit fixed-point ADSP-21xx DSPs and Blackfin® processors. These files are then programmed/burned into an external memory device within your target system.

Intended Audience

The primary audience for this manual is DSP programmers who are familiar with Analog Devices DSPs. This manual assumes that the audience has a working knowledge of the appropriate DSP architecture and instruction set. Programmers who are unfamiliar with Analog Devices DSPs can use this manual but should supplement it with other texts, such as *Hardware Reference* and *Instruction Set Reference* manuals, that describe your target architecture.

Manual Contents

The manual contains:

- Chapter 1, "Introduction"
- Chapter 2, "Blackfin Processor Loader/Splitter"
- Chapter 3, "ADSP-219x DSP Loader/Splitter"
- Chapter 4, "ADSP-2192-12 DSP Loader"
- Chapter 4, "ADSP-218x DSP Loader/Splitter"
- Appendix A, "File Formats"

Technical or Customer Support

You can reach DSP Tools Support in the following ways.

- Visit the DSP Development Tools website at `www.analog.com/technology/dsp/developmentTools/index.html`
- Email questions to `dsptools.support@analog.com`
- Phone questions to 1-800-ANALOGD
- Contact your ADI local sales office or authorized distributor
- Send questions by mail to:

```
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

Supported Processors

The name “*ADSP-21xx*” refers to two families of Analog Devices 16-bit, fixed-point processors. VisualDSP++ for ADSP-21xx DSPs currently supports the following processors.

- ADSP-218x family DSPs: ADSP-2181, ADSP-2183, ADSP-2184/84L/84N, ADSP-2185/85L/85M/85N, ADSP-2186/86L/86M/86N, ADSP-2187L/87N, ADSP-2188L/88N, and ADSP-2189M/89N
- ADSP-219x family DSPs: ADSP-2191, ADSP-2192-12, ADSP-2195, ADSP-2196, ADSP-21990, ADSP-21991, and ADSP-21992

The name “*Blackfin*” refers to a family of Analog Devices 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin processors.

- Blackfin Processors: ADSP-BF531, ADSP-BF532 (formerly ADSP-21532), ADSP-BF533, ADSP-BF535 (formerly ADSP-21535), ADSP-BF561, and AD6532

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices website that allows customization of a webpage to display only the latest information on products you are interested in. You can also choose to receive weekly email notification containing updates to the webpages that meet your interests. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration:
Visit `www.myanalog.com` to sign up. Click Register to use MyAnalog.com. Registration takes about five minutes and serves as means for you to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your email address.

Embedded Processor and DSP Product Information

For information on digital signal processors, visit our website at `www.analog.com/processors`, which provides access to technical publications, datasheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- Email questions or requests for information to
  `dsp.support@analog.com`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49 (0) 089 76 903 557 (Europe)
- Access the Digital Signal Processor Division's FTP website at
  `ftp ftp.analog.com` or ftp 137.71.23.21
  `ftp://ftp.analog.com`

Related Documents

For information on product related development software, see the following publications.

VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors

VisualDSP++ 3.5 User's Guide for 16-Bit Processors

VisualDSP++ 3.5 Product Release Bulletin for 16-Bit Processors

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for ADSP-219x DSPs

VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x DSPs

VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for Blackfin Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs

VisualDSP++ 3.5 Kernel (VDK) User's Guide for 16-Bit Processors

VisualDSP++ 3.5 Component Software Engineering User's Guide for 16-Bit Processors

Quick Installation Reference Card

For hardware information, refer to your DSP's *Hardware Reference* manual and datasheet.

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system and tools manuals, Dinkum Abridged C++ library, and FlexLM network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary `.PDF` files for the tools manuals are also provided.

A description of each documentation file type is as follows.

File	Description
.CHM	Help system files and VisualDSP++ tools manuals.
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the .HTML files require a browser, such as Internet Explorer 4.0 (or higher).
.PDF	VisualDSP++ manuals in Portable Documentation Format, one .PDF file for each manual. Viewing and printing a .PDF file require a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by rerunning the Tools installation.

Access the online documentation from the VisualDSP++ environment, Windows Explorer, or Analog Devices Web site.

From VisualDSP++

- Access VisualDSP++ online Help from the Help menu's Contents, Search, and Index commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

From Windows

In addition to shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (.CHM files) are located in the Help folder, and .PDF files are located in the Docs folder of your VisualDSP++ installation. The Docs folder also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click any file that is part of the VisualDSP++ documentation set.
- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.

Using the Windows Start Button

Access VisualDSP++ online Help by clicking the Start button and choosing Programs, Analog Devices, VisualDSP++ for 16-bit processors , and VisualDSP++ Documentation.

From the Web

To download the tools manuals, point your browser at `www.analog.com/technology/dsp/developmentTools/gen_purpose.html`.

Select a DSP family and book title. Download archive (`.ZIP`) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at 1-800-ANALOGD (1-800-262-5643) and follow the prompts.

VisualDSP++ Documentation Set

VisualDSP++ manuals may be purchased through Analog Devices Customer Service at 1-781-329-4700; ask for a Customer Service representative. The manuals can be purchased only as a kit. For additional information, call 1-603-883-2430.

If you do not have an account with Analog Devices, you will be referred to Analog Devices distributors. To get information on our distributors, log onto `http://www.analog.com/salesdir/continent.asp`.

Hardware Manuals

Hardware reference and instruction set reference manuals can be ordered through the Literature Center or downloaded from the Analog Devices Web site. The phone number is 1-800-ANALOGD (1-800-262-5643). The manuals can be ordered by a title or by product number located on the back cover of each manual.

Datasheets

All datasheets can be downloaded from the Analog Devices Web site. As a general rule, any datasheet with a letter suffix (L, M, N) can be obtained from the Literature Center at 1-800-ANALOGD (1-800-262-5643) or downloaded from the Web site. Datasheets without the suffix can be downloaded from the Web site only—no hard copies are available. You can ask for the datasheet by a part name or by product number.

If you want to have a datasheet faxed to you, the phone number for that service is 1-800-446-6212. Follow the prompts and a list of datasheet code numbers will be faxed to you. Call the Literature Center first to find out if requested datasheets are available.

Contacting DSP Publications

Please send your comments and recommendation on how to improve our manuals and online Help. You can contact us at `dsp.techpubs@analog.com`.

Notation Conventions

The following table identifies and describes text conventions used in this manual.

ⓘ Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu)	Text in **bold** style indicates the location of an item within the VisualDSP++ environment's menu system. For example, the **Close** command appears on the **File** menu.
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
ⓘ	A note, providing information of special interest or identifying a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
🛇	A caution, providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word **Caution** appears instead of this symbol.

ⓘ Code has been formatted to fit this manual's page width.

1 INTRODUCTION

The majority of this manual describes the loader program (or loader utility) as well as the process of loading and splitting, the final phase of a DSP application program's development flow. The process of initializing on-chip and off-chip memories, is often referred to as *booting*.

The majority of this chapter applies to all 16-bit processors. Information applicable to a particular target processor, or to a particular processor family, is provided in the following chapters.

- Chapter 2, "Blackfin Processor Loader/Splitter" on page 2-1
- Chapter 3, "ADSP-219x DSP Loader/Splitter" on page 3-1
- Chapter 4, "ADSP-2192-12 DSP Loader" on page 4-1
- Chapter 5, "ADSP-218x DSP Loader/Splitter" on page 5-1

Program Development Flow

The flow can be split into three phases:

1. "Compiling and Assembling"
2. "Linking"
3. "Loading and Splitting"

A brief description of each phase is as follows.

Compiling and Assembling

Input source files are compiled and assembled to yield object files. Source files are text files containing C/C++ code, compiler directives, possibly a mixture of assembly code and directives, and, typically, preprocessor commands. Refer to the *VisualDSP++ 3.5 Assembler and Preprocessor Manual* or the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual* for information about the assembler and compiler source files.

Linking

Under the direction of the Linker Description File (LDF) and linker settings, the linker consumes separately assembled object and library files to yield an executable file. If specified, shared memory and overlay files are also produced. The linker output conforms to the Executable and Linkable Format (ELF), an industry-standard format for executable files. The linker also produces map files and other embedded information used by the debugger (DWARF-2).

These executable files (`.DXE`) are not readable by the processor hardware directly. They are neither supposed to be burned onto a EPROM or Flash memory device. Executable files are consumed by VisualDSP++ debugging targets, such as the simulator or emulator. Refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors* and online Help for information about linking and debugging.

Loading and Splitting

Upon completing the debug cycle, the processor hardware needs to run on its own, without any debugging tools connected. After power-up, processor memories need to be initialized to be booted. Therefore, the linker output must be transformed to a format readable by the processor. This process is handled by the loader/splitter utility. The loader/splitter uses the debugged and tested executable as well as shared memory and overlay files as inputs to yield a processor-loadable file.

VisualDSP++ includes two loader/splitter programs:

- `elfloader.exe` for ADSP-BF5xx and ADSP-219x processors
- `elfspl21.exe` for ADSP-218x processors

You can run the loader/splitter from the IDDE. In order to do so, change you project's type from **DSP Executable** to **DSP Loader File**. If preferred, the command-line interface is also available.

Loader operations depend on loader options, which control how the loader processes executable files, letting you select features such as kernels, boot modes, and output file formats. These options are set on the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment or on the loader's command line. Option settings on the **Load** page correspond to switches typed on the command line.

The loader/splitter output is either a boot-loadable or non-bootable file (described in the following "Boot-loadable Files Versus Non-bootable Files"). The output is meant to be loaded onto the target. There are several ways to use the output:

- Download the loadable file into the processor's PROM space on an EZ-KIT Lite board via the Flash Programmer plug-in. Refer to VisualDSP++ Help or the *EZ-KIT Lite Evaluation System Manual* for information on the Flash Programmer.
- Use VisualDSP++ to simulate booting in a simulator session (where supported). Load the loader file and then reset the processor to debug the booting routines. No hardware is required: just point to

the location of the loader file, letting the simulator to do the rest. You can step through the boot kernel code as it brings the rest of the code into memory.

- Store the loader file in an array on a multiprocessor system. A master (host) processor has the array in its memory, allowing a full control to reset and load the file into the memory of a slave processor.

Boot-loadable Files Versus Non-bootable Files

A boot-loadable file is transported into and run from a processor's internal memory (on-chip boot ROM). (Note: This is different for ADSP-218x processors.) The file is then programmed (burned) into an external memory device within your target system. The loader outputs files in industry-standard file formats, such as Intel hex-32 and Motorola S, which are readable by most EPROM burners. For advanced usage, other file formats are supported.

A non-bootable EPROM-image file executes from the processor's external memory, bypassing the build-in boot mechanisms. Preparing a non-bootable EPROM image is called *splitting*. In most cases, developers working with 16-bit processors use the loader instead of the splitter.

A processor's booting sequence and an application program's design dictate the way you call the loader/splitter programs to consume and transform executables. For 16-bit processors, splitter and loader features are handled by a single program. The splitter is invoked by a completely different set of command-line switches than the loader. Refer to the guide sections of the following chapters for information about splitting.

Booting Modes

A fully debugged program can be automatically downloaded to the processor after power-up or after a software reset. This process is called *booting*. The way the loader creates a boot-loadable file depends upon how your program is booted into the processor.

Once an executable is fully debugged, it is ready to be converted into a processor-loadable file.

The exact boot mode of the processor is determined by sampling one or more of input flag pins. Booting sequences, highly processor-specific, are detailed in the following chapters.

ADSP-218x, ADSP-219x, and Blackfin processors support different boot mechanisms. Generally spoken, the following schemes can be used to provide program instructions to the processors after reset.

- "No-boot Mode"
- "PROM Booting Mode"
- "Host Booting Mode"

No-boot Mode

The processors starts fetching and executing instructions from EPROM/Flash memory devices directly. This scheme does not require any loader mechanism. It is up to the user program to initialize volatile memories.

The splitter utility helps to generate a file that can be burned into the PROM memory.

PROM Booting Mode

After reset, the processor starts reading data from any parallel or serial PROM device. The PROM stores a formatted boot stream rather than raw instruction code. Beside application data, the boot stream contains additional data, such as destination addresses and word counts. A small program called *kernel* or *loader kernel* (described on page 1-7) parses the boot stream and initializes memories accordingly. The loader kernel runs on the target processors. Depending on the architecture, the loader kernel may execute from on-chip boot ROM or may be pre-loaded from the PROM device into on-chip SRAM and execute from there.

The loader utility generates the boot stream from the linker's executable file and stores it to file format that can be burned into the PROM.

Host Booting Mode

In this scheme, the target processor is slave to a host system. After reset, the processor delays program execution until it gets signalled by the host system that the boot process has completed. Depending on hardware capabilities, there are two different methods of host booting. In the first case, the host system has full control over all target memories. It halts the target while it is initializing all memories as required. In the second case, the host communicates by a certain handshake with the loader kernel running on the target processor. This kernel may execute from on-chip ROM or may be pre-loaded by the host devices into the target's SRAM by any boot-strapping scheme.

The loader/splitter utility generates a file that can be consumed by the host device. It depends on the intelligence of the host device and on the target architecture whether the host expects raw application data or a formatted boot stream.

In this context, a boot-loadable file is a file that stores instruction code in a formatted manner in order to be processed by a boot kernel. A non-bootable file stores raw instruction code. Note that in some case, a single file may contain both types of data.

Boot Kernels

A (*loader) boot kernel* refers to the resident program in the boot ROM space responsible for booting the processor. Alternatively (or in absence of the boot ROM), the boot kernel can be pre-loaded from the boot source by a boot-strapping scheme.

When a reset signal is sent to the processor, the processor starts booting from a PROM, host device, or through a communication port. For example, a ADSP-218x/219x processor brings a 256-word program in internal memory for execution. This small program is called a *boot kernel*. The boot kernel then brings the rest of the booting routines into the processor's memory. Finally, the boot kernel overwrites itself with the final block and jumps to the beginning of the application program.

On the ADSP-219x DSPs, the highest 16 locations in page 0 program memory and the highest 272 locations in page 0 data memory are reserved for use by the ROM boot routines (typically for setting up DMA data structures and for bookkeeping operations). Ensure that the boot sequence entry code or boot-loaded program do not need to initialize this space at boot time. However, the program can use these locations at run-time.

Some of the newer Blackfin processors (ADSP-BF531, ADSP-BF532, and ADSP-BF533) do not require a boot kernel: the advanced on-chip boot ROM allows the entire application program body to be booted into the internal memory of the processor. The on-chip boot ROM for the former processors behaves similar to the second-stage loader of ADSP-BF535 processors. The boot ROM has the capability to parse address and count information for each bootable block.

Loader Tasks

Common tasks perform by the loader include:

- Processing loader option settings or command-line switches.
- Formatting the output `.LDR` file according to user specifications. Supported formats are binary, ASCII, hex-32, and more. Valid file formats are described in Appendix A on page A-1.
- Packing the code for a particular data format: 8- or 16-bit.
- If specified, adding a boot kernel on top of the user code.
- If specified, preprogramming the location of the `.LDR` file in PROM space.
- Specifying processor IDs for multiple input `.DXEs` for a multiprocessor system.

Loader Files

The loader/splitter output is essentially the same executable code as in the input `.DXE` file. The loader repackages the executable, as illustrated in Figure 1-1.

Processor code in a loader file is split into blocks. Each code block is marked with a tag that contains information about the block, such as a number of words or destination in processor's memory. Depending on the processor family, there may be additional information in the tag. Common block types are "zero" (memory is filled with 0s); non-zero (code or data); and final (code or data). Depending on the processor family, there may be other block types.

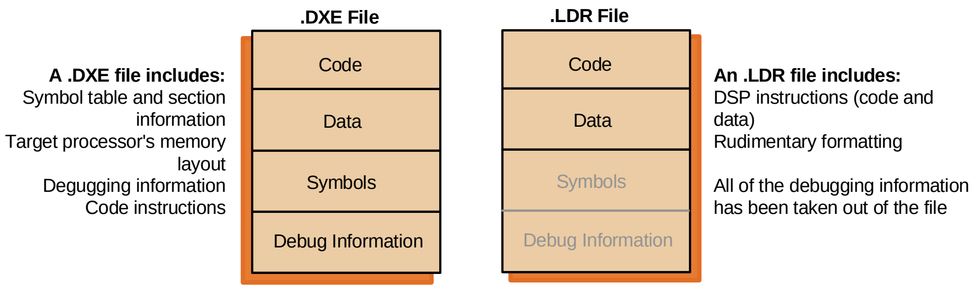


Figure 1-1. .DXE Files versus .LDR Files

File Searches

File searches are important in the loader operation. The loader supports relative and absolute directory names, default directories. File searches occur as follows.

- Specified path—If you include relative or absolute path information in a file name, the loader searches only in that location for the file.
- Default directory—If you do not include path information in the file name, the loader searches for the file in the current working directory.
- Overlay and shared memory files—the loader recognizes overlay memory files but does not expect these files on the command line. Place the files in the same directory as the executable file that refers to them. The loader can locate them when processing the executable.

When providing an input or output file as a loader/splitter command-line parameter, use the following guidelines.

- Enclose long file names within straight quotes, “`long file name`”.
- Append the appropriate file extension to each file.

2 BLACKFIN PROCESSOR LOADER/SPLITTER

This chapter explains how the loader/splitter program (`elfloader.exe`) is used to convert executable files (`.DXE`) into boot-loadable or non-bootable files for the ADSP-BF5xx Blackfin processors.

Refer to "Introduction" on page 1-1 for the loader overview; the introductory material applies to all processor families. Loader operations specific to ADSP-BF5xx Blackfin processors are detailed in the following sections.

- "Blackfin Processor Booting" on page 2-2

  Provides general information on various booting modes, including information on second-stage kernels:

  - "ADSP-BF535 Processor Booting" on page 2-3
  - "ADSP-BF531/BF532/BF533 Processor Booting" on page 2-16
  - "ADSP-BF561 Processor Booting" on page 2-28

- "Blackfin Processor Loader Guide" on page 2-40

  Provides reference information on the loader's command-line syntax and switches.

Blackfin Processor Booting

Figure 2-1 is a simplified view of the Blackfin processor's booting sequence.

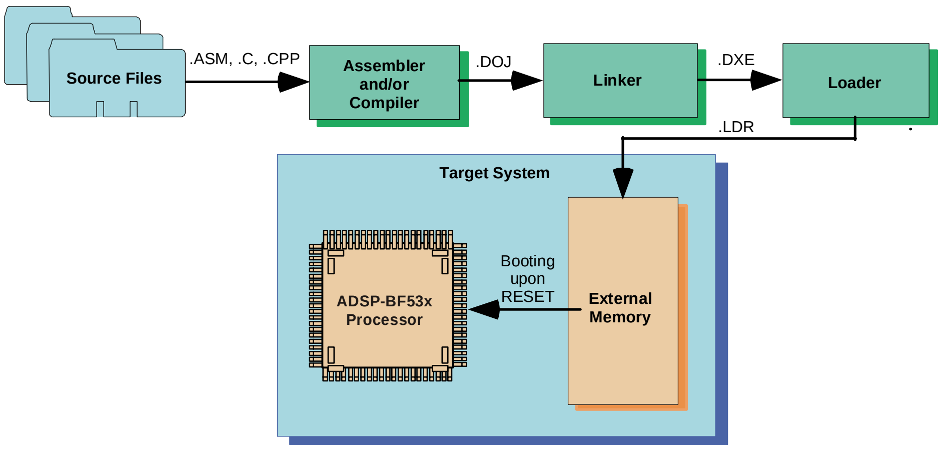


Figure 2-1. Blackfin Processors: Booting Sequence

A Blackfin processor can be booted from an 8- or 16-bit Flash/PROM memory or an 8-,16-, or 24-bit addressable SPI memory. (24-bit addressable SPI memory booting supported only on ADSP-BF531/BF532/BF533 processors.) There is also a no-boot option (bypass mode), in which execution occurs from a 16-bit external memory.

At powerup, after the reset, the processor transitions into a boot mode sequence configured by the `BMODE` pins. These pins can be read through bits in the System Reset Configuration Register (`SYSCR`). The `BMODE` pins are dedicated mode-control pins; that is, no other functions are shared with these pins.

ⓘ Refer to the processor's data sheet and *Hardware Reference* for more information on system configuration, peripherals, registers, and operating modes.

ADSP-BF535 Processor Booting

Upon reset, an ADSP-BF535 processor jumps to an external 16-bit memory for execution (if `BMODE = 000`) or to the on-chip boot ROM (if `BMODE = 001`, `010`, `011`). Table 2-1 summarizes booting modes and code execution start addresses for ADSP-BF535 processors.

Table 2-1. ADSP-BF535 Processor Boot Mode Selections

Boot Source	BMODE[2:0]	Execution Start Address
Execute from 16-bit external memory (Async Bank 0); no-boot mode (bypass on-chip boot ROM)	000	0x2000 0000
Boot from 8-bit/16-bit Flash memory	001	0xF000 0000[1]
Boot from 8-bit address SPI0 serial EEPROM	010	0xF000 0000[1]
Boot from 16-bit address SPI0 serial EEPROM	011	0xF000 0000[1]
Reserved	111–100	N/A

1 The processor jumps to this location after the booting is complete.

A description of each boot mode is as follows.

- "ADSP-BF535 Processor On-Chip Boot ROM" on page 2-4
- "ADSP-BF535 Processor Second-Stage Loader" on page 2-6
- "ADSP-BF535 Processor Boot Streams" on page 2-8
- "ADSP-BF535 Processor Memory Ranges" on page 2-13

ADSP-BF535 Processor On-Chip Boot ROM

The on-chip boot ROM for the ADSP-BF535 processor does the following (Figure 2-2).

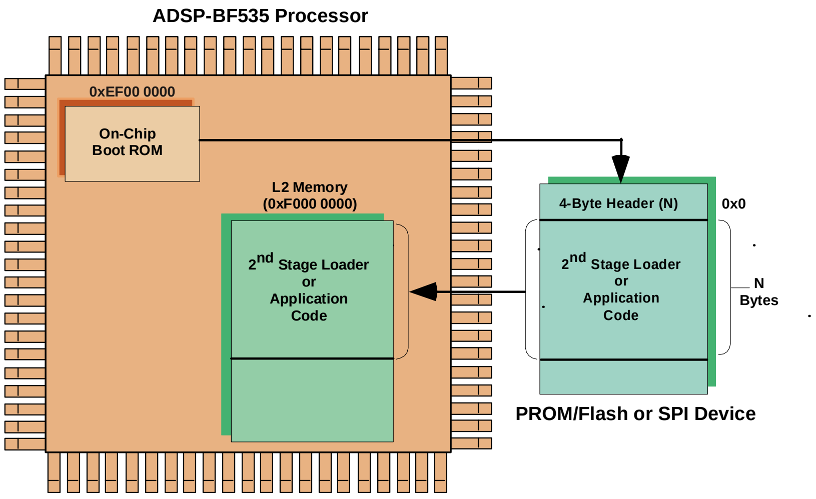


Figure 2-2. ADSP-BF535 Processors: On-Chip Boot ROM

1. Sets up Supervisor mode by exiting the `RESET` interrupt service routine and jumping into the lowest priority interrupt (`IVG15`).

2. Checks whether the `RESET` was a software reset and if so, whether to skip the entire boot sequence and jump to the start of L2 memory (`0xF000 0000`) for execution. The on-chip boot ROM does this by checking bit 4 of the `SYSCR`. If bit 4 is not set, the on-chip boot ROM performs the full boot sequence. If bit 4 is set, the on-chip boot ROM bypasses the full boot sequence and jumps to `0xF000 0000`. The register settings are shown in Figure 2-3.

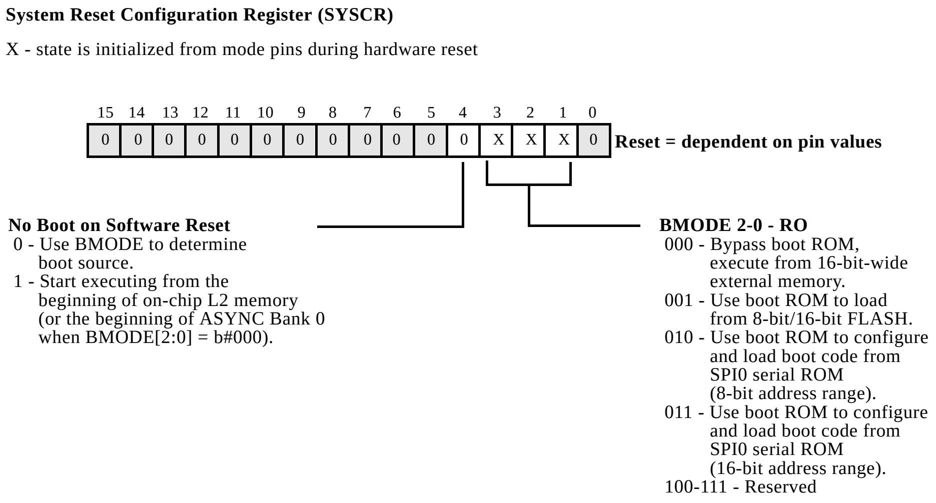

System Reset Configuration Register (SYSCR)
X - state is initialized from mode pins during hardware reset
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 X X X 0
Reset = dependent on pin values
No Boot on Software Reset
0 - Use BMODE to determine boot source.
1 - Start executing from the beginning of on-chip L2 memory (or the beginning of ASYNC Bank 0 when BMODE[2:0] = b#000).
BMODE 2-0 - RO
000 - Bypass boot ROM, execute from 16-bit-wide external memory.
001 - Use boot ROM to load from 8-bit/16-bit FLASH.
010 - Use boot ROM to configure and load boot code from SPI0 serial ROM (8-bit address range).
011 - Use boot ROM to configure and load boot code from SPI0 serial ROM (16-bit address range).
100-111 - Reserved

Figure 2-3. ADSP-BF535 Processors: System Reset Configuration Register

3. Finally, if bit 4 of the `SYSCR` register is not set, the on-chip boot ROM performs the full boot sequence. The full boot sequence includes:

   - Checking the boot source (either Flash/PROM or SPI memory) by reading `BMODE[2:0]` from the `SYSCR` register.
   - Reading the first four bytes from location `0x0` of the external memory device. These four bytes contain the byte count (`N`), which specifies the number of bytes to boot in.
   - Booting in `N` bytes into internal L2 memory starting at location `0xF000 0000`.
   - Jumping to the start of L2 memory for execution.

The on-chip boot ROM boots in `N` bytes from the external memory. These `N` bytes can define the size of the actual application code or a second-stage loader (boot kernel) that boots in the application code.

ADSP-BF535 Processor Second-Stage Loader

The only situation where a second-stage loader is unnecessary is when the application code contains only one section starting at the beginning of L2 memory (`0xF000 0000`).

A second-stage loader must be used in applications in which multiple segments reside in L2 memory or in L1 memory and/or SDRAM. In addition, a second-stage loader must be used to change the wait states or hold time cycles for a Flash/PROM booting or to change the baud rate for a SPI boot (see "Command-Line Switches" on page 2-42 for more information on these features).

When a second-stage loader is used for booting, the following sequence takes place.

1. Upon `RESET`, the on-chip boot ROM downloads `N` bytes (the second-stage loader) from external memory to address `0xF000 0000` in L2 memory (Figure 2-4).

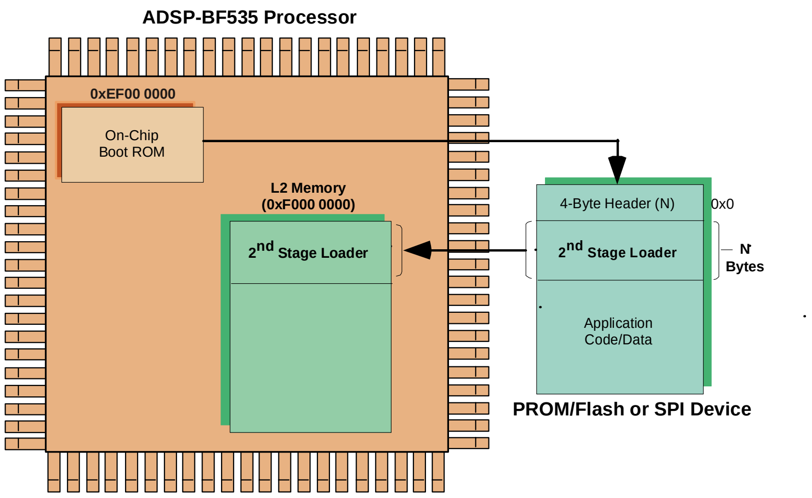


Figure 2-4. ADSP-BF535 Processors: Booting With Second-Stage Loader

2. The second-stage loader copies itself to the bottom of L2 memory.

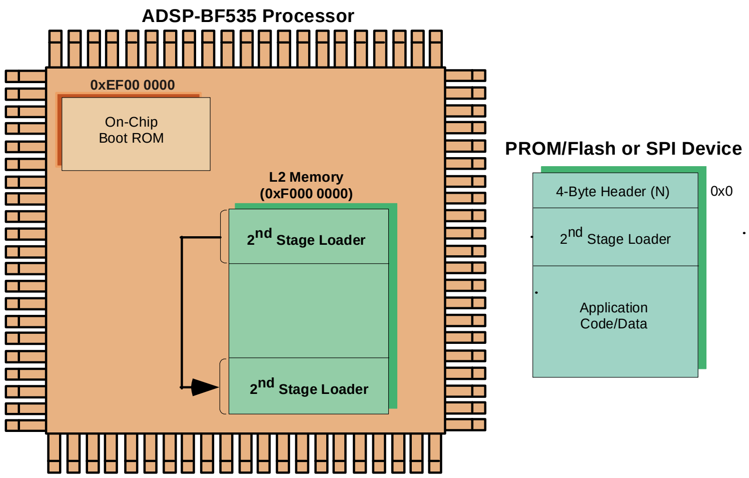


Figure 2-5. ADSP-BF535 Processors: Copying Second-Stage Loader

3. The second-stage loader boots in the application code/data into the various memories of the Blackfin processor.

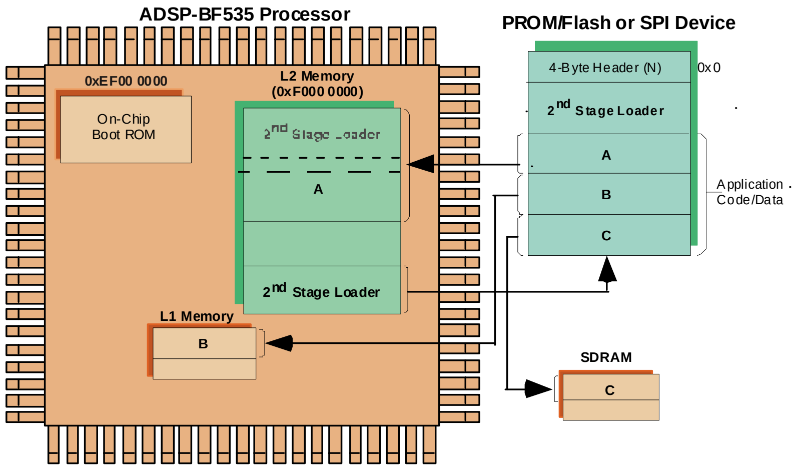


Figure 2-6. ADSP-BF535 Processors: Booting Application Code

4. Finally, after booting, the second-stage loader jumps to the start of L2 memory (`0xF000 0000`) for application code execution (Figure 2-7).

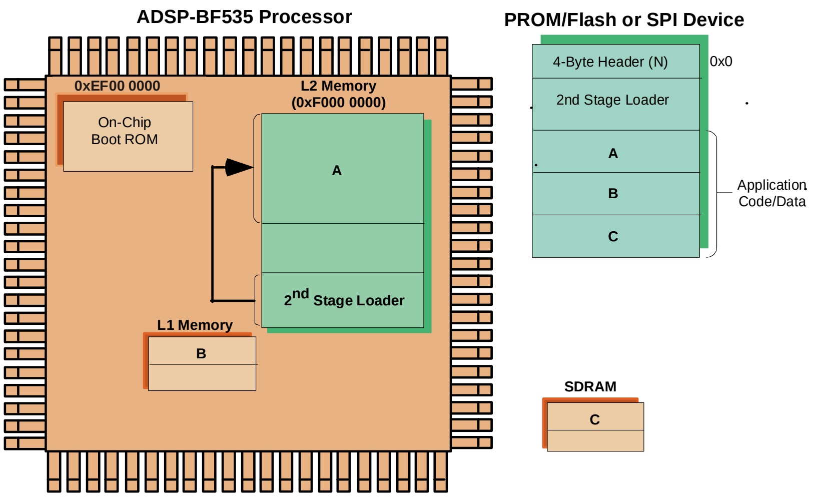


Figure 2-7. ADSP-BF535 Processors: Starting Application Code

ADSP-BF535 Processor Boot Streams

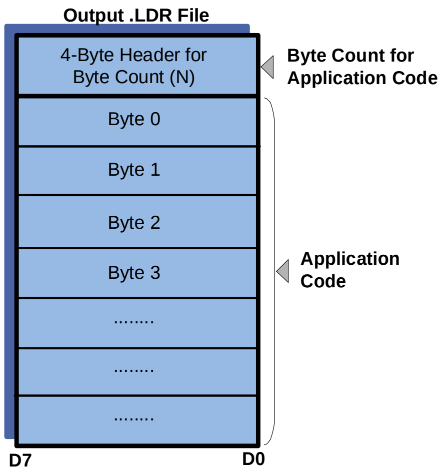
The loader generates the boot stream and places the boot stream in the output loader file (`.LDR`). The loader prepares the boot stream in such a way that the on-chip boot ROM and the second-stage loader can correctly load the application code and data to the processor memory. Therefore, the boot stream contains not only the user application code but also header and flag information that is used by the on-chip boot ROM and the second-stage loader.

Diagrams in this section illustrate boot streams utilized by the ADSP-BF535 processor's boot kernel. The elements are covered as follows.

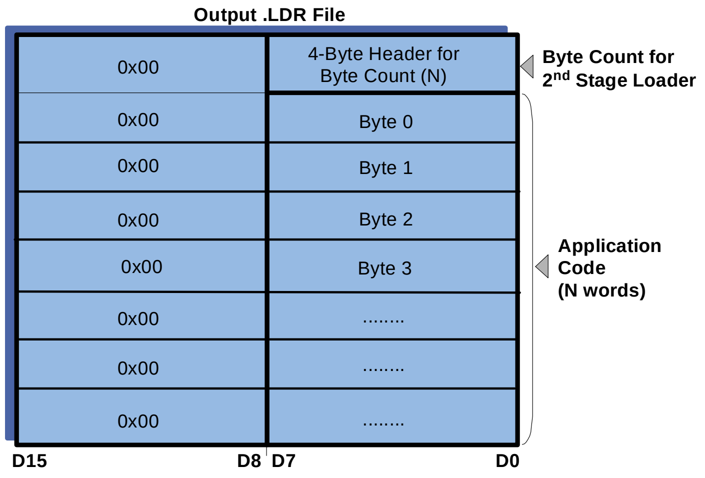
- "Output Loader Files" on page 2-9
- "Global Headers" on page 2-12
- "Block Headers" on page 2-13
- "Flags" on page 2-13

Output Loader Files

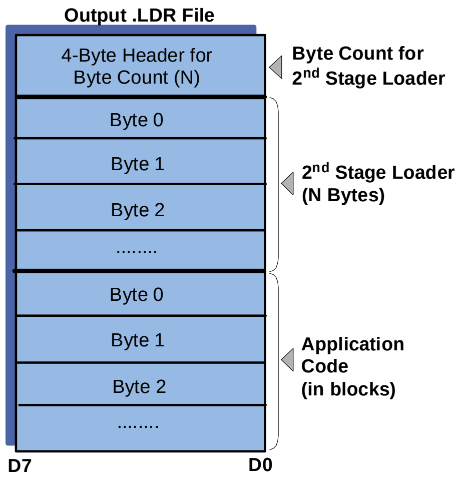
An output loader file for 8-bit PROM/Flash booting and 8-/16-bit addressable SPI booting without the second-stage loader:



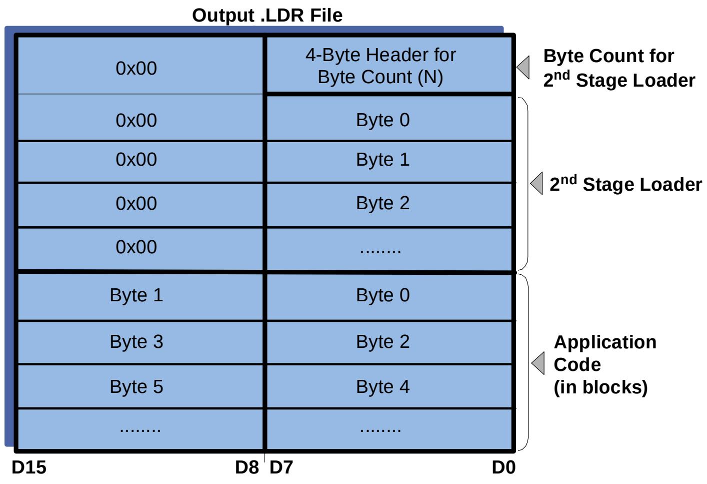
An output loader file for 16-bit PROM/Flash booting without the second-stage loader:



An output loader file for 8-bit PROM/Flash booting and 8- or 16-bit addressable SPI booting with the second-stage loader or kernel:

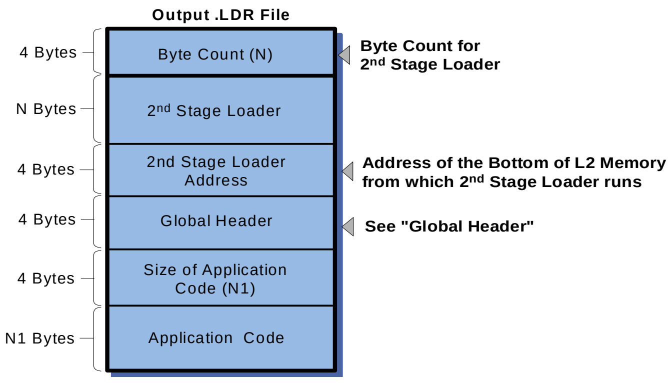

Output .LDR File
4-Byte Header for Byte Count (N)
Byte Count for 2nd Stage Loader
Byte 0
Byte 1
Byte 2
....
2nd Stage Loader (N Bytes)
Byte 0
Byte 1
Byte 2
....
Application Code (in blocks)
D7
D0

An output loader file for 16-bit PROM/FLASH booting with the second-stage loader or kernel:

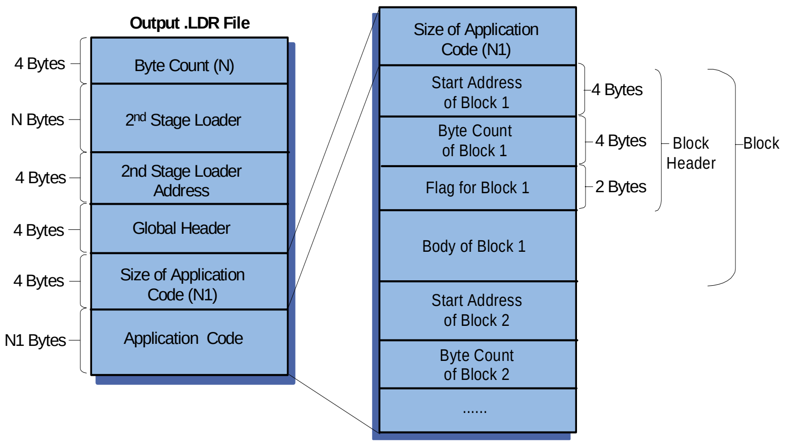


Global Headers and Blocks

Following kernel code and kernel address in a loader file, there is a 4-byte global header. The header provides the global settings for a booting process:

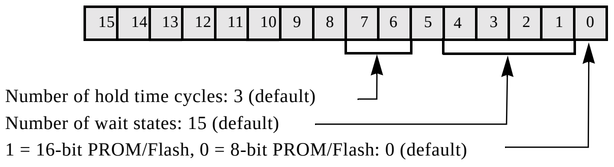


A block is the basic structure of the output `.LDR` file for application code when the second-stage loader is used. All the application code is grouped into blocks. A block always has a block header an a block body if it is a non-zero block. A block does not have a block body if it is a zero block. A block header is illustrated below:

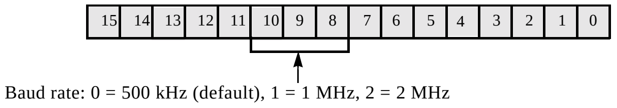


Global Headers

A global header for 8- and 16-bit PROM/Flash booting:


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Number of hold time cycles: 3 (default)
Number of wait states: 15 (default)
1 = 16-bit PROM/Flash, 0 = 8-bit PROM/Flash: 0 (default)

A global header for 8- and 16-bit addressable SPI booting:

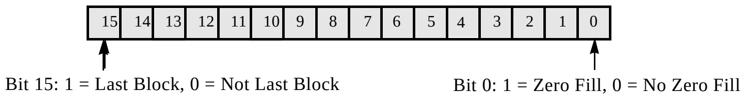


Block Headers

A block header has three words: 4-byte clock start address, 4-byte block byte count, and 2-byte flag word.

Flags

The ADSP-BF535 block flag word's bits are illustrated below.


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Bit 15: 1 = Last Block, 0 = Not Last Block
Bit 0: 1 = Zero Fill, 0 = No Zero Fill

ADSP-BF535 Processor Memory Ranges

Second-stage loaders are available for ADSP-BF535 processors in VisualDSP++ 3.0 and higher. They allow booting to:

- L2 memory (`0xF000 0000`)
- L1 memory
  - Data Bank A SRAM (`0xFF80 0000`)
  - Data Bank B SRAM (`0xFF90 0000`)
  - Instruction SRAM (`0xFFA0 0000`)
  - Scratchpad SRAM (`0xFFB0 0000`)
- SDRAM
  - Bank 0 (`0x0000 0000`)
  - Bank 1 (`0x0800 0000`)

- ✓ Bank 2 (`0x1000 0000`)
- ✓ Bank 3 (`0x1800 0000`)

ⓘ SDRAM must be initialized by user code before any instructions or data are loaded into it.

For more information see “Using Second-Stage Loader” on page 2-49.

Second-Stage Loader Restrictions

When using the second-stage loader:

- The bottom of L2 memory must be reserved during booting. These locations can be reallocated during runtime. The following locations pertain to the current second-stage loaders.
  - ✓ For 8- and 16-bit PROM/Flash booting, reserve `0xF003 FE00-0xF003 FFFF` (last 512 bytes).
  - ✓ For 8- and 16-bit addressable SPI booting, reserve `0xF003 FD00-0xF003 FFFF` (last 768 bytes).
- If segments reside in SDRAM memory, configure the SDRAM registers accordingly in the second-stage loader kernels before booting.
  - ✓ Modify section of code called “`SDRAM setup`” in the second-stage loader and rebuild the second-stage loader.
- Any segments residing in L1 instruction memory (`0xFFA0 0000-0xFFA0 3FFF`) must be 8-byte aligned.
  - ✓ Declare segments, within the `.LDF` file, that reside in L1 instruction memory at starting locations that are 8-byte aligned (for example, `0xFFA0 0000`, `0xFFA0 0008`, `0xFFA0 0010`, and so on).
  - ✓ Or use the `.ALIGN 8;` directives in the application code.

The two reasons for this restriction are:

- Core writes into L1 instruction memory are not allowed.
- DMA from an 8-bit external memory is not possible since the minimum width of the External Bus Interface Unit (EBIU) is 16 bits.

Load bytes into L1 instruction memory by using the instruction test command and data registers, as described in the *Memory* chapter of the appropriate *Hardware Reference* manual. These registers transfer 8-byte sections of data from external memory to internal L1 instruction memory.

ADSP-BF531/BF532/BF533 Processor Booting

Upon reset, an ADSP-BF531/BF532/BF533 processor jumps to the on-chip boot ROM (if `BMODE = 01`, `11`) or jumps to 16-bit external memory for execution (if `BMODE = 00`) located at `0xEF00 0000`. Table 2-2 shows booting modes and execution start addresses for ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors.

Table 2-2. ADSP-BF531/BF532/BF533 Processor Boot Mode Selections

Boot Source	BMODE[1:0]	Execution Start Address	
		ADSP-BF531 ADSP-BF532 Processors	ADSP-BF533 Processor
Execute from 16-bit External ASYNC Bank0 memory (no-boot mode or bypass on-chip boot ROM)	00	0x2000 0000	0x2000 0000
Boot from 8- or 16-bit Prom/Flash	01	0xFFA0 8000	0xFFA0 0000
Reserved	10	0xFFA0 8000	0xFFA0 0000
Boot from a 8-, 16-, or 24-bit addressable SPI memory	11	0xFFA0 8000	0xFFA0 0000

A description of each boot mode is as follows.

- “ADSP-BF531/BF532/BF533 Processor On-Chip Boot ROM” on page 2-17
- “ADSP-BF531/BF532/BF533 Processor Boot Streams” on page 2-19

ADSP-BF531/BF532/BF533 Processor On-Chip Boot ROM

The on-chip boot ROM for ADSP-BF531/BF532/BF533 processors does the following.

1. Sets up supervisor mode by exiting the `RESET` interrupt service routine and jumping into the lowest priority interrupt (`IVG15`).
2. Checks whether the `RESET` was a software reset and if so, whether to skip the entire boot sequence and jump to the start of L1 memory (`0xFFA0 0000` for ADSP-BF533 processor; `0xFFA0 8000` for ADSP-BF531 and ADSP-BF532 processors) for execution. The on-chip boot ROM does this by checking bit 4 of the System Reset Configuration Register (Figure 2-8). If bit 4 is not set, the on-chip boot ROM performs the full boot sequence. If bit 4 is set, the on-chip boot ROM bypasses the full boot sequence and jumps to the start of L1 memory.

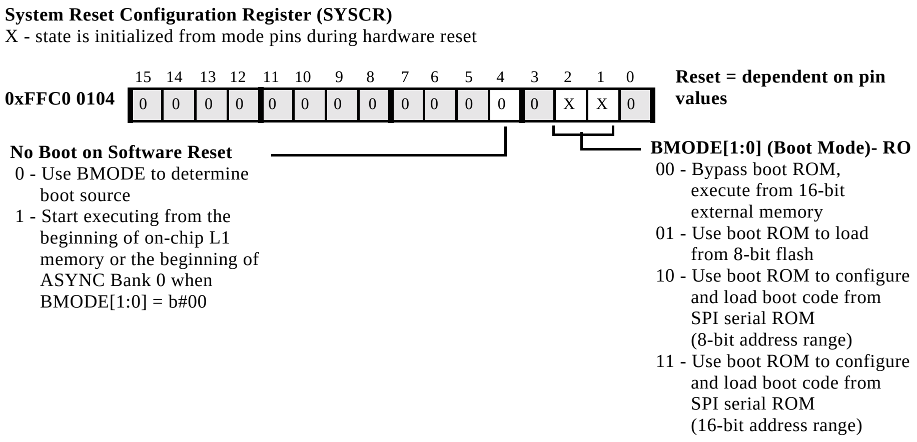

System Reset Configuration Register (SYSCR)
X - state is initialized from mode pins during hardware reset
0xFFC0 0104
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 X X 0
Reset = dependent on pin values
No Boot on Software Reset
0 - Use BMODE to determine boot source
1 - Start executing from the beginning of on-chip L1 memory or the beginning of ASYNC Bank 0 when BMODE[1:0] = b#00
BMODE[1:0] (Boot Mode)- RO
00 - Bypass boot ROM, execute from 16-bit external memory
01 - Use boot ROM to load from 8-bit flash
10 - Use boot ROM to configure and load boot code from SPI serial ROM (8-bit address range)
11 - Use boot ROM to configure and load boot code from SPI serial ROM (16-bit address range)

Figure 2-8. ADSP-BF533 Processors: System Reset Configuration Register

3. Eventually, if bit 4 of the `SYSCR` register is not set, the on-chip boot ROM performs the full boot sequence (Figure 2-9).

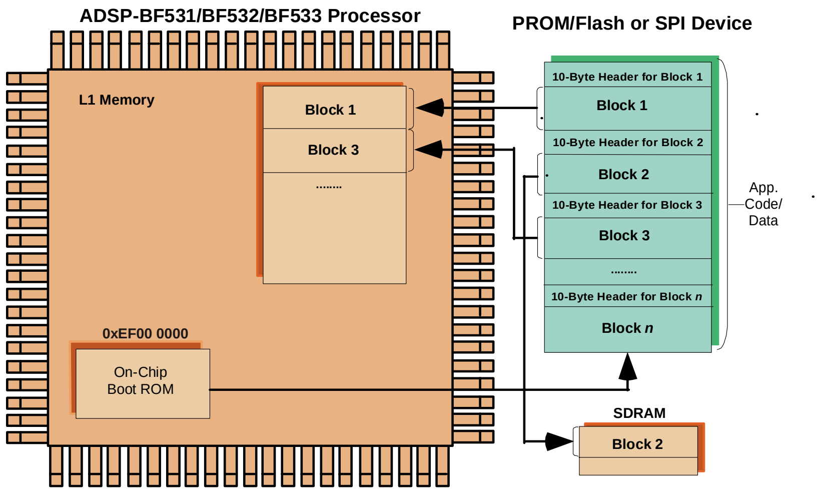


Figure 2-9. ADSP-BF531/BF532/BF533 Processors: Booting Sequence

The booting sequence for ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors is quite different from that of ADSP-BF535 processors. The on-chip boot ROM for the former processors behaves similar to the second-stage loader of ADSP-BF535 processors. The boot ROM has the capability to parse address and count information for each bootable block. This alleviates the need for a second-stage loader for ADSP-BF531/BF532/BF533 processors because a full application can be booted to the various memories with just the on-chip boot ROM.

The loader converts the application code (`.DXE`) into the loadable file by parsing the code and creating a file that consists of different blocks. Each block is encapsulated within a 10-byte header which is illustrated in Figure 2-9 and detailed in the following section. These headers, in turn, are read and parsed by the on-chip boot ROM during booting. The 10-byte header provides all the information the on-chip boot ROM requires: where to boot the block to, how many bytes to boot in, and what to do with the block.

ADSP-BF531/BF532/BF533 Processor Boot Streams

The following sections describe the boot stream, header, and flag framework for ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors.

- "Blocks and Block Headers" on page 2-19
- "Flags of Block Header" on page 2-20
- "Initialization Blocks" on page 2-21

The ADSP-BF531/BF532/BF533 processor boot stream is similar to the boot stream that uses a second-stage kernel of ADSP-BF535 processors (detailed in "ADSP-BF535 Processor Boot Streams" on page 2-8). However, since the former processors do not employ a kernel, their boot streams do not include the kernel code and the associated 4-byte header on the top of the kernel code. There is also no 4-byte global header.

Blocks and Block Headers

As the loader converts the code from an input `.DXE` file into blocks comprising the output loader file, each block is getting preceded by a 10-byte header (Figure 2-10), followed by a block body (if it is a non-zero block) or no block body (if it is a zero block). A description of the header structure can be found in Table 2-3.

Table 2-3. ADSP-BF531/BF532/BF533 Block Header Structure

Bit Field	Description
Address	4-byte address at which the block resides in memory.
Count	4-byte number of bytes to boot.
Flag	2-byte flag containing information about the block "Flags of Block Header" on page 2-20 describes the flag structure.

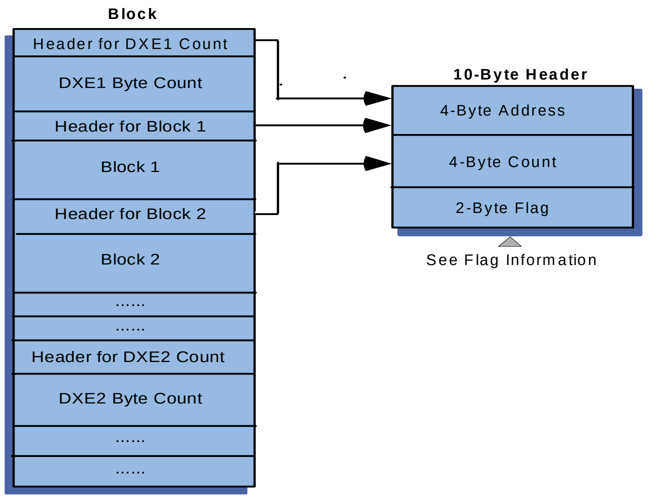


Figure 2-10. ADSP-BF531/BF532/BF533 Processor Boot Stream Structure

Flags of Block Header

Refer to the following figure and Table 2-4 for the flag's bit descriptions.

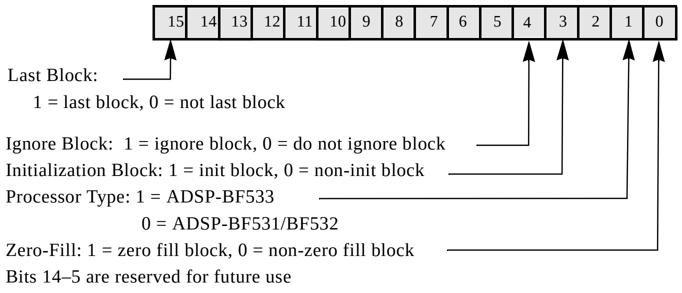

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Last Block:
1 = last block, 0 = not last block
Ignore Block: 1 = ignore block, 0 = do not ignore block
Initialization Block: 1 = init block, 0 = non-init block
Processor Type: 1 = ADSP-BF533
0 = ADSP-BF531/BF532
Zero-Fill: 1 = zero fill block, 0 = non-zero fill block
Bits 14–5 are reserved for future use

Table 2-4. Flag Structure

Bit Field	Description
Zero-Fill Block	Indicates that the block is a buffer filled with zeros. Zero Block is not included within loader file. When the loader parses through the `.DXE` file and encounters a large buffer with zeros, it creates a zero-fill block to reduce `.LDR` file size and boot time. If this bit is set, there is no data in the block.
Ignore Block	Indicates that the block is not to be booted into memory; skips the block and move on to the next one. Currently is not implemented for application code.
Initialization Block	Indicates that the block is to be executed before booting. The initialization block indicator allows the on-chip boot ROM to execute a number of instructions before booting the actual application code. When the on-chip boot ROM detects an Init Block, it boots the block into internal memory and makes a `CALL` to it (Initialization code must have a RTS at the end). This option allows the user to run initialization code (such as SDRAM initialization) before the full boot sequence proceeds. Figure 2-11 and Figure 2-12 illustrate the process. Initialization code can be included within the `.LDR` file by using the `-init` switch (see "-init filename" on page 2-43).
Processor Type	Indicates the processor, either ADSP-BF531/BF532 or ADSP-BF533. After booting is complete, the cn-chip boot ROM jumps to `0xFFA0 0000` for a ADSP-BF533 processor and to `0xFFA0 8000` for a ADSP-BF531/BF532 processor.
Last Block	Indicates that the block is the last block to be booted into memory. After the last block, the processor jumps to the start of L1 memory for application code execution. When it jumps to L1 memory for code execution, the processor is still in Supervisor Mode and in the lowest priority interrupt (IVG15).

Initialization Blocks

The `-init` *`filename`* option directs the loader to produce an initialization block from the code of the initialization section of the named file. The initialization block is placed at the top of a loader file. It is executed before the rest of the code in the loader file is booted into the memory (see Figure 2-11).

Following execution of the initialization block, the booting process continues with the rest of data blocks until it encounters a final block (see Figure 2-12). The initialization code example follows in Listing 2-1

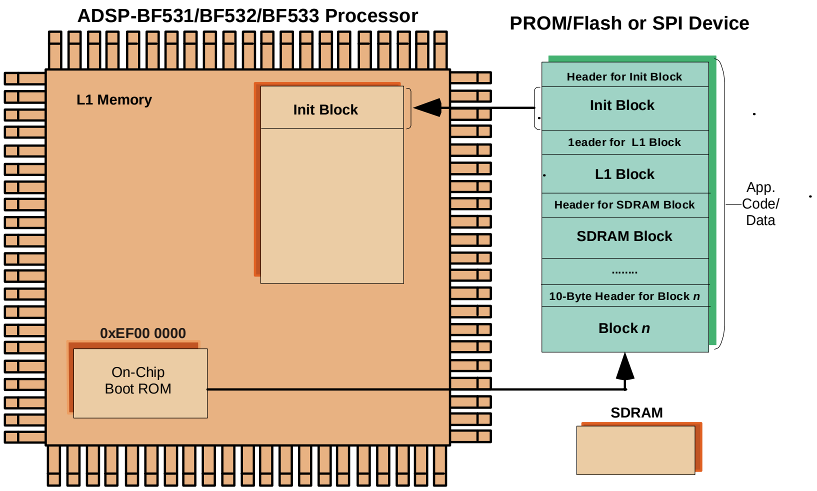


Figure 2-11. ADSP-BF531/BF532/BF533: Initialization Block Execution

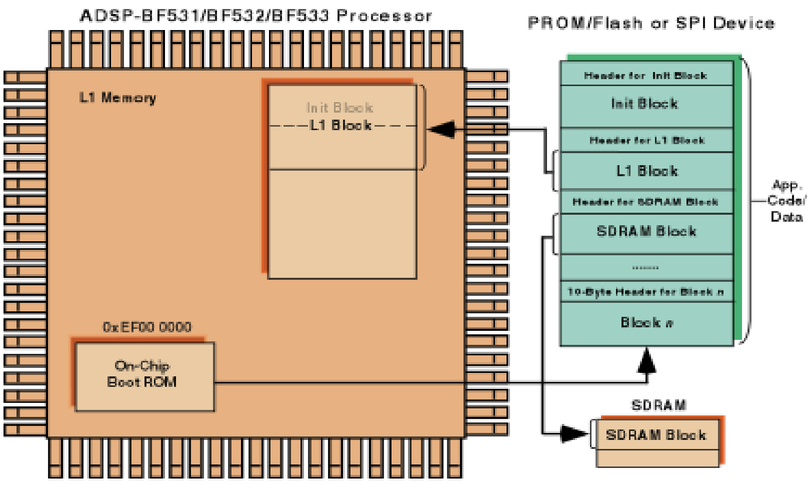


Figure 2-12. ADSP-BF531/BF532/BF533: Booting Application Code

Listing 2-1. Initialization Block Code Example

```
/*  This file contains 3 sections: */
/*  1) A Pre-Init Section-this section saves off all the
       DSP registers onto the stack.
    2) An Init Code Section-this section is the initialization
       code which can be modified by the customer
       As an example, an SDRAM initialization code is supplied.
       The example setups the SDRAM controller as required by
       certain SDRAM types. Different SDRAMs may require
       different initialization procedure or values.
    3) A Post-Init Section-this section restores all the register
       from the stack. Customers should not modify the Pre-Init
       and Post-Init Sections. The Init Code Section can be
       modified for a particular application.*/

#include <defBF532.h>
.SECTION program;
/**********************Pre-Init Section************************/
[--SP] = ASTAT;   /* Stack Pointer (SP) is set to the end of */
[--SP] = RETS;    /* scratchpad memory (0xFFB00FFC) */
[--SP] = (r7:0);  /* by the on-chip boot ROM */
[--SP] = (p5:0);
[--SP] = I0;[--SP] = I1;[--SP] = I2;[--SP] = I3;
[--SP] = B0;[--SP] = B1;[--SP] = B2;[--SP] = B3;
[--SP] = M0;[--SP] = M1;[--SP] = M2;[--SP] = M3;
[--SP] = L0;[--SP] = L1;[--SP] = L2;[--SP] = L3;

/******************Init Code Section*************************/
/*******Please insert Initialization code in this section******/
/***********************SDRAM Setup****************************/
Setup_SDRAM:
     P0.L = EBIU_SDRRC & 0xFFFF;
     /* SDRAM Refresh Rate Control Register */
     P0.H = (EBIU_SDRRC >> 16) & 0xFFFF;
     R0 = 0x074A(Z);
     W[P0] = R0;
     SSYNC;

     P0.L = EBIU_SDBCTL & 0xFFFF;
     /* SDRAM Memory Bank Control Register */
     P0.H = (EBIU_SDBCTL >> 16) & 0xFFFF;
     R0 = 0x0001(Z);
     W[P0] = R0;
     SSYNC;

     P0.L = EBIU_SDGCTL & 0xFFFF;
     /* SDRAM Memory Global Control Register */
     P0.H = (EBIU_SDGCTL >> 16) & 0xFFFF;//
     R0.L = 0x998D;
     R0.H = 0x0091;
     [P0] = R0;
     SSYNC;
/********************Post-Init Section***********************/
L3 = [SP++]; L2 = [SP++]; L1 = [SP++]; L0 = [SP++];
M3 = [SP++]; M2 = [SP++]; M1 = [SP++]; M0 = [SP++];
B3 = [SP++]; B2 = [SP++]; B1 = [SP++]; B0 = [SP++];
I3 = [SP++]; I2 = [SP++]; I1 = [SP++]; I0 = [SP++];
(p5:0) = [SP++];
(r7:0) = [SP++];
RETS = [SP++];
ASTAT = [SP++];
/************************************************************/
RTS;
```

ADSP-BF531/BF532/BF533 Processor Memory Ranges

The on-chip boot ROM on ADSP-BF531, ADSP-BF532, and ADSP-BF533 Blackfin processors allows booting to the following memory ranges.

- L1 memory
  - ADSP-BF531 processor
    - ✓ Data Bank A SRAM (`0xFF80 4000`–`0xFF80 7FFF`)
    - ✓ Instruction SRAM (`0xFFA0 8000`–`FFA0 BFFF`)
  - ADSP-BF532 processor
    - ✓ Data Bank A SRAM (`0xFF80 4000`–`0xFF80 7FFF`)
    - ✓ Data Bank B SRAM (0`xFF90 4000`–`0xFF90 7FFF`)
    - ✓ Instruction SRAM (0`xFFA0 8000`–`FFA1 3FFF`)
  - ADSP-BF533 processor
    - ✓ Data Bank A SRAM (`0xFF80 0000`–`0xFF80 7FFF)`
    - ✓ Data Bank B SRAM (`0xFF90 000`–`0xFF90 7FFF)`
    - ✓ Instruction SRAM (`0xFFA0 0000`–`FFA1 3FFF`)
- SDRAM memory
  - ✓ Bank 0 (`0x0000 0000`–`0x07FF FFFF`)

ⓘ Booting to scratchpad memory (`0xFFB0 0000`) is not supported.

ⓘ SDRAM must be initialized by user code before any instructions or data are loaded into it.

ADSP-BF531/BF532/BF533 Processor SPI Memory Boot Sequence

The ADSP-BF531/BF532/BF533 processors support booting from 8-, 16-, or 24-bit addressable SPI memories (`BMODE = 11`).

To determine the memory type connected to the processor (8-, 16-, or 24-bit), the processor sends signals to the SPI memory until it responds back. The SPI memory does not respond back until it is properly addressed.

The on-chip boot ROM does the following.

1. Sends a `READ` command, `0x03`, then does a dummy `READ`.
2. Sends an address byte, `0x00`, then does a dummy `READ`.
3. Sends another byte, `0x00`, and verifies if the incoming byte is a zero. If the byte is a zero, an 8-bit addressable SPI memory device is connected.
4. If the incoming byte is not a zero, the on-chip boot ROM sends another byte, `0x00`, and verifies if the incoming byte is a zero. If the byte is a zero, a 16-bit addressable SPI memory device is connected.
5. If the incoming byte is not a zero, the on-chip boot ROM sends another byte, `0x00`, and verifies if the incoming byte is a zero. The last byte is a zero when a 24-bit addressable SPI memory device is connected.

The `MISO` line must be pulled high for `BMODE = 11`. Since the `MISO` line is pulled up high, the processor receives one of the following.

- A `0xFF` if the part is not responding back with valid data
- A `0x00` if the part is responding back with valid data.

The boot uses Slave Select 2 that maps to `PF2`. The on-chip boot ROM sets the `Baud Rate` register to "`133`", which, based on a 133 MHz system clock, results in a 133 MHz/(2*133) = 500 kHz baud rate.

Analog Devices recommends the following SPI memory devices.

- 8-bit addressable SPI memory: 25LC040 from Microchip (`http://www.microchip.com/download/lit/pline/memory/spi/21204c.pdf`)
- 16-bit addressable SPI memory: 25CL640 from Microchip (`http://www.microchip.com/download/lit/pline/memory/spi/21223e.pdf`)
- 24-bit addressable SPI memory: M25P80 from STMicroelectronics (`http://www.st.com/stonline/books/pdf/docs/8495.pdf`)

ADSP-BF561 Processor Booting

The booting sequence for the ADSP-BF561 dual-core processor is similar to the ADSP-BF531/BF532/BF533 processor booting sequence (described on page 2-16). Differences occur because the ADSP-BF561 processor has two cores: core A and core B. After reset, core B remains idle, but core A executes the on-chip boot ROM located at address `0xEF00 0000`.

Please refer to Chapter 3 of the *ADSP-BF561 Hardware Reference Manual* for information about the processor's operating modes and states. Please refer to “System Reset and Power up Configuration” for background information on reset and booting.

The boot ROM loads an application program from an external memory device and starts executing that program by jumping to the start of core A's L1 instruction SRAM, at address `0xFFA0 0000`.

Table 2-5 summarizes the boot modes and execution start addresses for ADSP-BF561 processors.

Table 2-5. ADSP-BF561 Processor Boot Mode Selections

Boot Source	BMODE [2:0]	Execution Start Address
Reserved	`000`	Not applicable
Boot from 8-bit/16-bit PROM/Flash memory	`001`	`0xFFA0 0000`
Boot from 8-bit addressable SPI0 serial EEPROM	`010`	`0xFFA0 0000`
Boot from 16-bit addressable SPI0 serial EEPROM	`011`	`0xFFA0 0000`
Reserved	`111-100`	Not applicable

Just like the ADSP-BF531/BF532/BF533 processor, the ADSP-BF561 boot ROM uses the interrupt vectors to stay in supervisor mode. The boot ROM code transitions from the `RESET` interrupt service routine into the lowest priority user interrupt service routine (`Int 15`) and remains in the

ISR. The boot ROM then checks to see if it has been invoked by a software reset by examining bit 4 of the System Reset Configuration Register (`SYSCR`).

If bit 4 is not set, the boot ROM presumes that a hard reset has occurred and performs the full boot sequence. If bit 4 is set, the boot ROM understands that the user code has invoked a software reset and restarts the user program by jumping to the beginning of core A's L1 memory (`0xFFA0 0000`), bypassing the entire boot sequence.

When developing an ADSP-BF561 processor application, you start with compiling and linking your application code into an executable file (`.DXE`). The debugger loads the `.DXE` into the processor's memory and executes it. With two cores, two `.DXE` files can be loaded at once. In the real-time environment, there is no debugger, which allows the boot ROM to load the executables into memory.

ADSP-BF561 Processor Boot Streams

The loader converts the `.DXE` into a boot stream file (`.LDR`) by parsing the executable and creating blocks. Each block is encapsulated within a 10-byte header. The `.LDR` file is burned into the external memory device (Flash, PROM, or EEPROM). The boot ROM reads the external memory device, parsing the headers and copying the blocks to the addresses where they reside during program execution. After all the blocks are loaded, the boot ROM jumps to address `0xFFA0 0000` to execute the core A program.

ⓘ When running code on both cores, the core A program is responsible for releasing core B from the idle state by clearing bit 5 in core A's System Configuration Register. Then core B begins execution at address `0xFF60 0000`.

Multiple `.DXE` files are often combined into a single boot stream.

Unlike the ADSP-BF531/BF532/BF533 processor, the ADSP-BF561 boot stream begins with a 4-byte global header, which contains information about the external memory device. The global header also contains a signature in the upper 4 bits that prevents the boot ROM from trying to read a boot stream from a blank device.

Table 2-6. ADSP-BF561 Global Header Structure

Bit Field	Description
`0`	`1` = 16-bit Flash, `0` = 8-bit Flash; default is `0`
`1-4`	Number of wait states; default is `15`
`5`	Unused bit
`6-7`	Number of hold time cycles for Flash; default is 3
`8-10`	Baud rate for SPI boot: `00` = 500k, `01` = 1M, `10` = 2M.
`11-27`	Reserved for future use
`28-31`	Signature that indicates valid boot stream

Following the global header is a `.DXE` count block, which contains a 32-bit byte count for the first `.DXE` in the boot stream. Though this block contains only a byte count, it is encapsulated by a 10-byte block header, just like the other blocks.

The 10-byte header tells the boot ROM where in memory to place each block, how many bytes to copy, and whether the block needs any special processing. The header structure is the same as that of the ADSP-BF531/BF532/BF533 processors (described in “Blocks and Block Headers” on page 2-19). Each header contains a 4-byte start address for the data block, a 4-byte count for the data block, and a 2-byte flag word, indicating whether the data block is a “zero-fill” block or a “final block” (the last block in the boot stream).

For the .DXE count block, the address field is irrelevant since the block is not going to be copied to memory. The "ignore bit" is set in the flag word of this header, so the boot loader does not try to load the .DXE count but skips the count. For more details, see "Flags of Block Header" on page 2-20

Following the .DXE count block are the rest of the blocks of the first .DXE.

A bit-by-bit description of the boot steam is presented in Table 2-7. When learning about the ADSPP-BF561 boot stream structure, keep in mind that the count byte for each .DXE is, itself, a block encapsulated by a block header.

Table 2-7. ADSP-BF561 Processor Boot Stream Structure

Bit Field	Description
0–7	LSB of the Global Header
8–15	8-15 of the Global Header
16–23	16-23 of the Global Header
24–31	MSB of the Global Header
32–39	LSB of the address field of 1st DXE count block (no care)
40–47	8-15 of the address field of 1st DXE count block (no care)
48–55	16-23 of the address field of 1st DXE count block (no care)
56–63	MSB of the address field of 1st DXE count block (no care)
64–71	LSB (4) of the byte count field of 1st DXE count block
72–79	8-15 (0) of the byte count field of 1st DXE count block
80–87	16-23 (0) of the byte count field of 1st DXE count block
88–95	MSB (0) of the byte count field of 1st DXE count block
96–103	LSB of the flag word of 1st DXE count block – ignore bit set
104–111	MSB of the flag word of 1st DXE count block
112-119	LSB of the first 1st .DXE byte count

Table 2-7. ADSP-BF561 Processor Boot Stream Structure (Cont'd)

Bit Field	Description
120-127	8-15 of the first 1st .DXE byte count
128-135	16-23 of the first 1st .DXE byte count
136-143	24-31 of the first 1st .DXE byte count
144-151	LSB of the address field of the 1st data block in 1st .DXE
152-159	8-15 of the address field of the 1st data block in 1st .DXE
160-167	16-23 of the address field of the 1st data block in 1st .DXE
168-175	MSB of the address field of the 1st data block in 1st .DXE
176-183	LSB of the byte count of the 1st data block in 1st .DXE
184-191	8-15 of the byte count of the 1st data block in 1st .DXE
192-199	16-23 of the byte count of the 1st data block in 1st .DXE
200-207	MSB of the byte count of the 1st data block in 1st .DXE
208-215	LSB of the flag word of the 1st block in 1st .DXE
216-223	MSB of the flag word of the 1st block in 1st .DXE
224-231	Byte 3 of the 1st block of 1st .DXE
232-239	Byte 2 of the 1st block of 1st .DXE
240-247	Byte 1 of the 1st block of 1st .DXE
248-255	Byte 0 of the 1st block of 1st .DXE
256-263	Byte 7 of the 1st block of 1st .DXE
…	And so on …
…	LSB of the address field of the nth data block of 1st .DXE
…	8-15 of the address field of the nth data block of 1st .DXE
…	16-23 of the address field of the nth data block of 1st .DXE
…	MSB of the address field of the nth data block of 1st .DXE
…	LSB of the byte count field of the nth block of 1st .DXE

Table 2-7. ADSP-BF561 Processor Boot Stream Structure (Cont'd)

Bit Field	Description
…	8-15 of the byte count field of the nth block of 1st .DXE
…	16-23 of the byte count field of the nth block of 1st .DXE
…	MSB of the byte count field of the nth block of 1st .DXE
…	LSB of the flag word of the nth block of 1st .DXE
…	MSB of the flag word of the nth block of 1st .DXE
…	…
…	Byte 1 of the nth block of 1st .DXE
…	Byte 0 of the nth block of 1st .DXE
	LSB of the address field of 2nd .DXE count block (no care)
	8-15 of the address field of 2nd .DXE count block (no care)
	And so on…

ADSP-BF561 Processor Memory Ranges

The on-chip boot ROM of the ADSP-BF561 processor can load a full application to the various memories of both cores. Booting is allowed to the following memory ranges. The boot ROM clears these memory ranges before booting in a new application.

- Core A
  - ✓ L1 Instruction SRAM (`0xFFA0 0000`–`0xFFA0 3FFF`)
  - ✓ L1 Instruction Cache/SRAM (`0xFFA1 0000`–`0xFFA1 3FFF`)
  - ✓ L1 Data Bank A SRAM (`0xFF80 0000`–`0xFF80 3FFF`)
  - ✓ L1 Data Bank A Cache/SRAM (`0xFF80 4000`–`0xFF80 7FFF`)
  - ✓ L1 Data Bank B SRAM (`0xFF90 0000`–`0xFF90 3FFF`)
  - ✓ L1 Data Bank B Cache/SRAM (`0xFF90 4000`–`0xFF90 7FFF`)
- Core B
  - ✓ L1 Instruction SRAM (`0xFF60 0000`–`0xFF6 03FFF`)
  - ✓ L1 Instruction Cache/SRAM (`0xFF61 0000`–`0xFF61 3FFF`)
  - ✓ L1 Data Bank A SRAM (`0xFF40 0000`–`0xFF40 3FFF`)
  - ✓ L1 Data Bank A Cache/SRAM (`0xFF40 4000`–`0xFF40 7FFF`)
  - ✓ L1 Data Bank B SRAM (`0xFF50 0000`–`0xFF50 3FFF`)
  - ✓ L1 Data Bank B Cache/SRAM (`0xFF50 4000`–`0xFF50 7FFF`)
- Four Banks of Configurable Synchronous DRAM (`0x0000 0000`–(up to) `0x1FFF FFFF`)

The boot ROM does not support booting to core A scratch memory (`0xFFB0 0000`–`0xFFB0 0FFF`) and to core B scratch memory (`0xFF70 0000`–`0xFF70 0FFF`). Data that needs to be initialized prior to runtime should not be placed in scratch memory.

ADSP-BF561 Processor Initialization Blocks

An initialization block or a second-stage loader must be used to initialize the SDRAM memory of the ADSP-BF561 processor before any instructions or data are loaded into it.

The initialization block is identified by a bit in the flag word of the 10-byte block header. When the boot ROM encounters an initialization block in the boot stream, it loads the block and executes it immediately. The initialization block must save and restore registers and return to the boot ROM, so the boot ROM can load the rest of the blocks. For more details, see "Flags of Block Header" on page 2-20.

Both the initialization block and second stage loader can be used to force the boot ROM to load a specific `.DXE` from the external memory device if the boot ROM stores multiple executable files. The initialization block can manipulate the `R0` or `R3` register, which the boot ROM uses as external memory pointers for Flash/PROM or SPI memory boot, respectively.

After the processor returns from the initialization block, the boot ROM continues to load blocks from the location specified in the `R0` or `R3` register, which can be any `.DXE` in the boot stream. This option requires the starting locations of specific executables within external memory. The `R0` or `R3` register must point to the 10-byte count header, as illustrated in "ADSP-BF531/BF532/BF533 and ADSP-BF561 Multiple .DXE Booting" on page 2-37.

ADSP-BF561 Multiple .DXE Booting

A typical dual-core application is separated into two executable files; one for each core. The default linker description files (LDFs) for the ADSP-BF561 processor creates two separate executable files (`p0.dxe` and `p1.dxe`) and some shared memory files (`sml2.sm` and `sml3.sm`). By modifying the LDF, it is possible to create a dual-core application that combines both cores into a single `.DXE` file. This is not recommended unless the application is a simple assembly language program which does not link any C runtime libraries. When using shared memory and/or C runtime routines on both cores, it is best to generate a separate `.DXE` file for each core. The loader combines the contents of the shared memory files (`sml2.sm`, `sml3.sm`) into the `.DXE` file for core A (`p0.dxe`).

The boot ROM only loads one single executable before the ROM jumps to the start of core A instruction SRAM (`0xFFA0 0000`). When two `.DXEs` must be loaded, a second stage loader should be used. The second stage boot loader must start at `0xFFA0 0000`. The boot ROM loads and executes the second stage loader. A default second stage loader is provided for each boot mode and can be customized by the user.

Unlike the initialization block, the second-stage loader takes full control over the boot process and never returns to the boot ROM.

The second-stage loader can use the `.DXE` byte count blocks to find specific `.DXE` s in external memory.

The default second stage loader uses the last 1024 bytes of L2 memory. The area must be reserved during booting but can be reallocated at runtime.

ADSP-BF531/BF532/BF533 and ADSP-BF561 Multiple .DXE Booting

This section describes how to boot more than one `.DXE` file into a ADSP-BF531/BF532/BF533 and ADSP-BF561 processor. The information presented in this section applies to all of the named processors. For additional information on the ADSP-BF561 processor, refer to "ADSP-BF561 Multiple .DXE Booting" on page 2-36.

The ADSP-BF531/BF532/BF533 and ADSP-BF561 loader file structure and the silicon revision of 0.1 and higher allow to boot multiple `.DXE` files into a single processor from external memory. Each executable file is preceded by a 4-byte count header, which is the number of bytes within the executable, including headers. This information can be used to boot a specific `.DXE` into the processor. The 4-byte `.DXE` count block is encapsulated within a 10-byte header to be compatible with the silicon revision 0.0. For more information, see "Blocks and Block Headers" on page 2-19.

Booting multiple executables can be accomplished by one of the following methods.

1. Use the second-stage loader switch, "-l userkernel". This option allows to use your own second-stage loader or kernel.

   After the second-stage loader gets booted into internal memory via the on-chip boot ROM, it has full control over the boot process. Now the second-stage loader can use the `.DXE` byte counts to boot in one or more `.DXE`s from external memory.

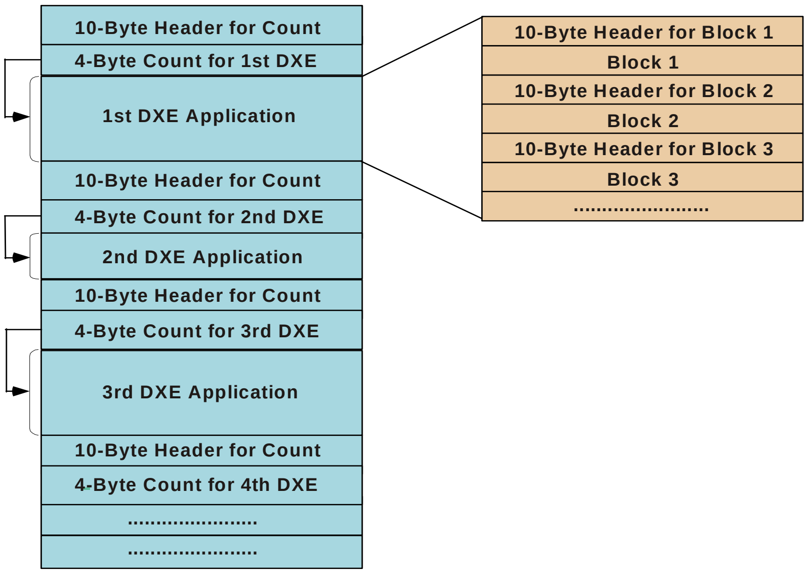


Figure 2-13. ADSP-BF531/BF32/BF33/BF561: Multi-Application Booting

2. Use the initialization block switch, "-init filename", where "filename" is the name of the executable file containing the init code. This option allows you to change the external memory pointer and boot a specific `.DXE` via the on-chip boot ROM.

   A sample initialization code is included in Listing 2-2. The `R0` and `R3` registers are used as external memory pointers by the on-chip boot ROM. The `R0` register is for Flash/PROM boot, and `R3` is for SPI memory boot. Within the initialization block code, change the value of `R0` or `R3` to point to the external memory location at which the specific application code starts. After the processor returns from the initialization block code to the on-chip boot ROM, the on-chip boot ROM continues to boot in bytes from the location specified in the `R0` or `R3` register.

Listing 2-2. Initialization Block Code Example for Multiple .DXE Boot

```
#include <defBF532.h>
.SECTION program;
/*******Pre-Init Section***************************************/
     [--SP] = ASTAT;
     [--SP] = RETS;
     [--SP] = (r7:0);
     [--SP] = (p5:0);
     [--SP] = I0;[--SP] = I1;[--SP] = I2;[--SP] = I3;
     [--SP] = B0;[--SP] = B1;[--SP] = B2;[--SP] = B3;
     [--SP] = M0;[--SP] = M1;[--SP] = M2;[--SP] = M3;
     [--SP] = L0;[--SP] = L1;[--SP] = L2;[--SP] = L3;
/**************************************************************/
/*******Init Code Section**************************************
R0.H = High Address of DXE Location (R0 for Flash/Prom Boot,
                                     R3 for SPI boot)
R0.L = Low Address of DXE Location. (R0 for Flash/Prom Boot,
                                     R3 for SPI boot)
**************************************************************/
/*******Post-Init Section**************************************/
     L3 = [SP++]; L2 = [SP++]; L1 = [SP++]; L0 = [SP++];
     M3 = [SP++]; M2 = [SP++]; M1 = [SP++]; M0 = [SP++];
     B3 = [SP++]; B2 = [SP++]; B1 = [SP++]; B0 = [SP++];
     I3 = [SP++]; I2 = [SP++]; I1 = [SP++]; I0 = [SP++];
     (p5:0) = [SP++];
     /* MAKE SURE NOT TO RESTORE
     R0 for Flash/Prom Boot, R3 for SPI Boot */
     (r7:0) = [SP++];
     RETS = [SP++];
     ASTAT = [SP++];
/**************************************************************/
     RTS;
```

Blackfin Processor Loader Guide

Loader operations depend on the loader options, which control how the loader processes executable files, letting you select features such as boot mode, boot kernel, and output file format. These options are specified on the loader's command line or via the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment. The **Load** page consists of multiple panes and is the same for all Blackfin processors. When you open the **Load** page, the default loader settings for the selected processor are already set.

ⓘ Option settings on the **Load** page correspond to switches displayed on the command line.

These sections describe how to produce a bootable or non-bootable loader file (`.LDR`):

- "Using Loader Command Line" on page 2-40
- "Using Base Loader" on page 2-47
- "Using Second-Stage Loader" on page 2-49
- "Using ROM Splitter" on page 2-51

Using Loader Command Line

The Blackfin loader uses the following command-line syntax.

For a single input file:

`elfloader` *`sourcefile`* `-proc` *`processor`* [`-`*`switch`* …]

For multiple input files:

`elfloader` *`sourcefile1 sourcefile2`* … `-proc` *`processor`* [`-`*`switch`* …]

where:

- `sourcefile`—Name of the executable file (`.DXE`) to be processed into a single boot-loadable or non-bootable file. An input file name can include the drive and directory. For multiprocessor or multiinput systems, specify multiple input `.DXE`s. Put the input filenames in the order in which you want the loader to process the files. Enclose long file names within straight-quotes, “`long file name`”.

- `-proc processor`—Part number of the processor (for example, `ADSP-BF531`) for which the loadable file is to be built. Provide a processor part number for every input `.DXE` if designing multiprocessor systems. If the processor is not specified, the default is `ADSP-BF535`.

- `-switch` ...—One or more optional switches to process. Switches select operations and modes for the loader.

Command-line switches may be placed on the command line in any order, except the order of input files for a multiinput system. For a multiinput system, the loader processes the input files in the order presented on the command line.

File Searches

File searches are important in the loader processing. The loader supports relative and absolute directory names, default directories. File searches occur as described on page 1-9.

File Extensions

Some loader switches take a file name as an optional parameter. Table 2-8 lists the expected file types, names, and extensions.

Table 2-8. File Extensions

Extension	File Description
`.DXE`	Loader input files, boot-kernel files, and initialization files.
`.LDR`	Loader output file.
`.KNL`	Loader output files containing kernel code only when two output files are selected.

Command-Line Switches

A summary of the loader command-line switches appear in Table 2-9.

Table 2-9. Blackfin Loader Command-Line Switches

Switch	Description

`-b prom`
`-b flash`
`-b spi`
	Specifies the boot mode. The `-b` switch directs the loader to prepare a boot-loadable file for the specified boot mode. Valid boot modes include PROM, Flash, and SPI.
	If `-b` does not appear on the command line, the default is `-b prom`.

`-baudrate #`	Accepts a baud rate for SPI booting only. Note: Currently supported only for ADSP-BF535 processors. Valid baud rates and corresponding values (#) are: • `500K`—500 kHz, the default • `1M`—1 MHz • `2M`—2 MHz Boot kernel loading supports an SPI baud rate up to 2 MHz.
`-enc` *`dll_filename`*	Encrypts the data stream from the application `.DXE` files. If the filename parameter does not appear on the command line, the encryption algorithm from the default ADI's file is used.
`-kenc` *`dll_filename`*	Specifies the user encryption file for the data stream from the kernel file. If the filename parameter does not appear on the command line, the encryption algorithm from the default ADI's file is used.

`-f hex`
`-f ASCII`
`-f binary`
	Specifies the boot file's format. The `-f` switch prepares a boot-loadable file in the specified format (Intel hex 32, ASCII, binary)
	If the `-f` switch does not appear on the command line, the default boot-mode format is `hex` for Flash/PROM, and `ASCII` for SPI.

Table 2-9. Blackfin Loader Command-Line Switches (Cont'd)

Switch	Description

`-h`
or
`-help`
	Invokes the command-line help, outputs a list of command-line switches to standard output, and exits. By default, the `-h` switch alone provides help for the loader driver. To obtain a help screen for your target Blackfin processor, add the `-proc` switch to the command line. For example: type `elfloader -proc ADSP-BF535 -h` to obtain help for the ADSP-BF535 processor.

`-ghc #`	Specifies a 4-bit value (global header cookie) for bits 31–28 of the global header.
`-HoldTime #`	Allows the loader to specify a number of hold-time cycles for PROM/Flash boot. The valid values (#) are from `0` through `3`. The default value is `3`. **Note:** Currently supported only for ADSP-BF535 processors.
`-init` *`filename`*	Directs the loader to include the initialization block from the named file. The loader places the code from the initialization section of the specified `.DXE` file in the boot stream. The kernel loads the block and then calls it. It is the responsibility of the code within the block to save/restore state/registers and then perform a RTS back to the kernel. **Note:** This switch cannot be applied to ADSP-BF535 processors.

`-kb prom`
`-kb flash`
`-kb spi`
	Specifies the boot mode (PROM, Flash, or SPI) for the boot kernel output file if you select to generate two output files from the loader: one for the boot kernel and another for the user application code. This switch must be used in conjunction with the `-o2` switch.
	If the `-kb` switch is absent on a command line, the loader generates the file for the boot kernel in the same boot mode as used to output the user application code file.

`-kf hex`
`-kf ascii`
`-kf binary`
	Specifies the output file format (hex, ASCII, or binary) for the boot kernel if you select to output two files from the loader: one for the boot kernel and another for user application code.
	This switch must be used in conjunction with the `-o2` switch. If the `-kf` switch is absent from the command line, the loader generates the file for the boot kernel in the same format as for the user application program.

Table 2-9. Blackfin Loader Command-Line Switches (Cont'd)

Switch	Description
`-kp #`	Specifies a hex PROM/Flash output start address for kernel code. A valid value is between [`0x0`, `0xFFFFFFFF`]. The specified value will not be used if no kernel or/and initialization code is included in the loader file.
`-k`*`Width #`*	Specifies the width of the boot kernel output file when there are two output files: one for boot kernel and one for user application code. Valid values are: • `8` or `16` for PROM or Flash boot kernel • `8` for SPI boot kernel If this switch is absent from the command line, the default file width is: • the *`-width`* parameter when booting from PROM/Flash • `8` when booting from SPI. This switch should be used in conjunction with the `-o2` switch.
`-l` *`userkernel`*	Specifies the user's boot kernel. The loader utilizes the user-specified kernel and ignores the default boot kernel if there is one. Note: Currently, only ADSP-BF535 processors have default kernels.
`-M`	Generates make dependencies only, no output file will be generated.
`-maskaddr #`	Masks all EPROM address bits above or equal to `#`. For example, `-maskaddr 29` (default) masks all the bits above and including `A29` (ANDed by `0x1FFF FFFF`). For example, `0x2000 0000` becomes `0x0000 0000`. The valid `#`s are integers `0` through `32`, but based on your specific input file, the value can be within a subset of `[0,32]`. This switch requires `-romsplitter` and affects the ROM section address only.
`-MaxBlockSize #`	Specifies the maximum block byte count, which must be a multiply of 16.
`-MM`	Generates make dependencies while producing the output files.
`-Mo` *`filename`*	Writes make dependencies to the named file. The `-Mo` option is for use with either the `-M` or -`MM` option. If `-Mo` is not present, the default is `a <stdout>` display.

Table 2-9. Blackfin Loader Command-Line Switches (Cont'd)

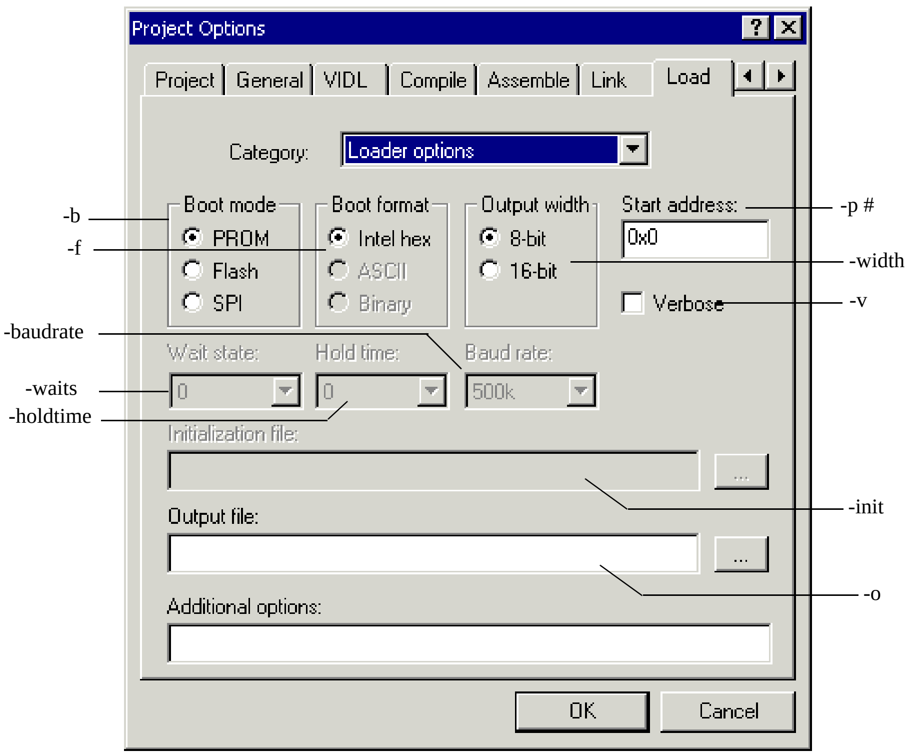
Switch	Description
`-Mt` *`filename`*	Specifies the make dependencies target output filename. The `-Mt` option is for use with either the `-M` or -`MM` option. If `-Mt` is not present, the default is the name of the input file with the `.LDR` extension.
`-no2kernel`	Produces the output file without the boot kernel but uses the boot-strap code from the internal boot ROM. The boot stream generated by the loader is different from the one generated by the boot kernel. Note: Currently supported only for ADSP-BF535 processors.
`-o` *`filename`*	Directs the loader to use the specified *`filename`* as the name for the loader's output file. If the *`filename`* is absent, the default name is the name of the input file with an `.LDR` extension.
`-o2`	Produces two output files: one for the Init block (if present) and boot kernel and another for the user application code. To have a different format from the application code output file, use the `-kb -kf -kwidth` switches to specify the boot mode, the boot format, and the boot width for the output kernel file. If you intent to use the `-O2` switch, do not combine it with: • `-nokernel` on ADSP-BF535 processors • `-l` *`filename`* and/or `-init` *`filename`* on ADSP-BF531/BF532/BF535/BF561 processors.
`-p #`	Specifies a hex PROM/Flash output start address for the application code. A valid value is between [`0x0`, `0xFFFFFFFF`]. A specified value must be greater than that specified by `-kp` if both kernel and/or initialization and application code are in the same output file (do not use `-o2`).
`-proc` *`processor`*	Specifies the target processor. The *`processor`* can be one of the following: `ADSP-BF531`, `ADSP-BF532`, `ADSP-BF533`, `ADSP-BF535`, and `ADSP-BF561`.
`-romsplitter`	Creates a non-bootable image only. This switch overwrites the `-b` switch and any other switch bounded by the boot modes. Note: In the `.LDF` file, declare memory segments to be 'splitted' as type "`ROM`". The splitter skips "`RAM`" segments, resulting in an empty file if all segments are declared as "`RAM`". The `-romsplitter` switch supports hex and ASCII formats.

Table 2-9. Blackfin Loader Command-Line Switches (Cont'd)

Switch	Description
`-ShowEncryptionMessage`	Displays a message returned from the encryption function.
`-si-revision` *`version`*	Provides a silicon revision of the specified processor. The *`version`* parameter represents a silicon revision of the processor specified by the `-proc` switch. The revision version takes one of two forms: • One or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Version `0.1` is distinct from and "lower" than version `0.10`. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • A `none` version value is also supported, indicating that the VDSP++ tool should ignore silicon errata. This switch either generates a warning about any potential anomalous conditions or generates an error if any anomalous conditions occur. Note: In the absence of the silicon revision switch, the loader selects the greatest silicon revision it is aware of, if any. Note: In the absence of the *`version`* parameter (a valid version value)—`-si-revision` alone or with an invalid value—the loader generates an error.
`-v`	Outputs verbose loader messages and status information, as the loader processes files.
`-waits #`	Specifies the number of the wait states for external access. Valid inputs are `0` through `15`. Default is `15`. Wait states apply to the Flash/PROM boot mode only. Note: Currently supported only for ADSP-BF535 processors.
`-width #`	Specifies the loader output file's width in bits. Valid values are `8` and `16`, depending on the boot mode. The default is `8`. The switch has no effect on boot kernel code processing on ADSP-BF535 processors. The loader processes the kernel in 8-bit widths regardless of selection of the output data width. • For Flash/PROM booting, the size of the output file depends on the `-width #` switch. • For SPI booting, the size of the output `.LDR` file is the same for both `-width 8` and `-width 16`. The only difference is the header information.

Using Base Loader

The **Load** page of the **Project Options** dialog consists of multiple panes and is the same and for all Blackfin processors. When you open the **Load** page, the default loader settings (**Loader options**) for the selected processor are already set. As an example, Figure 2-14 shows the ADSP-BF532 processor's default **Load** settings for PROM booting. Command-line switches equivalent to the dialog box options are also identified. Refer to "Command-Line Switches" on page 2-42 for more information on the switches.


Project Options

Category: Loader options

Boot mode: [✓ PROM] Flash SPI

Boot format: [✓ Intel hex] ASCII Binary

Output width: [✓ 8-bit] 16-bit

Start address: 0x0

Verbose

Wait state: 0

Hold time: 0

Baud rate: 500k

Initialization file:

Output file:

Additional options:

Figure 2-14. Base Load Page: Loader File Options Pane

Using the page controls, you can select or modify the loader settings. Table 2-10 describes each loader control and corresponding setting. When you are satisfied with default settings, click OK to complete the loader setup.

Table 2-10. Base Loader Page Settings

Setting	Description
Category	Selections in the drop-down box display panes of options. The options are: • Loader options – default booting options (this section) • Boot kernel options – specification for a second-stage loader (see on page 2-49) • ROM splitter options – specification for the no-boot mode (see on page 2-51) If you do not use the boot kernel for ADSP-BF535 processors, the second Load pane appears with all kernel option fields grayed out. The loader does not search for the boot kernel if you boot from the on-chip ROM by setting the `-no2kernel` command-line switch as described on page 2-45. For ADSP-BF531/BF532/BF533 and ADSP-BF561 processors, which do not have software boot kernels by default, you need to select the boot kernel to use one.
Boot mode	Specifies PROM, Flash, or SPI as a boot source.
Boot format	Specifies Intel hex, ASCII, or binary formats.
Output width	Specifies 8 or 16 bits. If `BMODE = 01` or `001` and Flash/PROM is 16-bit wide, the 16-bit option must be selected.
Start address	Specifies a PROM/Flash output start address in hex format for the application code.
Verbose	Generates status information as the loader processes the files.
Wait state	Specifies the number of the wait states for external access (`0–15`). The selection is active for ADSP-BF535 processors. For ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors, the field is grayed out.
Hold time	Specifies the number of the hold-time cycles for PROM/Flash boot (`0–3`). The selection is active for ADSP-BF535 processors. For ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors, the field is grayed out.

Table 2-10. Base Loader Page Settings (Cont'd)

Setting	Description
Baud rate	Specifies a baud rate for SPI booting (`500` kHz, `1` MHz, and `2` MHz). The selection is active for ADSP-BF535 processors. For ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors, the field is grayed out.
Initialization file	Directs the loader to include the initialization file (Init code). The **Initialization file** selection is active for ADSP-BF531/BF532/BF533, and ADSP-BF561 processors. For ADSP-BF535 processors, the field is grayed out.
Kernel file	Specifies the boot kernel file. Can be used to override the default boot kernel if there is one by default, as on ADSP-BF535 processors.
Output file	Names the loader's output file.
Additional options	Specifies additional loader switches. You can specify additional input files for a multiinput system. **Note:** The loader processes the input files in order the files appear on the command line, starting with the one from the project.

Using Second-Stage Loader

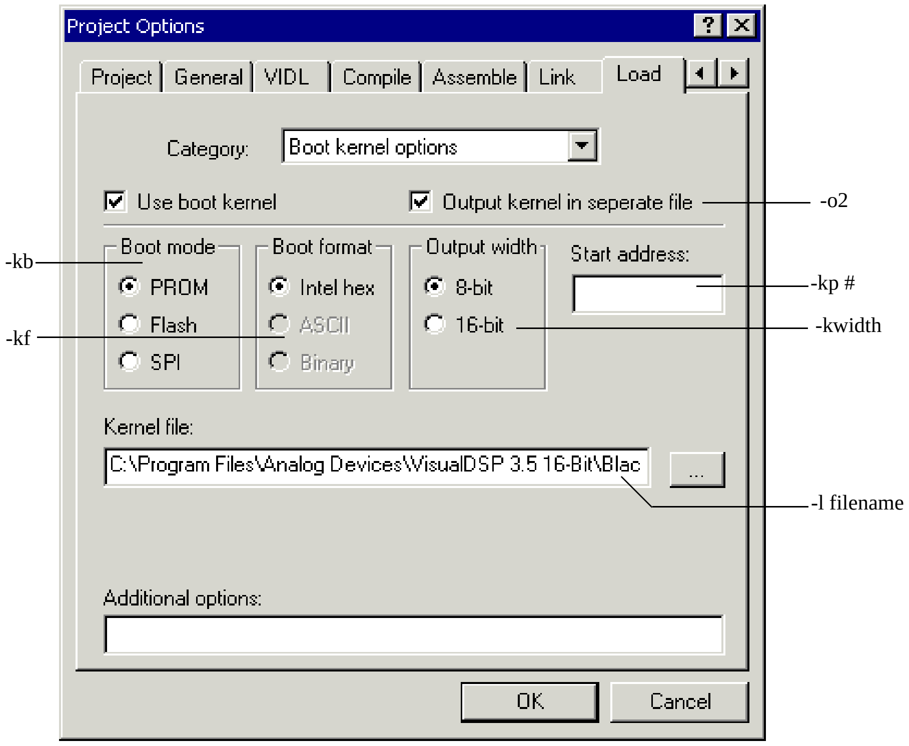
If you to use a second-stage loader, select **Boot kernel options** in the **Category** drop-down menu. The page shows how to configure the loader for boot loading and to output file generation using the boot kernel.

Figure 2-15 shows an example boot kernel **Load** pane for a Blackfin processor.

To create a loader file which includes a second-stage loader:

1. Use the **Loader options** pane to set up base booting options (see "Using Base Loader" on page 2-47).

2. Select **Boot kernel options** from the **Category** drop-down box to open the second **Load** pane with the second-stage loader settings, shown in Figure 2-15.



Project Options

Category: Boot kernel options

✓ Use boot kernel

✓ Output kernel in seperate file

Boot mode: [✓ PROM] Flash SPI

Boot format: [✓ Intel hex] ASCII Binary

Output width: [✓ 8-bit] 16-bit

Start address:

Kernel file: C:\Program Files\Analog Devices\VisualDSP 3.5 16-Bit\Blac

Additional options:

Figure 2-15. ADSP-BF53x Processors: Boot Kernel Pane

3. Select **Use boot kernel**. By default, this option is selected for ADSP-BF535 and grayed out for ADSP-BF531/BF532/BF533 and ADSP-BF561 processors.

4. If you want to produce two output files (boot kernel file and application code file), select the **Output kernel in separate file** check box. This option boots the second-stage loader from one source and the application code from another source. If the **Output kernel in separate file** box is selected, you can specify the kernel output file options such as the **Boot mode** (source), **Boot format**, and **Output width**.

5. Enter the Kernel file (`.DXE`). You must either use the default kernel (in case of a ADSP-BF535 processor) or enter a kernel filename if Use boot kernel in step 3 is selected.

   The following second-stage loaders are currently available for the ADSP-BF535 processor.

Boot Source	Second Stage Loader File (or Boot Kernel File)
8-bit Flash/PROM	`535_prom8.dxe,`
16-bit Flash/PROM	`535_prom16.dxe`
SPI	`535_spi.dxe`

For ADSP-BF531, ADSP-BF532, ADSP-BF533, and ADSP-BF561 processors, no second-stage loaders are required; hence, no default kernel files are provided. Users can supply their own second-stage loader file, if so desired.

6. Specify the Start address (FLash/PROM output address in hexadecimal format) for the kernel code. This option allows you to place the kernel file at a specific location within the Flash/PROM in the loader file.

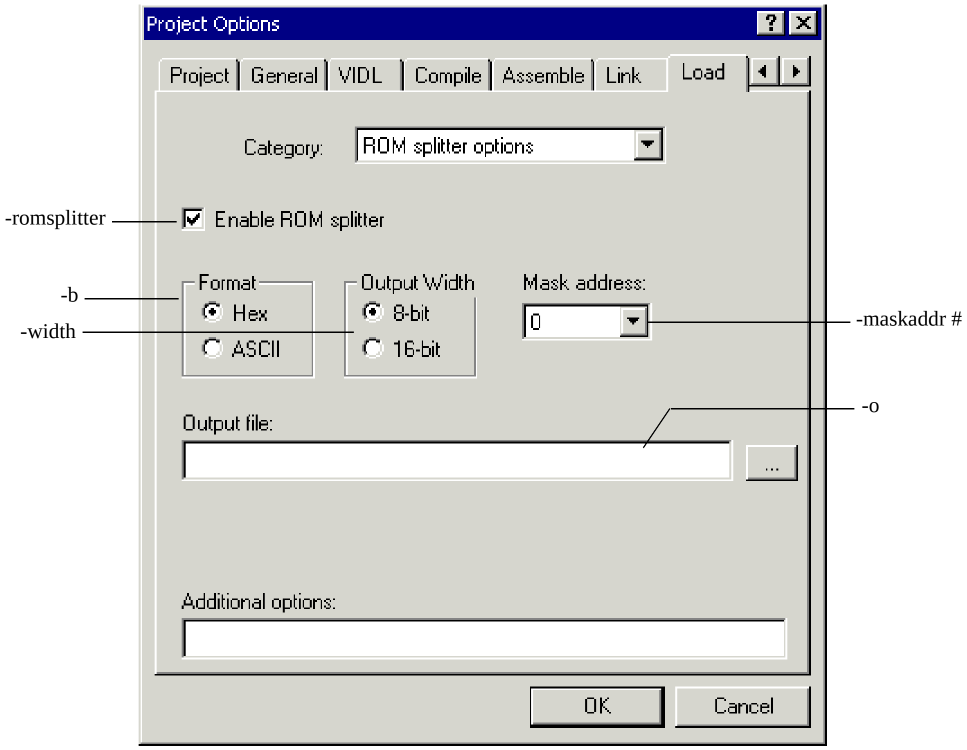
7. For ADSP-BF535 processors only, modify the Wait states and Hold time cycles for Flash/PROM booting or the Baud rate for SPI booting.

8. Click OK to complete the loader setup.

Using ROM Splitter

Unlike the loader utility, the splitter does not format the application data when transforming an `.DXE` file to an `.LDR` file. It emits raw data only. Whether data and/or instruction segment are processed by the loader or splitter utility is controlled by the LDF's `TYPE()` command. Segments declared with `TYPE(RAM)` are consumed by the loader utility, and segments declared by `TYPE(ROM)` are consumed by the splitter.

Figure 2-16 shows an example ROM splitter options pane of the Load page. With the Enable ROM splitter box unchecked, only `TYPE(RAM)` segments are processed and all `TYPE(ROM)` segments are ignored by the elfloader utility. If the box is checked, `TYPE(RAM)` segments are ignored and `TYPE(ROM)` segments are processed by the splitter utility.



Category: ROM splitter options

[✓] Enable ROM splitter

Format: [✓ Hex] ASCII

Output Width: [✓ 8-bit] 16-bit

Mask address: 0

Output file:

Additional options:

Figure 2-16. ROM Splitter Pane

The Mask Address field masks all EPROM address bits above or equal to the number specified. For example, Mask Address = `29` (default) masks all the bits above and including `A29` (ANDed by `0x1FFF FFFF`). Thus, `0x2000 0000` becomes `0x0000 0000`. The valid numbers are integers `0` through `32` but, based on your specific input file, the value can be within a subset of [`0`, `32`].

No-boot Mode

The hardware settings of `BMODE = 000` for ADSP-BF535 processors or `BMODE = 00` for ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors select the no-boot option. In this mode of operation, the on-chip boot kernel is bypassed after reset and the processor starts fetching and executing instructions from address `0x2000 0000` in the Asynchronous Memory Bank 0. The processor assumes 16-bit memory with valid instructions at that location.

To create a proper `.LDR` file that can be burned into either a parallel Flash or EPROM device, you must modify the standard LDF file in order the reset vector is to be located accordingly. The following code fragments illustrate the required modifications in case of an ADSP-BF533 processor.

Listing 2-3. Section Assignment (LDF File)

```
MEMORY
{
   /* Off-chip Instruction ROM in Async Bank 0 */
  MEM_PROGRAM_ROM { TYPE(ROM) START(0x20000000) END(0x2009FFFF)
WIDTH(8) }
   /* Off-chip constant data in Async Bank 0   */
   MEM_DATA_ROM    { TYPE(ROM) START(0x200A0000) END(0x200FFFFF)
WIDTH(8) }
   /* On-chip SRAM data, is not booted automatically */
   MEM_DATA_RAM    { TYPE(RAM) START(0xFF903000) END(0xFF907FFF)
WIDTH(8) }
```

Listing 2-4. ROM Segment Definitions (LDF File)

```
PROCESSOR p0
{
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

  SECTIONS
  {
    program_rom
    {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS( $OBJECTS(rom_code) )
    } >MEM_PROGRAM_ROM
    data_rom
    {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS($OBJECTS(rom_data) )
     } >MEM_DATA_ROM
    data_sram
    {
         INPUT_SECTION_ALIGN(4)
         INPUT_SECTIONS($OBJECTS(ram_data) )
    } >MEM_DATA_RAM
```

With the LDF file modified this way, the source files can now take advantage of the newly introduced sections, as in Listing 2-5.

Listing 2-5. Section Handling (Source Files)

```
.SECTION rom_code;
_reset_vector: l0 = 0;
               l1 = 0;
               l2 = 0;
               l3 = 0;
               /* continue with setup and application code */
```

```
                /* . . . */
.SECTION rom_data;
.VAR myconst x = 0xdeadbeef;
                /* . . . */
.SECTION ram_data;
.VAR myvar y; /* note that y cannot be initialized automatically */
```

3 ADSP-219X DSP LOADER/SPLITTER

This chapter explains how the loader/splitter program (`elfloader.exe`) is used to convert executable files (`.DXE`) into boot-loadable, no-bootable, or combined output files for ADSP-219x DSPs.

"ADSP-219x loader/splitter" refers to the loader/splitter program designed for ADSP-2191, ADSP-2195, ADSP-2196, ADSP-21990, ADSP-21991, and ADSP-21992 DSPs. The ADSP-2192-12 loader is described in Chapter 4, "ADSP-2192-12 DSP Loader" on page 4-1.

Refer to "Introduction" on page 1-1 for the loader overview; the introductory material applies to all processor families. Loader operations specific to the listed above processors are detailed in the following sections.

- "ADSP-219x DSP Booting" on page 3-2

  Provides general information on boot sequences, kernels, and streams.

- "ADSP-219x DSP Loader Guide" on page 3-19

  Provides reference information on the command-line interface.

ADSP-219x DSP Booting

The ADSP-219x loader/splitter creates a boot stream, non-boot stream, or combinational output. The program accepts one executable file (`.DXE`) as input and generates one file (`.LDR`) as output.

Upon powerup, a ADSP-219x DSP can be booted from the EPROM, UART, SPI, or Host port. Booting can also be initiated in software after `RESET`. Refer to the *Application Note EE-131* and your DSP's data sheet or *Hardware Reference* manual for more information on system configuration, peripherals, registers, and operating modes.

You can run the loader/splitter program from a command-line or from within the VisualDSP++ IDDE. When working within the VisualDSP++, specify options via the **Load** page of the **Project Options** dialog box.

Option setting on the **Load** page correspond to switches displayed on the command line.

To ensure correct operation of the loader, familiarize yourself with:

- "ADSP-219x DSP Boot Modes" on page 3-3
- "ADSP-219x DSP Boot Kernel" on page 3-4
- "ADSP-219x DSP Boot Streams" on page 3-4
- "Parallel EPROM Boot Streams" on page 3-4
- "Host Booting" on page 3-10
- "UART Booting" on page 3-11
- "Serial EPROM Booting" on page 3-12
- "No-booting" on page 3-12

ADSP-219x DSP Boot Modes

At powerup, after the reset, the processor transitions into a boot mode sequence configured by the `BMODE2-0` pins. The `BMODE` pins are dedicated mode-control pins; the pin states are captured and placed in the Reset Configuration register as `BMODE0`, `BMODE1`, and `OPMODE` (see Table 3-1). The register is also known as the System Configuration Register (`SYSCR`) with I/O address `0x0 0204`.

Table 3-1. ADSP-219x DSP Operation Modes

BMODE1 Pin	BMODE0 Pin	OPMODE Pin	Description
0	0	0	No-boot mode. Run from external 16-bit memory at logical address `0x10000`. Bypass ROM.
0	1	0	Boot from EPROM.
1	0	0	Boot from Host.
1	1	0	Reserved.
0	0	1	No-boot mode. Run from external 8-bit memory at logical address `0x10000`. Bypass ROM.
0	1	1	Boot from UART.
1	0	1	Boot from SPI (up to 4K bits).

The `OPMODE` pin has a dual role; it acts as a boot-mode select during `RESET` and determines whether the DSP's third SPORT functions as a SPORT or an SPI. It is possible for an application to require `OPMODE` to operate differently at runtime than at `RESET` (that is, boot from an SPI but use SPORT2 during runtime). In this case, the boot kernel is responsible for setting `OPMODE` accordingly at the end of the booting process. Therefore, software can change `OPMODE` anytime during runtime, as long as the corresponding peripherals are disabled at that time.

ADSP-219x DSP Boot Kernel

A loader boot kernel refers to the resident program stored in a 24-bit-wide, 1K portion of ROM space responsible for booting the DSP. The starting address of the boot ROM is `0xFF 0000` to `0xFF 03FF`, the first location of page 256, in 1-wait-stated memory. A boot interrupt vectors to address `0xFF 0000`. When a ADSP-219x DSP comes out of a hardware reset, program control jumps to `0xFF 0000`, and execution of the boot ROM code begins.

On ADSP-219x DSPs, the highest 16 locations in page 0 program memory and the highest 272 locations in page 0 data memory are reserved for use by the ROM boot routines (for setting up DMA data structures and for initializing registers, among other tasks). Ensure that the boot sequence entry code or boot-loaded program are not allowed into this space.

ADSP-219x DSP Boot Streams

The ADSP-219x ROM-resident loader is designed to parse and load a specific boot-stream format. When booting from an external 8- or 16-bit EPROM, the boot stream consists of header and block fields.

The first header in the boot stream is a common word that applies to all booting modes, except UART. This header field specifies whether the stream is guarded by a `checksum`. Individual bits within this word are set or cleared, based on the booting method and specific command-line switches specified by the user.

Parallel EPROM Boot Streams

When booting from an external 8- or 16-bit EPROM, the first 16-bit header field contains information on the number of wait states and the physical width (8- or 16-bit) of the EPROM. The first header is also known as a global header or a control word.

This first header is followed by the regular boot stream, that is, a series of headers and data blocks. Most headers are followed by corresponding blocks of data, but some headers indicate regions of memory that need to be "zero-filled" and are not followed by data blocks.

Block Headers

Each block header consists of four or six 16-bit words:

- The first word consists of a flag that indicates whether the following block of data is a 24-bit or 16-bit payload or zero-initialized data. The flag uniquely identifies the last block that needs to be transferred.
- The second word contains the lower 16 bits of the 24-bit start address for data loading (destination). The first octet is the 8 LSBs, followed by the next most significant bits (`15-8`), and so on.
- The third word contains the upper-most 8 bits of the 24-bit destination address, padded (suffixed) with one byte of zeros.
- The fourth word contains the payload's word count. Similar to the address, the first octet is the 8 LSBs, and the second octet is the 8 MSBs.

An extra word appears when a `checksum` function is used to verify booting accuracy. This fifth word (also a 16-bit field) is the CRC-16 `checksum` for the header, and the data block immediately follows it. The CRC `checksum` word is optional. Activate the `checksum` by running the loader utility with the `-checksum` command-line switch (on page 3-21).

The header is buffered with a dummy word of zeros to ease the EMI addressing issue.

EPROM booting can be performed in an 8-bit or 16-bit scenario. This information is controlled by the "-width #" switch and is finally embedded in the boot stream. Unlike non-boot mode, bus width is not controlled by the mode pins. The width information configures the EMI interface and remains valid during the entire boot process. If you want to boot off-chip memories, be aware that the width of the memory you want to boot must be identical to the width of the interface from which you are booting.

Data Blocks

The header is followed by the data block (also called payload data). The 16-bit block is sent in a 16-bit field, and the 24-bit block is sent in a 32-bit field.

24-bit data block is represented differently in the boot stream from 24-bit addresses. 32-bit data block is transmitted the following way—a byte of zeros (inserted by the loader), bits 0-7, followed by bits 8-15, and finally bits 16-24.

When booting from an 8- or 16-bit EPROM, direct DSP core accesses and Memory DMA (under the control of an EPROM boot routine located in the ROM space) are used to load a boot-stream formatted program located in the boot space. Appropriate packing modes are selected, based on the requirements of the boot stream.

Each page of boot space is 64K words long, and 16-bits can address the EPROM per page. The upper 8 address bits specify the boot page. Upon hardware reset, booting is from address `0x0000` of logical page `0x80`, which mirrors physical address `0x000000` because the upper address lines are not available off-chip.

The External Port Interface (EPI) uses its default configuration (divide-by-128 clock and 7 wait states) to access the EPROM. While booting via EPROM boot space, the highest 16 locations in page 0 program memory block (`0x7FF0` to `0x7FFF`) and the top 272 locations of page 0 data memory block (`0xFEF0` to `0xFFFF`) are reserved for use by the ROM boot routines.

ADSP-219x DSP Multiple .DXE Support

VisualDSP++ 3.5 introduces support for multiple `.DXE` booting in Parallel EPROM booting mode. Boot streams of multiple projects or applications can be stored in a single EPROM. The on-chip boot kernel always boots in application number 0. Its boot stream starts at EPROM address `0x000000`. User-defined second-stage loaders or boot management software may boot any application depending on application-specific circumstances. Alternatively, one application can be loaded and executed after the other terminates. The loader/splitter utility can consume multiple `.DXE` files and arrange their boot streams in several manners.

Use the following syntax to submit two or more executable files to the loader:

```
elfloader File1.dxe [outputfile] -proc processor [-switch ...]
[-pd addr [switches_specific_to_dxe_file_that_follows]] File2.dxe
[-pd addr [switches_specific_to_dxe_file_that_follows]] File3.dxe ...
```

`File1.dxe` is the default application that is booted by the on-chip boot kernel after reset. Unless the `-p` switch is specified, the boot stream of `File1.dxe` starts at EPROM address `0x000000`.

If there is a `-pd` *`addr`* switch specified for an executable file (*`File.dxe`*), the switches between "`-pd`" and "*`File.dxe`*" are called a *pd grouping*. The "*`addr`*" parameter is the address in the byte-based PROM address space. The address should be a hex value. If `-pd` *`addr`* is absent from a command line, no pd grouping is associated with the executable file, but the address in the byte-based PROM address space remains associated with the `.DXE`.

Usually the `-pd` *`addr`* switch is applied if the individual boot streams need to be located a given addresses. For example, when the individual boot streams need to reside is certain pages of a Flash memory device. If the different boot streams are going to reside in different physical PROM devices, this syntax enables the user to assign different settings, such as wait-states or even output file name to the individual `.DXE`s.

If more than one `.DXE` file is listed at the command line without `-pd` switch, the loader utility appends the boot stream of the second `.DXE` immediately to the one of the first `.DXE` and so on. Executable files inherit boot stream settings from previous one if not explicitly set by a `-pd` grouping.

The pd. grouping enables various options for the loader stream of the input `.DXE` file. In the multiinput `.DXE` scenario, the loader stream for each executable is same as when there is only one input executable supplied to `elfloader`, with an exception of:

1. An artificial loader block (with header and data block) is created right after the first (global) 16-bit header and before the regular boot stream. The destination address in the header is same as that of the first loader block in the regular boot stream. This means that any data booted by the artificial loader block is to be overwritten by the real data from the regular boot stream.

   The content of this block's payload is the pd value for the next `.DXE`. If no pd grouping is specified for the next `.DXE`, the pd value for the next `.DXE` is calculated by adding the pd value of the current `.DXE` and the size (in bytes) of the current `.DXE` loader stream. If there are no other `.DXE`s in the stream, the default value is zero. You can overwrite the default by the `-pdAddrNext` switch.

   The artificial loader block can be removed by turning on the `-noDxeAddrHdr` command-line option.

2. The loader stream version, which is 4 bits in the first 16-bit header.

Note that both of these modifications are backward compatible with the ADSP-2191 boot kernel. The optional [*`switches_specific_to_dxe_file_that_follows`*] may include:

Example

`-o`	Output file (`.LDR`) for the current and following `.DXEs` (see "-o filename" on page 3-22).
`-opmode`	Opmode for the current and following `.DXEs` (see "-opmode #" on page 3-22).
`-p`	The Intel hex offset for the current loader file (see "-p address" on page 3-22).
`-width`	Width for the current and following `.DXEs` (see "-width #" on page 3-24).
`-wait`	The number of wait states for this and following `.DXEs` (see "-waits #" on page 3-24).
`-clkdivide`	Clkdivide for this and following `.DXEs` (see "-clkdivide #" on page 3-21).
`-maskaddr`	Address bits to be masked off for this and following `.DXEs` (see "-maskaddr #" on page 3-22).

There are four executable files (streams):

- Application 1 (`app1.dxe`) starts at byte address `0x000000` (ROM)
- Application 2 (`app2.dxe`) appends to Application 1 (ROM)
- Application 3 (`app3.dxe`) starts at `0x020000` (flash page 1)
- Application 4 (`app4.dxe`) starts at `0x030000` (flash page 2)

The task is to create two loader files, one is 16-bit ROM from `0x000000` to `0x01FFFF` and one is 8-bit Flash from `0x020000` to `0x03FFFF`.

There are two different ways to accomplish the task:

1. Use VisualDSP++ Flash Programmer plug-in to burn the Flash. In this scenario, the real byte address is expected:

```
elfloader -proc ADSP-2191 -b PROM -width 16 app1.dxe
app2.dxe -pd 0x20000 -p 0x20000 -width8 -o flash.ldr
app3.dxe -pd 0x30000 app4.dxe
```

Since the `-p` value is reset to a zero whenever an `-o` is specified, the addresses in the Intel hex record starts at zero for `flash.ldr`.

2. Equivalently, invoke the loader twice:

```
elfloader -proc ADSP-2191 -b PROM -width 16 app1.dxe
app2.dxe -pdAddrNext 0x20000
```

```
elfloader -proc ADSP-2191 -b PROM -width 8 -o flash.ldr -pd
0x20000 -p 0x20000 app3.dxe -pd 0x30000 app4.dxe
```

In cases where the `-pd` and `-p` values are expected to be the same, you may specify `-pEqualPD`:

```
elfloader -proc ADSP-2191 -b PROM -width 16 app1.dxe
app2.dxe -pd 0x20000 -pEqualPD -width8 -o flash.ldr
app3.dxe -pd 0x30000 app4.dxe
```

With the `-NoDxeAddrHdr` switch, this artificial block is not inserted. Then the user loader can still parse the complete boot stream, block by block until it detects a final-init block. Since the default `-p` value is reset to a zero whenever an `-o` is specified, the addresses in the Intel hex record has to be explicitly set to `0x20000` for `flash.ldr`.

It is very likely that second-stage loaders and similar type of programs execute directly from the EPROM. Thus, this multiple `.DXE` scenarios are often combined with the features discussed in "Enriching Boot EPROMs with No-boot Data" on page 3-16.

Host Booting

Host booting is performed in either an 8-bit or 16-bit scenario. By default, little-endian format is used. If configured in Host boot mode, the DSP does not support the boot process actively. It is the host's responsibility to initialize the DSP memories properly. The DSP passively waits until the host writes a "`1`" to the Semaphore A register (IO address `0x1CFC`). Then the DSP starts fetching and executing instructions at address `0x00 0000`.

It is recommended that the host parses the boot stream and downloads segment by segment. The `elfloader.exe` may store the boot stream as an Intel hex-32 file typically required by embedded host devices. PC-based hosts may favor the ASCII format, which stores one byte or one 16-bit word per line.

UART Booting

When booting via the UART port, a host downloads a boot stream formatted program to the DSP following an auto-baud handshake sequence. The auto-baud feature simplifies system design because you do not need to calculate boot clock rates as a function of crystal frequency, core clock divider, peripheral clock mode, and so on.

The booting host can select a baud rate (including a non-standard rate) anywhere within the UART clocking capabilities.

Following a hardware or software reset, the ADSP-219x DSP monitors the UART transceiver channel and expects the predefined character (`0xAA`) to determine the bit rate. The DSP replies an “`OK`” string to acknowledge the bit rate.

Afterwards, the host may send the complete boot stream (8 data bits, no parity, 1 stop bit) without further handshake. The boot stream is decoded by the DSP and starts program execution automatically.

Compared to other formats, the ADSP-2191/2195/2196 UART boot stream suppresses the very first byte. To force the loader utility to include the first byte, use the `-forcefirstbyte` switch (on page 3-21).

This boot operation is controlled by a UART boot routine in the internal ROM space. While booting via UART, the highest 16 locations in page 0 program memory block (`0x7FF0` to `0x7FFF`) and the top 272 locations of page 0 data memory block (`0xFEF0` to `0xFFFF`) are reserved for use by the ROM boot routine.

Serial EPROM Booting

The SPI0 port is used when booting from an SPI-compatible EPROM. The SPI port selects a single serial EPROM device using the `PF0` pin as a chip select, submits a read command and address `0x00`, and begins to clock consecutive data into memory (internal memory or external memory) at a `SCK` clock frequency of HCLK/60. The DSP streams the complete boot image in and processes it without further handshake with the SPI EPROM.

Two types of SPI EEPROM devices are supported: devices of 4K bytes and smaller (12-bit address range), and those larger than 4K bytes (16-bit address range). The SPI boot stream may not exceed 64 kilobytes.

This boot operation is controlled by an SPI boot routine in internal ROM space. While booting via serial EPROM, the highest 16 locations in page 0 program memory block (`0x7FF0` to `0x7FFF`) and the top 272 locations of page 0 data memory block (`0xFEF0` to `0xFFFF`) are reserved for use by the ROM boot routine. Refer to the *Application Note EE-145* for SPI booting examples.

No-booting

When `BMODE2-0` is strapped to a `000` or `001`, the ADSP-219x DSP comes out of hardware reset and begins to execute code from page 1 memory space (`0x01 0000`). The specified packing mode depends on the state of the `BMODE0` pin (`0` = 8-bit `ext`/24-bit `int`, `1` = 16-bit `ext`/24-bit `int`). By default, the External Port Interface (EPI) is configured to operate with the divide-by-128 clock and a read wait-state count of 7.

ⓘ No-boot mode does not use boot-stream format.

After reset, the DSP starts program execution from external address `0x010000`. When the no-boot option is selected, the DSP typically expects an 8- or 16-bit EPROM or Flash device connected to the memory strobe signal (/MS0). Splitter capabilities of the ADSP-219x loader utility support the generation of the required EPROM image files. Refer to the loader's -romsplitter switch (see on page 3-23) for more information.

Non-bootable memory segments are declared by the TYPE(ROM) command in the Linker Description File (.LDF). The WIDTH() command specifies the physical EPROM width (which equals to the EMI port setting). Every .LDF file that belongs to a no-boot project should define a proper memory segment; as in the following example,

```
MEMORY {
. . .
   seg_ext_code {
       TYPE(PM ROM) START(0x010000) END(0x017FFF) WIDTH(16) }
. . .
}
```

The START(), END(), and LENGTH() commands expect logical addresses. Since the example segment stores 24-bit-wide instructions, the TYPE(PM) command defines the logical width of the segment to be 24-bits. The example assumes no-boot mode (000) and runs from external memory starting at address 0x010000. This is why the WIDTH(16) command sets the physical width to 16 bits. Due to ADSP-219x EMI packing rules, the first instruction is stored in the physical 16-bit EPROM location 0x020000.

The 16-bit EPROM address locations between 0x010000 and 0x01FFFF can be used by an additional read-only data segment as shown below.

```
MEMORY {
. . .
   seg_ext_code {
       TYPE(PM ROM) START(0x010000) END(0x017FFF) WIDTH(16) }
   seg_ext_data {
       TYPE(DM ROM) START(0x010000) END(0x01FFFF) WIDTH(16) }
. . .
}
```

The data segment `seg_ext_data` is defined by the `TYPE(DM)` command, which sets the logical width to 16 bits. Since the `WIDTH(16)` command also sets the physical width to 16 bit, no data packing and no address multiply is required. Logical addresses are equal to physical EPROM addresses in this special case. The 16-bit EPROM image generated by the loader is described in Table 3-2:

Table 3-2. EPROM Image Description

Address range	Purpose
`0x000000-0x00FFFF`	Not used
`0x010000-0x01FFFF`	`seg_ext_data`
`0x020000-0x02FFFF`	`seg_ext_code`

The DSP cannot access off-chip addresses lower than `0x010000`. Typically this address space is accessed by taking advantage of address aliasing. The memory strobe (`/MS0`) covers an address range from `0x010000` to `0x400000`. For example, if the EPROM size is less than 4M words and the EPROM is the only device connected to `/MS0`, the first 64K words can be accessed through addresses `0x200000` to `0x20FFFF`.

If a project consists only of two segments (`seg_ext_data` and `seg_ext_code`), a 128K x 16-bit EPROM device would be sufficient to store all the required data and instructions. If the loader utility is invoked with the `-maskaddr 17` switch (on page 3-22), all physical address bits greater than or equal to `A17` are masked out. The loader ANDs all physical addresses with `0x01FFFF`. The resulting EPROM image maps segment `seg_ext_code` into the unused space below `0x010000` (see Table 3-3).

This way, a 128K x 16-bit EPROM can be burned properly. At runtime, `seg_ext_code` aliases back to the addresses above `0x020000` when address lines `A17` through `A21` are not connected.

Table 3-3. EPROM Image—Two Segments Only

Address range	Purpose
`0x000000 - 0x00FFFF`	`seg_ext_data`
`0x010000 - 0x01FFFF`	`seg_ext_code`

Typically, the loader utility generates an Intel hex-32 file, which is readable by most EPROMs. If the image must be post-processed, the loader may also generate ASCII files.

DM Example:

```
ext_data { TYPE(DM ROM) START(0x010000) END(0x010003) WIDTH(8) }
```

The above DM segment results in the following code.

```
00010000   // 32-bit logical address field
00000004   // 32-bit logical length field
00020201   // 32-bit control word: 2x address multiply
           // 02 bytes logical width, 01 byte physical width
00000000   // reserved
1234       // 1st data word, DM data is 16 bits
5678
9ABC
DEF0       // 4th (last) data word
CRC16      // optional, controlled by the -checksum switch
```

PM Example:

```
ext_code { TYPE(PM ROM) START(0x040000) END(0x040007) WIDTH(16)}
```

The above PM segment results in the following code.

```
00040000   // 32-bit logical address field
00000008   // 32-bit logical length field
00020302   // 32-bit control word: 2x address multiply
           // 03 bytes logical width, 02 bytes physical width
00000000   // reserved
123456     // 1st data word, PM data is 16 bits
```

```
789ABC
DEF012
345678
9ABCDE
F01234
56789A
BCDEF0      // 8th (last) data word optional,
            // controlled by the -checksum switch
```

Enriching Boot EPROMs with No-boot Data

The loader's splitter functionality (refer to "No-booting" on page 3-12) enables powerful memory utilization in combination with the parallel EPROM boot mode. The same EPROM used for booting can also be used at runtime for read-only data and overlay storage. Furthermore, the DSP can execute non-speed-critical parts of the program directly from the EPROM, whereas real-time algorithms have been booted into on-chip memory.

When invoked with the `-readall` switch (on page 3-23), the loader processes all kinds of LDF segments. `TYPE(RAM)` segments are passed to the loader's boot stream generator, and `TYPE(ROM)` segments are passed to its splitter. Boot stream and splitter data can be combined within a single EPROM image.

Assuming a cost-sensitive application comprising an ADSP-2196 DSP and a 64-Kbyte EPROM, the boot stream probably does not exceed 40 kilobytes (8K x 3 bytes + 8K x 2 bytes) of length. The rest of the EPROM can be used to store different sets of coefficients and the slow initialization and control code. A reasonable organization of the 8-bit EPROM is described in Table 3-4.

Since the DSP cannot access off-chip memories with addresses lower than `0x010000`, it needs to access segments `seg_ext_data` and `seg_ext_code` through alias windows. If only address lines A0 through A15 are connected, address `0x00 A000` aliases to any `0x00 A000` address. Segment `seg_ext_data` stores 16-bit data. Thus, its physical addresses must be

Table 3-4. EPROM Image With No-boot Data

Address range	Purpose
0x000000-0x009FFF	Boot stream
0x00A000-0x00AFFF	seg_ext_data (External read-only data)
0x00B000-0x00FFFF	seg_ext_code (External program)

divided by 2 to obtain the corresponding logical address. The first alias window of seg_ext_data that can accessed properly is range 0x02 A000 to 0x02 AFFF; this results in the logical range 0x01 5000 to 0x01 57FF.

Similarly, the addresses of segment seg_ext_code can be calculated by dividing the physical addresses by 4. For example, the alias window between 0x04 B000 and 0x04 FFFF can be used, resulting in the logical addresses 0x01 2C00 to 0x01 3FFF.

The corresponding .LDF file would include the following.

```
MEMORY {
. . .
   seg_int_code {
       TYPE(PM RAM) START(0x000000) END(0x000000) WIDTH(24) }
   seg_int_data {
       TYPE(DM RAM) START(0x008000) END(0x009FFF) WIDTH(16) }
   seg_ext_code {
       TYPE(PM ROM) START(0x012C00) END(0x013FFF) WIDTH(8)  }
   seg_ext_data {
       TYPE(DM ROM) START(0x015000) END(0x0157FF) WIDTH(8)  }
...
}
```

By default, the elfloader emits true EPROM addresses by multiplying the logical addresses accordingly. The address aliasing, which this example takes advantage of, can be corrected if the loader is invoked with the

`-maskaddr 16` switch (see on page 3-22). Then all EPROM addresses are ANDed by `0xFFFF`, and the resulting EPROM image fits into a 64-Kbyte EPROM.

The EPROM boot process assumes the boot device is connected to the DSP's `/BMS` strobe. During runtime, typically the `/MSx` strobes are used. To use one EPROM for both booting and run-time issues, set the proper `/BMS` control bits in the `E_STAT` register. If several devices are connected to the individual `/MSx` strobes, an off-chip AND gate is recommended to OR the `/BMS` and the `/MS0` strobes properly.

Please refer to the *Application Note EE-164* for further details and code example.

ADSP-219x DSP Loader Guide

This section provides reference information about the loader's command line interface. A list of the command-line switches appears in Table 3-6 on page 3-21.

When using the loader within VisualDSP++, settings on the Load page of the Project Options dialog box correspond to the loader's command-line switches. For more information, see the *VisualDSP++ 3.5 User's Guide for 16-Bit Processors* or online Help.

ADSP-219x Loader Command-Line Reference

Use the following syntax for the loader's command line.

```
elfloader sourcefile [outputfile] -proc processor [-switch …]
```

where:

- `sourcefile`—Identifies the executable file (`.DXE`) to be processed into a single-processor boot-loadable file. A file name can include the drive and directory. Enclose long file names within straight-quotes, "`long file name`".
- `outputfile`—Optional name of the loader's output, a file with the `.LDR` extension. Each run generates a single output file.
- `-proc processor`—Part number of the processor for which the loader file is to be built. Provide a part number for every input `.DXE` if designing multi-processor systems. Running the loader without `-proc` results in an error.
- `-switch …`—One or more optional switches to process. Switches select operations and modes for the loader. See Table 3-5 on page 3-20 for a complete list of the loader command-line switches.

Example:

```
elfloader p0.dxe -proc ADSP-2191
```

In the above example, the loader runs with:

- `p0.dxe`—the name of the executable file to be processed into a boot-loadable file. The output file's name is `p0.ldr` because a name is not specified.
- `-proc ADSP-2191`—the target processor, ADSP-2191 DSP.

File Searches

Many loader switches take a file name as an optional parameter. Table 3-5 on page 3-20 lists the expected file types. File searches are important in the loader operation. The loader supports relative and absolute directory names, default directories. File searches occur as described on page 1-9.

File Extensions

Table 3-5 lists and describes file types input and output by the loader.

Table 3-5. File Extensions for ADSP-218x Loader Operation

File Extension	Description
`.DXE`	Executable files and boot-kernel files.
`.OVL`	Overlay memory files. The loader recognizes overlay memory files but does not expect these files on the command line. Place `.OVL` files in the same directory as the `.DXE` file that refers to them; the loader can locate them when processing the `.BNM` file.
`.LDR`	Loader output file.

Loader Switches

A description of each loader command-line switch appears in Table 3-6.

Table 3-6. Loader Command-Line Switches

Switch	Description
filename	Specifies an executable file to be processed into a single-processor loadable file. For multiprocessor system, use the `-id#exe=file` switch.

`-b prom`
`-b host`
`-b uart`
`-b spi`
	Specifies the boot mode. Prepares a boot-loadable file for the `PROM` (default), Host, UART, SPI, or no-boot booting. The specified mode must correspond to the boot kernel selected with the `-l` switch and the file format selected with the `-f` switch.

`-blocksize #`	Specifies the size (decimal #) in words for each block of data in the boot stream. Default is 6K words. Valid block sizes may vary up to 6K words.
`-checksum`	Calculates and generates checksums for each block of code and data.
`-clkdivide #`	Specifies the base clock divide factor. Valid values are `0` to `7`, inclusive. The default is `5`. Note: Applies to EPROM and Host boot modes only.

`-f hex`
`-f ASCII`
`-f binary`
	Specifies the boot file's format.
	Valid selections are `hex` (Intel hex-32), `ASCII`, and `binary`. The hexadecimal format is the default. For PROM booting, Intel `hex` is the only valid entry. When `-f ASCII` and `-romsplitter` are selected, regardless of the `-b file_type` setting, an ASCII file is produced.

`-forcefirstbyte`	Forces the writing of the first byte of the UART boot stream. Use this option when the bit rate is high.

`-h`
or
`-help`
	Invokes the command-line help, outputs a list of command-line switches to standard output, and exits. By default, the `-h` switch alone provides help for the loader driver. To obtain a help screen for the ADSP-2191 processor, type `-elfloader -proc ADSP-2191 -h`.

`-host3bytes`	Uses three bytes to represent 24-bit data. The default is four bytes with one byte of zero padding. Note: Used with Host boot only, when width is 8 bit.
`-M`	Shows dependencies only.
`-MM`	Generates make dependencies while producing the output files.

Table 3-6. Loader Command-Line Switches (Cont'd)

Switch	Description
`-Mo` *`filename`*	Writes make dependencies to the named specified. The `-Mo` option is for use with either the `-M` or -`MM` option. If `-Mo` is not present, the default is `<stdout>` display.
`-Mt` *`targetname`*	Specifies the make dependencies target name. The `-Mt` option is for use with either the `-M` or -`MM` option. If `-Mt` is not present, the default is the name of the input file with the `.DOJ` extension.
`-maskaddr #`	Masks all EPROM address bits above or equal to `#`. For example, `-maskaddr 18` masks all the bits above and including `A18` (ANDed by `0x3FFFF`). The switch does not require `-romsplitter` and affects ROM section address only.
`-NoDxeAddrHdr`	Does not generate the address header in the loader stream for the input `.DXE` file.
`-o` *`filename`*	Specifies the name of the output file. If no file name is specified, the output file takes the name of the input file. The extension is `.LDR`. For multi `.DXE` processing, when the `-o` *`filename`*`.ldr` is specified inside the `-pd` grouping, the file-relative byte address (that is, the value in the address portion of the Intel hex information) is set to zero. The value of zero is the default value but also can be set by `-PEqualZero`. The value can be set to the value provided to the `-pd` switch (specified by the `-PEqualPD` switch). Further, it also can be set by specifying a `-p` argument within the `-pd` grouping. See "ADSP-219x DSP Multiple .DXE Support" on page 3-7 and "-pd [address] inputfile" for more information on pd groupings.
`-opmode #`	Specifies whether the boot kernel sets the DSP to 3-SPORT mode (`0`) or 2-SPORT/1-SPI mode (`1`) at the end of booting. Default is `0`.
`-p` *`address`*	Specifies a hexadecimal integer as the PROM starting address. For multi `.DXE` processing, if `-p is` specified in the `-pd` group, a new `.LDR` file must be created using `-o` in the same pd group. See "-o file-name" and "-pd [address] inputfile" for details.
`-pEqualPd`	Used inside the "-pd [address] inputfile" grouping. Sets the default value for `-p` when `-o` specified in the `-pd` group is the value of the argument to `-pd`.
`-pEqualZero`	Used inside the "-pd [address] inputfile" grouping. Sets the default value for `-p` when `-o` specified in the pd group is zero (default behavior).

Table 3-6. Loader Command-Line Switches (Cont'd)

Switch	Description
`-pd` [*address*] *inputfile*	Appends another application program (`.DXE`) at the specified address. You can specify options applicable to the boot-stream general header between *address* and *inputfile*. If no `-pd` [*address*] is specified, application starts from the next free PROM location. There is no restriction on the number of time the switch is used.
`-pdAddrNext`	Specifies the value contained in the .DXE address header block. When elfloader is invoked multiple times to create loader files, this option specifies the next address to be placed in the new header record for the last `.DXE` on the command line. The address is in the byte-based PROM address space.

`-proc processor`
or
`-dADSP-21xx`
	The mandatory switch specifies the processor for which the loader file is created. For example, `-proc ADSP-2191` or `-proc ADSP-21990`.
	Note: `-proc ADSP-21xx` is the preferred form; `-dADSP-21xx` is for legacy support only.

`-readall`	Creates a non-bootable image and non-boot stream image in the same output file, together with the boot-loadable image. Boot mode must be set to PROM (`-b PROM`) and format must be set to hex (`-f hex`).
`-romsplitter`	Creates a non-bootable image only. This switch overwrites `-b` and any other switch bounded by boot types. An ASCII file is produced when `-f ASCII` and `-romsplitter` are specified, regardless of the `-b` *file_type* setting.
`-split # [#]`	The `-split 8 8` switch generates two output files for the 16-bit-wide EMI data bus; the `.LDU` file contains the upper eight data bits, and the `.LDL` file contains the lower eight data bits. The `-split 16` switch produces one 16-bit-wide file. Note: Valid only when width is 16 bit.
`-v`	Outputs status information as the loader processes files.

Table 3-6. Loader Command-Line Switches (Cont'd)

Switch	Description
`-waits #`	Determines the number of wait states for external accesses. Valid inputs are `0` to `7` (inclusive). Default is `7`. Note: For EPROM and Host boot modes only.
`-width #`	Specifies the bus width (in bits) for EPROM/Flash or Host booting. Valid numbers are `8` (default) and `16`. Width must correspond to the `EMICTL` register's `E_BWS` bit. For multi `.DXE` processing, if the width changes from one `-pd` group to the next, a new `.LDR` file must be created by specifying `-o` in the pd group where `-width` has changed. See "-o filename" and "-pd [address] inputfile" for details.

4 ADSP-2192-12 DSP LOADER

This chapter explains how the loader program (`elfloader.exe`) is used to convert executable files (`.DXE`) into boot-loadable files (`.H`) for ADSP-2192-12 DSPs.

You cannot produce a non-bootable PROM image file (that is, splitting is not supported) for an ADSP-2192-12 DSP.

Refer to "Introduction" on page 1-1 for the loader overview; the introductory material applies to all processor families. Loader operations specific to ADSP-2192-12 DSPs (ADSP-2192, for short) are detailed in the following sections.

- "ADSP-2192 DSP Booting" on page 4-2

  Provides general information on the loader commands and operations.

- "ADSP-2192 DSP Loader Guide" on page 4-10

  Provides reference information on the loader's command-line syntax and switches.

ADSP-2192 DSP Booting

An ADSP-2192-12 DSP can be boot-loaded through its PCI or USB interface. For PCI loading, the loadable executable (`.EXE`) must reside in the PC host's memory space before it is loaded into the DSP.

The ADSP-2192-12 loader repackages `.DXE` files and associated `.OVL` and `.SM` files, produced by the linker, into an `.H` file for use with a run-time loader (RTBL). The RTBL is a drive implemented by the user; the RTBL is responsible for transporting the loadable executable into the DSP. VisualDSP++ includes a reference RTBL that can be used to boot-load an ADSP-2192-12 EZ-KIT Lite evaluation system on Windows 98 and Windows 2000 platforms.

You can run the loader from a command-line or directly from within the VisualDSP++ IDDE. When working from the VisualDSP++, specify the loader options via the **Load** page of the **Project Options** dialog box.

Option setting on the **Load** page correspond to switches displayed on the command line.

To ensure correct operation of the loader, familiarize yourself with:

- "ADSP-2192 DSP Reset Types" on page 4-2
- "ADSP-2192 DSP RTBL" on page 4-4
- "ADSP-2192 DSP RBTL and Overlays" on page 4-8

ADSP-2192 DSP Reset Types

The ADSP-2192 DSP supports booting via the PCI interface or the USB interface. The internal ROM includes a small boot kernel program, which determines how the DSP boots.

Upon recovering from `RESET`, the ADSP-2192 DSP jumps to the first location of the boot ROM at address `0x14000`, which is the start of the boot kernel. The first task performed by the kernel is to determine the type of `RESET` and the source of booting (PCI or USB). The kernel then sets up and initializes appropriate DSP registers to facilitate the booting.

Three methods for resetting the ADSP-2192 DSP are: power-on reset, forced reset via PCI/USB, and software reset. The reset type is specified by bits 8 and 9 (`CRST<1:0>`) of the Chip Mode/Status Register (CMSR) as follows in Table 4-1.

Table 4-1. ADSP-2192-12 DSP CMSR Settings

CMSR Setting	RESET Type
`CRST<1:0>=00`	Power-on reset
`CRST<1:0>=10`	PCI/USB hard reset
`CRST<1:0>=10`	PCI/USB hard reset
`CRST<1:0>=11`	Soft reset from CMSR RST bit

If the reset source is a power-on reset, the processors's `BUSMODE` pins are read to determine whether boot is via PCI/USB/Sub-ISA or CardBUS interface (Table 4-2).

Table 4-2. ADSP-2192-12 DSP Bus Modes

Bus Type	BUS Mode Pin 0	BUS Mode Pin 1	SCFG:BUS(1:0) Register Field (Bits 1:10)
PCI or Mini-PCI	GND	GND	`00`
CardBUS PC-Card	GND	Open	`01`
Sub-ISA	Open	GND	`10`
USB Serial Bus	Open	Open	`11`

Once the bus configurations have been determined (assuming that a serial EEPROM exists), the boot kernel calls a function to commence reading data from the serial EEPROM. Data format of serial EEPROM boot stream is described in the "ADSP-2192 DSP RTBL" on page 4-4.

Once the kernel has finished reading data from the serial EEPROM, it proceeds to set up and commit bus configurations for the rest of booting via the PCI or USB interfaces. For PCI, the configuration registers are set to be read-only, and the DSP is to respond to PCI requests from the system host. For USB, the DSP is to enter an idle loop, allowing the system host to detect and configure the part.

The final task performed by the kernel after configuring the bus and transferring control to PCI or USB is to enter an infinite loop, waiting for instructions. A predefined memory address, DM `0x000000`, is regularly checked for commands. Once the PCI or USB device has completed booting the DSP, they can write an instruction to this predefined location and have the DSP execute any of supported commands.

Refer to the datasheet, *ADSP-219x/2192 DSP Hardware Reference* and *Application Note EE-124* for further details.

ADSP-2192 DSP RTBL

The VisualDSP++ linker produces files with `.DXE`, `.OVL`, and `.SM` extensions. Typically, these file are not shipped with your applications. Instead, the linker's output is run through the loader that repackages the linker output as an `.H` file, which is consumed by an RTBL, as illustrated in Figure 4-1. The RTBL output file (`.EXE`) is used by a bootable device (PCI or USB).

Given that booting from PCI or USB involves code running on the PC host, you must create a program, which executes on the host, to initiate and conduct the actual PCI or USB transfer. Because of this, the loader

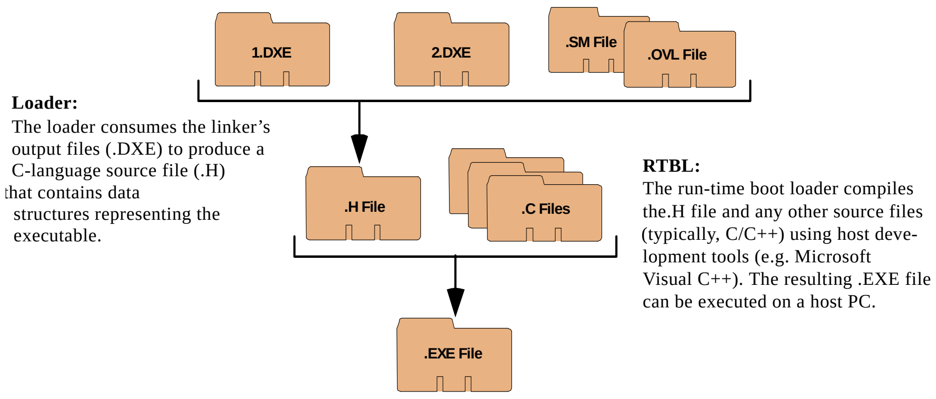

1.DXE
2.DXE
.SM File
.OVL File
Loader:
The loader consumes the linker's output files (.DXE) to produce a C-language source file (.H) that contains data structures representing the executable.
.H File
.C Files
RTBL:
The run-time boot loader compiles the.H file and any other source files (typically, C/C++) using host development tools (e.g. Microsoft Visual C++). The resulting .EXE file can be executed on a host PC.
.EXE File

Figure 4-1. ADSP-2192-12 DSP Loader Sequence

output file (`.H`) is the C-language source code for inclusion and compilation into a host program (`.EXE`) using a C compiler (such as Microsoft Visual C++) to create the RTBL.

The source code emitted by the loader is essentially arrays and structures representing the executable's sections and their contents. Refer to the comments contained within the generated `.H` file for specific information about the data structures and their usage.

Building .DXE Files

You must build two `.DXE` files, which serve as input to the loader. Then you build the `.H` file. Lastly, you create the `.EXE` file (see "Creating a .EXE File" on page 4-6). The following procedure suggests one method to build the `.DXE` file using the VisualDSP++ environment. You may choose to combine steps or use the loader's command-line, instead.

To build the `.DXE` files from VisualDSP++:

1. Open the **Project** page of the **Project Options** dialog box.
2. Under **Processor**, select **ADSP-2192-12**.

3. Under **Type**, select **DSP Loader file**.
4. Under **Name**, type a name for the DSP's core 0 `.DXE` file.
5. From the **Link** page, configure linker options for the core 0 file.
6. Run the project. This generates the `.DXE`, `.OVL`, and `.SM` files.
7. From the **Project** page, under **Name**, type a name for the DSP's core 1 `.DXE` file.
8. From the **Link** page, configure linker options for the core 1 file.
9. Run the project. This generates the `.DXE`, `.OVL`, and `.SM` files.

For more information about the VisualDSP++ IDDE, see the *VisualDSP++ 3.5 User's Guide for 16-bit Processors* or online Help.

If a DSP executable file changes, rerun the loader. The rerun creates a new `.H` file from the `.DXE`, `.OVL`, and `.SM` input files. Then run the RTBL, as described in "Creating a .EXE File" on page 4-6, to build an `.EXE` file from the `.H` file. Automate these tasks from the VisualDSP++ environment by specifying the target type as **DSP Loader file** on the **Project** page of the **Project Options** dialog box and running the RTBL with a post-build command (**Post Build** page of the **Project Options** dialog box); this invokes a makefile that builds the RTBL.

Creating a .EXE File

The loader generates a single output file with an `.H` extension. As described in "Building .DXE Files" on page 4-5, a host compiler inputs the loader output (`.H`) and other user-written code (`.C` or `.CPP`) to create a host executable (`.EXE`).

To create an `.EXE` file from VisualDSP++

1. From the **Load** page of the **Project Options** dialog box, specify the input (`.DXE`) files under **Core 0** and **Core 1**.
2. From the **Post Build** page, configure one or more command lines to execute the RTBL.
3. Run the project. This generates the `.H` file and creates the `.EXE` file.

The `.EXE` file, which is executed on the end-user system, traverses the data structures (`.H`) created by the loader and subsequently compiled by the RTBL. The content of these data structures are downloaded to the ADSP-2192-12 DSP through a user-provided Windows driver.

ⓘ A program running in virtual memory space (as do all Windows applications) cannot access the PCI-mapped memory of the ADSP-2192 DSP directly. A driver is mandatory.

Reference RTBL

Creating the RTBL and driver can be a complex task, especially the first time. To facilitate the process, VisualDSP++ includes a reference RTBL in the form of a Microsoft VisualC++ 6.0 project named `reference_rtbl.dsp`. This project is located in the `ldr` subdirectory of your VisualDSP++ installation directory. Refer to the files in the project for information on the operation of the RTBL and the provided driver interface.

The RTBL downloads the code via the PCI bus to the ADSP-2192-12 EZ-KIT Lite evaluation system through the EZ-KIT Lite's PCI driver. This reference can serve as an example for traversing and using the data structures emitted by the loader. Then download those structures to the target through a driver.

ⓘ The reference RTBL provided with the EZ-KIT Lite evaluation system does not support USB booting.

The EZ-KIT Lite evaluation system driver can be reused for targets other than Analog Devices EZ-KIT Lite evaluation systems, but this simple driver may be inadequate for anything other than prototyping. Furthermore, the EZ-KIT Lite driver can be reused only on so-called "plug-and-play" Windows operating systems like Windows 98 and Windows 2000.

ADSP-2192 DSP RBTL and Overlays

Using overlays in an executable can greatly complicate the use of the RTBL and PCI driver for two main reasons:

- The overlay's "live space" is on the PC host and not the DSP. As a result, the linker is not aware of these addresses at build time. The PCI driver must patch executables as they are downloaded to the DSP to insert the correct addresses at runtime.
- The DSP cannot access the Windows virtual memory space. An overlay must be copied from virtual memory space to PCI memory space, which can only be allocated in limited quantities by the PCI driver.

However, the loader output considers the need for run-time patches of the executable. A portion of the data structure created by the loader is for the express purpose of enabling user code in the RTBL to set up these addresses at runtime. Refer to the reference RTBL for an example of how this is done.

Due to the DSP's inability to access Windows virtual memory space, the RTBL and driver must be coordinated to make overlays available in PCI memory space. Typically, the overlay's image and size information is sent to the driver. The driver `malloc()`'s a memory buffer equal in size to the overlay and then copies the overlay to this space, thus making the overlay available in the PCI memory space.

Using Overlay Symbols

The loader utility, when running, searches for the `OvlPciAdrTbl` and `OvlMgrTbl` symbols. If the loader cannot resolve the two symbols, it sets both values to zero. You must declare one or both symbols in the assembly source file to make the run-time loader handle the overlay properly.

The `OvlPciAdrTbl` symbol holds the start address of the overlay live address table. The loader gets this symbol's value from the input `.DXE` file and places it in the "offset" of the first `relocation_type` array. The value of the "offset" of the second `relocation_type` array is then the value of the first `relocation_type` "offset" plus 2.

Consequently, each subsequent "offset" gets a value equal to the previous "offset" plus 2. The run-time loader determines the exact the overlay live address of each overlay according to the provided "offset" value.

Instead of defining `OvlPciAdrTbl`, define `OvlMgrTbl` in the assembly source code. This symbol should contain the start address of the overlay table, and the overlay table can be declared and defined in your assembly source code. The loader gets this symbol's value and makes the `#define` statement along with the other `#define` statement; for example:

```
#define CORE0_OVL_MGR_TBL   0

#define CORE0_OVL_COUNT     3
```

The first `#define` statement provides the start address of the overlay table. This is zero when the loader fails to find the symbol in the input `.DXE` file. Correct the value by manually changing the value or by defining it in the assembly source code and re-building the loader file.

The second `#define` statement provides the number of overlay for core 0. The run-time loader later uses the provided overlay table start address to find the entry for the live start address each overlay and changes it before loading the overlay table into DSP memory.

ADSP-2192 DSP Loader Guide

This section provides reference information on the ADSP-2192 loader's command-line interface.

When using the loader from within VisualDSP++, settings on the Load page of the Project Options dialog box correspond to the loader's command-line switches. For more information, see the *VisualDSP++ 3.5 User's Guide for 16-bit Processors* or online Help.

A list of switches and a description of each appear in "Loader Command-Line Switches" on page 4-13.

Single-Processor Command Line

Use the following syntax for the loader's command line when there is only one input executable (`.DXE`).

```
elfloader -core0 sourcefile [-o outputfile] -proc ADSP-2192 [-switch…]
```

or

```
elfloader -core1 sourcefile [-o outputfile] -proc ADSP-2192 [-switch…]
```

where:

- `-core0`, `-core1`—Specify that the sourcefile is for `core0` or `core1`, respectively. The loader makes up the array and structure names according to the supplied core number.
- *`sourcefile`*—Identifies the input executable file (`.DXE`) to process. A file name can include the drive and directory; enclose long file name within straight-quotes, "long file name". Before running the loader, ensure that all `.OVL` and `.SM` files reside in the same working directory as the executable. The loader automatically opens the overlay and shared memory files to read in the data while processing the executables.

- `-o` *`outputfile`*—Optional name of the loader's output, a C-language header file (`.H`).
- `-proc ADSP-2192`—The mandatory switch directs the loader to produce an output file for the ADSP-2192 processor.
- *`-switch…`*—One or more optional switches to pass to the loader. Command-line switches may be placed in any order.

Two-Processor Command Line

Use the following syntax for the loader's command line when there are two input executables (`.DXE`).

```
elfloader -core0 sourcefile0 -core1 sourcefile1 -proc ADSP-2192
[-switch…]
```

where:

- `-core0 sourcefile0`—Identifies the `sourcefile0` as the input file to process for core 0. The loader creates the array and structure names according to the supplied core number. Before running the loader, ensure that all `.OVL` and `.SM` files reside in the same working directory as the executables. The loader automatically opens the overlay and shared memory files to read in the data while processing the executables.
- `-core1 sourcefile1`—Identifies the `sourcefile1` as the input file to process for core 1. The loader utility creates the array and structure names according to the supplied core number.
- `-proc ADSP-2192`—The mandatory switch produces an output file for the ADSP-2192-12 processor. By default, the output file is a C-language header file (`.H`).
- *`-switch…`*—One or more optional switches to pass to the loader. Command-line switches may be placed in any order.

Example

```
elfloader -proc ADSP-2192 -core0 p0.dxe -core1 p1.dxe
```

This command line runs the loader utility with:

- `-proc ADSP-2192`—Directs the loader to produce an output file for the ADSP-2192-12 processor.
- `-core0 p0.dxe`—Identifies the input file (`p0.dxe`) to be processed for core 0.
- `-core1 p1.dxe`—Identifies the input file (`p1.dxe`) to be processed for core 1.
- By default, since no output file is specified, the default output file is named `p0.h`.

File Searches

Many loader switches take a file name as an optional parameter. Table 4-4 on page 4-13 lists types the loader expect on files. File searches are important in the loader operation. The loader supports relative and absolute directory names, default directories. File searches occur as follows.

- Specified path—If you include relative or absolute path information in a file name, the loader searches only in that location for the file.
- Default directory—If you do not include path information in the file name, the loader searches for the file in the current working directory.

When you provide an input or output file name as a command-line parameter, use the following guidelines.

- Enclose long file names within straight-quotes, “`long file name`“.
- Append the appropriate file extension to each file.

File Extensions

Table 4-3 lists and describes file types input and output by the loader.

Table 4-3. ADSP-2192 DSP Loader File Extensions

File Extension	Description
`.DXE`	Executable files.
`.OVL`	Overlay memory files. The loader recognizes overlay memory files but does not expect these files on the command line. Place `.OVL` files in the same directory as the `.DXE` file that refers to them; the loader can locate them when processing the `.DXE` file.
`.SM`	Shared memory files. The loader recognizes shared memory files but does not expect these files on the command line. Place `.SM` files in the same directory as the `.DXE` file that refers to them; the loader can locate them when processing the `.DXE` file.
`.H`	Loader output files, C-language header files.

Loader Command-Line Switches

Table 4-4 lists and describes the loader switches.

Table 4-4. ADSP-2192 DSP Loader Command-Line Switches

Switch	Description
`-f format`	Specifies the boot file format. Prepares an output file in the specified format. Currently, the loader utility supports the C-style header file (`.H`) only. This is the default.

`-h`
or
`-help`
	Invokes the command-line help, outputs a list of command-line switches to standard output, and exits. By default, the `-h` switch alone provides help for the loader driver. To obtain a help screen for the ADSP-2192 processor, type `-loader proc ADSP-2192 -h`.

`-M`	Generates make dependencies only.
`-MM`	Shows dependencies while processing the files.

Table 4-4. ADSP-2192 DSP Loader Command-Line Switches (Cont'd)

Switch	Description
`-Mo` *`filename`*	Writes make dependencies to the named file. The `-Mo` option is for use with either the `-M` or -`MM` option. If `-Mo` is not present, the default is a `<stdout>` display.
`-Mt` *`targetname`*	Specifies the make dependencies target name. The `-Mt` option is for use with either the `-M` or -`MM` option. If `-Mt` is not present, the default is the name of the input file with the `.LDR` extension.
`-o` *`file`*	Specifies the name of the output file.

`-proc ADSP-2192`
or
`-dADSP2192`
	Produces an output file for the ADSP-2192-12 processor.
	Note: `-proc ADSP-2192` is the preferred form. The `-dADSP2192` switch is for legacy support only.

`-si-revision` *`version`*	Provides a silicon revision of the specified processor. The *`version`* parameter represents a silicon revision of the processor specified by the `-proc` switch. The revision version takes one of two forms: • One or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: `0.0`; `1.12`; `23.1`. Version `0.1` is distinct from and "lower" than version `0.10`. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal `255`. • A `none` version value is also supported, indicating that the VDSP++ tool should ignore silicon errata. This switch either generates a warning about any potential anomalous conditions or generates an error if any anomalous conditions occur. Note: In the absence of the silicon revision switch, the loader selects the greatest silicon revision it is aware of, if any. Note: In the absence of the *`version`* parameter (a valid version value)—`-si-revision` alone or with an invalid value—the loader generates an error.
`-v`	Outputs verbose loader messages and status information as the loader processes files.

5 ADSP-218X DSP LOADER/SPLITTER

This chapter explains how the loader/splitter program (`elfspl21.exe`) is used to convert executable files into boot-loadable or non-bootable files for ADSP-218x DSPs.

Refer to “Introduction” on page 1-1 for the loader/splitter overview; the introductory material applies to all processor families. Loader and splitter operations specific to ADSP-218x DSPs are detailed in the following sections.

- “ADSP-218x DSP Loader Guide” on page 5-1

  Explains how a boot-loadable file is created, written to, and run from an ADSP-218x DSP’s internal memory.

- “ADSP-218x DSP Splitter Guide” on page 5-15

  Explains how a non-bootable PROM image file is created and executed from an ADSP-218x DSP’s external memory.

ADSP-218x DSP Loader Guide

The loader/splitter (`elfspl21.exe`) processes an executable file (`.DXE`), producing a boot-loadable (`.LDR`) or non-bootable file (`.BNL`, `.BNM`, or `.BNU`). The preparation of a non-bootable image is also called splitting (or PROM splitting). In most cases, developers working with ADSP-218x DSPs use the loader instead of the splitter.

Loader operations depend on loader options, which control how the loader processes executable files, letting you select features such as loader kernel, boot type, and output file format. These options appear on the loader's command line or the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment.

Option setting on the **Load** page correspond to switches displayed on the command line.

To generate a boot-loadable file, you can specify the loader options from within the VisualDSP++ environment. VisualDSP++ invokes the `elfspl21` and builds the output file. To generate a non-bootable PROM file, you must run the `elfspl21` utility from a command-line.

To ensure correct operation of the loader, familiarize yourself with:

- "Boot Modes" on page 5-2
- "Determining Boot Modes" on page 5-4
- "EPROM Booting (BDMA)" on page 5-6
- "Host Booting (IDMA)" on page 5-11
- "No Booting" on page 5-13

Boot Modes

A fully debugged program can be automatically downloaded to the processor after power-up or after a software reset. The way the loader creates a boot-loadable file depends upon how your program is booted into the DSP (booting mode).

ADSP-218x DSPs support the following boot modes.

- EPROM booting (BDMA)
- Host booting (IDMA)
- No boot

EPROM Booting – BDMA

In this mode, project data is stored in an 8-bit-wide PROM. After reset, the DSP follows a special bootstrapping scenario. The DSP reads the PROM's contents through the BDMA interface and initializes on-chip and off-chip memories. The `elfspl21` loader utility generates a PROM image that contains all project data and loader code.

Refer to "EPROM Booting (BDMA)" on page 5-6 for a detailed discussion of this mode.

Host Booting – IDMA Mode

In this mode, the DSP does not start program execution immediately but waits passively until a host DSP (such as a microcontroller or another ADSP-218x part) writes project data into the DSP's on-chip memory through the IDMA interface. The `elfspl21` loader processes the project data, but the data may require post-processing because each type of host processor requires its individual data format.

Refer to "Host Booting (IDMA)" on page 5-11 for a detailed discussion of this mode.

No Boot

In this mode, the DSP does not perform booting. After a reset, the DSP starts program execution directly from the off-chip 24-bit PROM memory. The splitter capabilities of the `elfspl21` generate a proper PROM hex file. This option is not often used. You must run `elfspl21.exe` from a command line.

Refer to "No Booting" on page 5-13 for a detailed discussion of this mode.

Determining Boot Modes

To determine the boot mode, an ADSP-218x DSP samples its mode pins (input flag pins) after reset. Table 5-1 and Table 5-2 explain how to configure various DSPs by pulling the proper pins up or down.

Table 5-1. Boot Modes: ADSP-2181 and ADSP-2183 DSPs

MMAP Pin	BMODE Pin	Description
0	0	BDMA is used in default mode to load the first 32 program memory words from byte memory space. Program execution is held off until all 32 words have been loaded.
0	1	IDMA is used to load any internal memory as desired. Program execution is held off until internal program memory location 0 is written to.
1	X	Bootstrap is disabled. Program execution immediately starts from location 0.

Table 5-2. Boot Modes: ADSP-2184 to ADSP-2189 DSPs

Mode D	Mode C	Mode B	Mode A	Description
X	0	0	0	BDMA is used to load the first 32 program memory words from byte memory space. Program execution is held off until all 32 words have been loaded. The chip is configured in Full Memory Mode.[1]
X	0	1	0	No automatic boot operations occur. Program execution starts at external memory location 0. The chip is configured in Full Memory Mode. BDMA can still be used, but the processor does not automatically use or wait for these operations.

Table 5-2. Boot Modes: ADSP-2184 to ADSP-2189 DSPs (Cont'd)

Mode D	Mode C	Mode B	Mode A	Description
0	1	0	0	BDMA is used to load the first 32 program memory words from the byte memory space. Program execution is held off until all 32 words have been loaded. Chip is configured in Host mode. `/IACK` has active pull-down.[1] (Note: Requires additional hardware.)
0	1	0	1	IDMA is used to load any internal memory as desired. Program execution is held off until the host writes to internal program memory location 0. Chip is configured in Host mode. `/IACK` has active pull-down.[1]
1	1	0	0	BDMA is used to load the first 32 program memory words from byte memory space. Program execution is held off until all 32 words have been loaded. Chip is configured in Host mode; `/IACK` requires external pull down. (Note: Requires additional hardware.)
1	1	0	1	IDMA is used to load any internal memory as desired. Program execution is held off until the host writes to internal program memory location 0. The chip is configured in Host mode. `/IACK` requires external pull-down.[1]

1 Considered as standard operating settings. Using these configurations allows easier design and better memory management.

Table 5-3 lists DSPs that support Mode D operation (`Mode D` pin).

Table 5-3. ADSP-218x DSPs Supporting Mode D Operation

ADSP-2184N	ADSP-2185M	ADSP-2185N
ADSP-2186M	ADSP-2186M	ADSP-2187L
ADSP-2187N	ADSP-2188M	ADSP-2188N
ADSP-2189M	ADSP-2189N	

EPROM Booting (BDMA)

To generate a PROM image for EPROM booting, invoke the `elfspl21.exe` loader from the command line or change the VisualDSP++ project type to Splitter file and specify options on the Project Options dialog box's Load page. The command-line syntax is discussed on page 5-7. For information on how to set up the loader options from within the VisualDSP++, see online Help.

After `RESET`, the ADSP-218x DSP loads the 96 bytes from PROM address `0x0000` into the first 32 locations of on-chip PM memory. Assuming these 96 bytes consist of 32 valid instructions, the DSP executes this piece of program (preloader) afterwards. Usually 32 instructions are not sufficient to load the complete project data; therefore, bootstrapping continues and the preloader loads a set of so-called page loaders beginning at PM address `0x0020`. After the preloader terminates, the DSP executes the page loaders, which load the project data page by page.

The loader uses a default preloader. You can force the loader with the `-uload` switch to use a customized preloader to reduce wait states or to implement a boot management scenario (discussed in detail in the *Application Note EE-146*).

Example

The following example lists the default preloader together with its opcode. Note that the `BWCOUNT` value is generated dynamically.

```
/* standard preloader (32 instructions)   address  opcodes */
ax0 = 0x0060; dm(0x3fe2) = ax0;  /* BEAD  0x0000:400600 93FE20 */
ax0 = 0x0020; dm(0x3fe1) = ax0;  /* BIAD  0x0002:400200 93FE10 */
ax0 = 0x0000; dm(0x3fe3) = ax0;  /* CTRL  0x0004:400000 93FE30 */
ax0 = 0x0087; dm(0x3fe4) = ax0;  /* BWCOUNT0x0006: 400870 93FE40 */
ifc = 0x0008;nop;                /* BDMA IRQ 0x0008: 3C008C 000000 */
imask = 0x0008;                       /* 0x000A: 3C0083 */
idle;                                 /* 0x000B: 028000 */
jump 0x0020; nop; nop; nop;           /* 0x000C: 18020F 000000 */
nop;         nop; nop; nop;           /* 0x0010: 000000 000000 */
```

```
nop;          nop; nop; nop;          /* 0x0014: 000000 000000 */
nop;          nop; nop; nop;          /* 0x0018: 000000 000000 */
rti;          nop; nop; nop;          /* 0x001C: 0A001F 000000 */
```

The page loaders are generated dynamically by the `elfspl21`, depending on the number of overlay pages to be initialized. A page loader sets up BDMA transfers. Since BDMA transfers cannot target off-chip DM and PM memories (overlay pages 1 and 2) directly, the corresponding page loaders first BDMA-load the data into on-chip memory and then copy the data by core instructions.

The final BDMA sequence has the BCR bit set. It overwrites the preloader and page loaders resident in internal PM memory and issues a context reset once it has finished. The program counter resets to `0x0000` and program execution begins.

Refer to the *ADSP-218x DSP Hardware Reference* for a detailed description of BDMA capabilities. You can debug the EPROM booting process using the VisualDSP++ simulator by loading the `.BNM` file (**Settings->Simulator**) and then resetting (**Debug->Reset**) the DSP to start booting.

The loader outputs files in industry-standard file formats, Intel hex-32 and Motorola S, which are readable by most EPROM burners. For advanced usage, other file formats are supported; refer to Appendix A, "File Formats" for details. The default file extension is `.BNM`.

ADSP-218x BDMA Loader Command-Line Reference

This section details the ADSP-218x BDMA loader's command-line interface.

The syntax for the loader's command line is:

```
elfspl21 sourcefile [outputfile] -218{x|1} [-switch …]
```

where:

- *`sourcefile`*—Identifies the executable file (`.DXE`) to process into a single-processor boot-loadable file. A file name can include the drive and directory. Enclose long file names within straight-quotes, `“long file name“`.

- *`outputfile`*—Specifies the output file. The optional parameter can include the drive, directory, file name, and file extension (`.BNM`).

- *`-switch…`*—Optional switches to process. The loader provides many switches to select operations and modes (see Table 5-5 on page 5-10).

- `-218{x|1}`—Specifies the target processor. Always specify either `-2181` or `-218x` when working with an ADSP-218x DSP (see Table 5-5 on page 5-10).

ⓘ The *`sourcefile`* and *`outputfile`* names must be placed first on the command line.Command-line switches may occur in any order, except for the `-2181` (or -`218x`) and `-loader` switches. When required, the `-2181` (or -`218x`) and `-loader` switches follow the *`outputfile`* name; the `-2181` (or -`218x`) switch always precedes the `-loader` switch.

Example

```
elfspl21 p0.dxe output.bnm -218x -loader -i -uload test.doj
```

In the above example, the ADSP-218x BDMA loader runs with:

- `p0.dxe`—the name of the executable file to process into a boot-loadable file.

- `output.bnm`—the name of the loader output file.

- `-218x`—the target processor, one of the ADSP-2184 through ADSP-2189 DSPs.

- `-loader`—EPROM booting as the boot mode for the boot-loadable file.
- `-i`—Intel hex-32 format for the boot-loadable file.
- `-uload`—`test.doj` as the boot kernel for the boot-loadable file.

File Searches

Many loader switches take a file name as an optional parameter. Table 5-4 on page 5-9 lists the expected file types. File searches are important in the loader operation. The loader supports relative and absolute directory names, default directories. File searches occur as described on page 1-9.

File Extensions

Table 5-4 lists and describes file types input and output by the loader.

Table 5-4. ADSP-218x Loader File Extensions

File Extension	Description
`.DXE`	Executable files and boot-kernel files.
`.OVL`	Overlay memory files. The loader recognizes overlay memory files but does not expect these files on the command line. Place `.OVL` files in the same directory as the `.DXE` file that refers to them; the loader can locate them when processing the `.BNM` file.
`.BNM`	Loader output file for EPROMs.
`.IDM`	Loader output file for IDMA.

Loader Switches

Table 5-5 lists and describes the loader switches used in BDMA mode.

Table 5-5. ADSP-218x DSP: BDMA Mode Command-Line Switches

Switch	Description
sourcefile	Specifies the executable file (`.DXE`) to be processed for a single-processor boot-loadable file.
outputfile	Specifies the loader's output file (`.BNM`).
`-h`	Outputs the list of command-line switches to standard output and exits.
`-i`	Produces Intel hex format.
`-s`	Produces Motorola S2 format.
`-byte`	Produces byte-stream output format.
`-218{x|1}`	Specifies the target processor: • `-2181`—ADSP-2181 or ADSP-2183 DSP • `-218x`—one of the ADSP-2184 through ADSP-2189 DSP. When used with `-loader`, keeps the image.
`-218{4|5|6|8|9}`	Use in place of `-218x`. Specifies the ADSP-2184, ADSP-2185, ADSP-2186, ADSP-2187, ADSP-2188, or ADSP-2189 DSP as a target processor. Supports `PMOVLAY/DMOVLAY`.
`-loader`	Includes the ADSP-218x default loader.
`-noloader`	Excludes the ADSP-218x loader. When used with `-2181 -loader` (or `-218x-loader`), generates a byte memory image without a loader. This suppresses the preloader and page loaders.
`-bdma` *inputfile start_address*	Use with `-2181 -loader`. Specifies placement of an additional `.DXE` file (*inputfile*) in byte memory, starting at the specified address. The loader returns an error if the specified address is in use by the loader, or by another additional `-bdma` specified file. Files specified this way may be BDMA loaded at runtime, but the individual start addresses, length, and target information must be predefined.
`-bdmaload` *inputfile start_address*	Specifies an additional `.DXE` file (*inputfile*) to be placed in byte memory, starting at the specified address. The address must be a multiple of `0x4000`. Unlike the `-bdma` switch, this option generates a preloader and page loaders for the specified `.DXE` file. This way, two or more applications can be stored in the same EPROM and may replace the default application at runtime.

Table 5-5. ADSP-218x DSP: BDMA Mode Command-Line Switches

Switch	Description
`-uload` *`filename`*	Reads the BDMA preloader from *`filename.doj`*. The preloader must consist of exactly 32 instructions. Preloaders generated by the `elfspl21` utility automatically determine the `BWCOUNT` value, which mirrors the numbers of instructions required by the page loaders. Customized preloaders do not have access to this length information and should always set `BWCOUNT` by assuming the maximal possible page loader length. In this software version, the maximal number of instructions required for the page loader is 658. This may change in future releases.
`-offsetaddr` *`#`*	Provides byte memory address offset (valid only with `-2181 -noloader` or with `-218x -noloader`).
`-offsetpage` *`#`*	Provides byte memory page offset (valid only with `-2181 -noloader` or with `-218x -noloader`).

Host Booting (IDMA)

When booted through the IDMA interface, an ADSP-218x DSP behaves like a slave. After reset, it does not start program execution until internal PM location `0x0000` is overwritten by an IDMA write access. The host processor initializes all on-chip memories of the DSP but finally writes to PM address `0x0000`. Then the DSP starts program execution.

The host is responsible for handling the IDMA traffic properly. Typically, the host boots the DSP page-by-page (segment-by-segment). To boot a segment, the host first performs an address latch cycle to program the IDMA Control register and the IDMA Overlay register (on ADSP-2184 through ADSP-2189 DSPs only). Afterwards, the host writes 16- or 24-bit words to the IDMA port. The DSP auto-increments its address counter.

Figure 5-1 on page 5-12 illustrates the algorithm the host processor must compute to boot the DSP successfully.

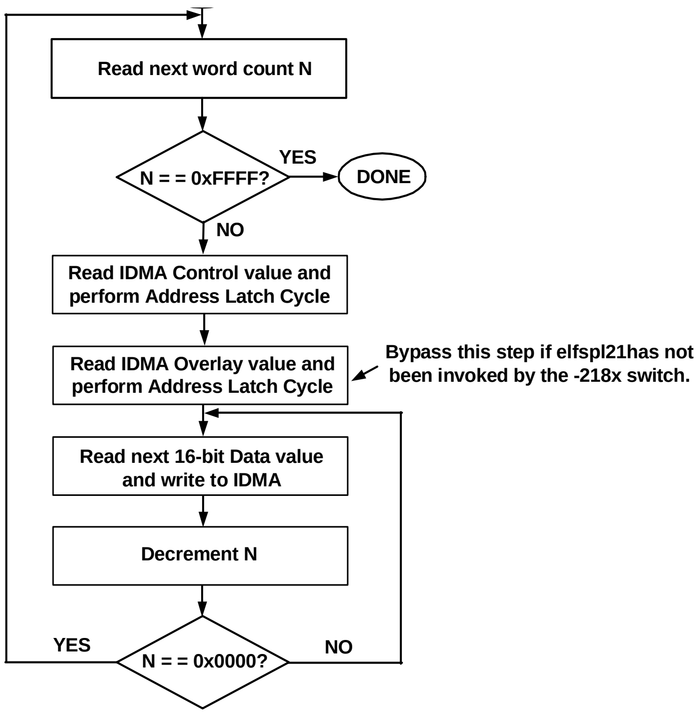

Read next word count N
N = = 0xFFFF?
YES
DONE
NO
Read IDMA Control value and perform Address Latch Cycle
Read IDMA Overlay value and perform Address Latch Cycle
Bypass this step if elfspl21has not been invoked by the -218x switch.
Read next 16-bit Data value and write to IDMA
Decrement N
YES
N = = 0x0000?
NO

Figure 5-1. Host Processor Algorithm

The loader generates an ASCII file (`.IDM`). Every segment data is headed by the following information: word count, IDMA control value, and overlay page number. Since the IDMA interface is a 16-bit interface, the `.IDM` file is organized in 16-bit portions.

Typically, embedded processors cannot directly process ASCII files in `.IDM` format. It is up to the user to post-process this file in a customized way.

Due to hardware restrictions, IDMA booting of off-chip memories is not possible. Refer to the description of IDMA capabilities in the *ADSP-218x DSP Hardware Reference*.

ADSP-218x IDMA Loader Command-Line Reference

To create an IDMA boot file, use the following syntax for the ADSP-218x loader's command line.

```
elfspl21 sourcefile [outputfile] -218{x|1} -idma
```

Table 5-6 lists and describes each switch used in IDMA mode.

Table 5-6. ADSP-218x DSP: IDMA Command-Line Switches

Switch	Description
sourcefile	Specifies the executable file (`.DXE`) to be processed for a single-processor boot-loadable file.
outputfile	Specifies the output file (`.IDM`).
`-218{x|1}`	Specifies the target processor: • `-2181`—ADSP-2181 or ADSP-2183 DSP • `-218x`—one of the ADSP-2184 through ADSP-2189 DSP. When used with `-loader`, keeps the image.
`-218{4|5|6|8|9}`	Use in place of `-218x`. Specifies the ADSP-2184, ADSP-2185, ADSP-2186, ADSP-2187, ADSP-2188, or ADSP-2189 DSP as a target processor. Supports `PMOVLAY/DMOVLAY`.
`-idma`	Forces the loader to create an IDMA boot file. It overwrites most of the BDMA boot-specific options.

No Booting

In very rare cases, applications require that the DSP not be booted after reset, or that the DSP is booted with a ROM connected to the DSP's off-chip DM/PM address space.

Preparing a non-bootable PROM image is called splitting. In most cases, developers working with ADSP-218x DSPs use the loader instead of the splitter. For ADSP-218x DSPs, splitter and loader features are handled by the `elfspl21.exe`. The splitter must be invoked by a completely different set of command-line switches. Refer to the following "ADSP-218x DSP Splitter Guide" for more information.

ADSP-218x DSP Splitter Guide

Use the splitter (`elfspl21.exe`) to process an executable file to produce a non-bootable, PROM-image file. In most cases, developers working with ADSP-21xx DSPs use the loader instead of the splitter.

This section contains the following information on the splitter.

- "Using Splitter" on page 5-15

  Describes how to use the splitter from a command line.

- "ADSP-218x Splitter Command-Line Reference" on page 5-16

  Summarizes and describes the splitter command-line switches.

Using Splitter

You must run the splitter (PROM splitter) from a command line. You cannot generate a non-bootable PROM file from within the VisualDSP++ environment. To automate the process, specify the splitter command-line within VisualDSP++ from the **Post Build** page of the **Project Options** dialog box.

The ADSP-218x splitter generates images for external `PMOVLAY` (1 and 2) and `DMOVLAY` (1 and 2) memory pages. If you use the splitter to produce ROM images (for example, the ADSP-2181DSP's program memory pages 1 and 2), the generated code must target ROM. Define the appropriate ROM segments in the `.LDF` file.

Splitter options control how the splitter processes executable files, letting you select features such as memory type and file format.

ADSP-218x Splitter Command-Line Reference

The splitter (`elfspl21.exe`) generates non-bootable, PROM-image files for ADSP-218x DSPs by processing executable files (`.DXE`).

The splitter command line is case sensitive.

Run the splitter from the command line using the following syntax.

```
elfspl21 sourcefile [outputfile] {-pm &| -dm} [-switch …]
```

where:

- *`sourcefile`*—Name of the executable file (`.DXE`) to be processed for a non-bootable, PROM-image file. A file name can include the drive and directory. Enclose long file names within straight-quotes, `“long file name”`.
- *`outputfile`*—Optional name of the splitter's output, a PROM file with the `.BNL`, `.BNU`, or `.BNM` file extension.
- *`-switch …`*—One or more optional switches to process. Switches select operations and modes for the splitter.
- `-pm &| -dm`—Indicates that either `-pm` or `-dm` or both can be used.

Example

The following two command lines run the splitter twice, first producing PROM files for program memory and then producing PROM files for data memory.

```
elfspl21 my_proj.dxe pm_stuff -pm
elfspl21 my_proj.dxe dm_stuff -dm
```

The switches on these command lines are as follows.

`-pm`, `-dm`

Select program memory or data memory as the source in the executable for extraction and placement into the image. Because these are the only switches used to identify the memory source, the sources are any PM or DM ROM memory segments. Because no other contents switches appear on these command lines, the format for the output defaults to Motorola S3 format, and the output PROM width defaults to 8 bits for all PROMs.

`pm_stuff`, `dm_stuff`

Specify names for the output files without file extension. Use different names so the output of the second run does not overwrite the output of the first run. The output names are `pm_stuff.s_#` and `dm_stuff.s_#`.

`my_proj.dxe`

Specify an executable file to process into a non-bootable PROM-image file.

File Searches

Many splitter switches take a file name as an optional parameter. Table 5-7 lists the type of files, names, and extensions that the splitter expects on files. File searches are important in the splitter's process. The splitter supports relative and absolute directory names, default directories, and user-selected directories for file search paths.

When you provide an input or output file name as a command-line parameter, use the guidelines stated on page 1-9.

Splitter File Extensions

Table 5-7 lists and describes file types input and output by the splitter.

Table 5-7. Splitter File Name Extensions

File Extension	File Description
`.DXE`	Executable files and boot-kernel files.
`.BNU`	Splitter binary output file—upper.
`.BNM`	Splitter binary output file—middle.
`.BNL`	Splitter binary output file—lower.

Splitter Switches

Table 5-8 lists and describes the available splitter command-line switches.

Table 5-8. Splitter Command-Line Switches

Switch	Description
`sourcefile`	Specifies the source (`.DXE`) for the splitter operation.
`outputfile`	Specifies the splitter's output file. If not specified, the name of the *`sourcefile`* (executable file) is used for the output. The extension depends on the output format.
`-byte`	Produces byte-stream output format.
`-dm`	Extracts data memory. Extracts segments from the executable declared as data memory. The splitter generates two one-byte files: `.BNM`—contains the upper bytes of the 16-bit data words `.BNL`—contains the lower bytes.
`-i`	Produces Intel hex output format.
`-pm`	Extracts program memory. Extracts segments from the executable declared as program memory. The splitter generates three one-byte files: `.BNU`—contains the upper bytes of the 24-bit words `.BNM`—contains the middle bytes `.BNL`—contains the lowest bytes.
`-readall`	Includes RAM and ROM in PROM. Extracts both RAM and ROM segments from the input file. By default, only ROM segments are extracted.

Table 5-8. Splitter Command-Line Switches (Cont'd)

Switch	Description
`-s`	Produces Motorola S1 output format

`-us`
`-us2`
`-ui`
	Produces a byte-stacked format file for 8-bit memory:
	`-us` yields Motorola S1 output format
	`-us2` yields Motorola S2 output format
	`-ui` yields Intel hex output format.

A FILE FORMATS

VisualDSP++ development tools support many file formats, in some cases several for each development tool. This appendix describes file formats that are prepared as inputs and produced as outputs.

The appendix describes three types of files:

- "Source Files" on page A-2
- "Build Files" on page A-5
- "Debugger Files" on page A-9

Most of the development tools use industry-standard file formats. These formats are described in "Format References" on page A-10.

Source Files

This section describes the following input file formats:

- "C/C++ Source Files" on page A-2
- "Assembly Source Files" on page A-3
- "Assembly Initialization Data Files" on page A-3
- "Header Files" on page A-4
- "Linker Description Files" on page A-4
- "Linker Command-Line Files" on page A-5

C/C++ Source Files

C/C++ source files are text files (`.C`, `.CPP`, `.CXX`, and so) containing C/C++ code, compiler directives, possibly a mixture of assembly code and directives, and, typically, preprocessor commands.

Several dialects of C code are supported: pure (portable) ANSI C, and at least two subtypes[1] of ANSI C with ADI extensions. These extensions include memory type designations for certain data objects, and segment directives used by the linker to structure and place executable files.

The C/C++ compiler, run-time library, as well as a definition of ADI extensions to ANSI C are detailed in your target processor's *VisualDSP++ 3.5 C/C++ Compiler and Library Manual*.

[1] With and without built-in function support; a minimal differentiator. There are others.

Assembly Source Files

Assembly source files (.ASM) are text files containing assembly instructions, assembler directives, and (optionally) preprocessor commands. For information on assembly instructions, see your processor's *Programming Reference*.

The processor's instruction set is supplemented with assembly directives. Preprocessor commands control macro processing and conditional assembly or compilation.

For information on the assembler and preprocessor, see the *VisualDSP++ 3.5 Assembler and Preprocessor Manual.*

Assembly Initialization Data Files

Assembly initialization data files (.DAT) are text files that contain fixed- or floating-point data. These files provide initialization data for an assembler .VAR directive or serve in other tool operations.

When a .VAR directive uses a .DAT file for data initialization, the assembler reads the data file and initializes the buffer in the output object file (.DOJ). Data files have one data value per line and may have any number of lines.

The .DAT extension is explanatory or mnemonic. A directive to #include <file> can take any file name and extension as an argument.

Fixed-point values (integers) in data files may be signed, and they may be decimal-, hexadecimal-, octal-, or binary-base values. The assembler uses the prefix conventions listed in Table A-1 to distinguish between numeric formats.

For all numeric bases, the assembler uses 16-bit words for data storage; 24-bit data is for the program code only. The largest word in the buffer determines the size for all words in the buffer. If you have some 8-bit data in a 16-bit wide buffer, the assembler loads the equivalent 8-bit value into the most significant 8 bits into the 8-bit memory location and zero-fills the lower eight bits.

Table A-1. Numeric Formats

Convention	Description
`0x`*number* `H#`*number* `h#`*number*	Hexadecimal number
number `D#`*number* `d#`*number*	Decimal number
`B#`*number* `b#`*number*	Binary number
`O#`*number* `o#`*number*	Octal number

Header Files

Header files (`.H`) are ASCII text files that contain macros or other preprocessor commands which the preprocessor substitutes into source files. For information on macros and other preprocessor commands, see the *VisualDSP++ 3.5 Assembler and Preprocessor Manual for 16-Bit Processors.*

Linker Description Files

Linker Description Files `.LDF` are ASCII text files that contain commands for the linker in the linker's scripting language. For information on this scripting language, see the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

Linker Command-Line Files

Linker command-line files (`.TXT`) are ASCII text files that contain command-line input for the linker. For more information on the linker command line, see the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

Build Files

Build files are produced by the VisualDSP++ development tools while building a project. This section describes the following build file formats:

- "Assembler Object Files" on page A-5
- "Library Files" on page A-6
- "Linker Output Files" on page A-6
- "Memory Map Files" on page A-7
- "Loader Output Files in Intel Hex-32 Format" on page A-7
- "Splitter Output Files in ASCII Format" on page A-9

Assembler Object Files

Assembler output object files (`.DOJ`) are binary, executable and linkable files (ELF). Object files contain relocatable code and debugging information for a DSP program's memory segments. The linker processes object files into an executable file (`.DXE`). For information on the object file's ELF format, see the "Format References" on page A-10.

Library Files

Library files (`.DLB`), the archiver's output, are binary, executable and linkable files (ELF). Library files (called archive files in previous software releases) contain one or more object files (archive elements).

The linker searches through library files for library members used by the code. For information on the ELF format used for executable files, refer to "Format References" on page A-10.

The archiver automatically converts legacy input objects from COFF to ELF format.

Linker Output Files

The linker's output files (`.DXE`, `.SM`, `.OVL`) are binary, executable and linkable files (ELF). The executable files contain program code and debugging information. The linker fully resolves addresses in executable files. For information on the ELF format used for executable files, see the TIS Committee texts cited in "Format References" on page A-10.

The loaders/splitters are used to convert executable files into boot-loadable or non-bootable files.

Executable files are converted into a boot-loadable file (`.LDR`) for the ADI processors using a loader program. Once an application program is fully debugged, it is ready to be converted into a boot-loadable file. A boot-loadable file is transported into and run from a processor's internal memory. This file is then programmed (burned) into an external memory device within your target system.

A splitter generates non-bootable, PROM-image files by processing executable files and producing an output PROM file. A non-bootable PROM image file executes from DSP external memory.

Memory Map Files

The linker can output memory map files (`.MAP`), which are ASCII text files that contain memory and symbol information for your executable file(s). The map contains a summary of memory defined with `MEMORY{}` commands in the `.LDF` file, and provides a list of the absolute addresses of all symbols.

Loader Output Files in Intel Hex-32 Format

The loader can output Intel hex-32 format files (`.LDR`). The files support 8-bit-wide PROMs and are used with an industry-standard PROM programmer to program memory devices. One file contains data for the whole series of memory chips to be programmed.

The following example shows how the Intel hex-32 format appears in the loader's output file. Each line in the Intel hex-32 file contains an extended linear address record, a data record, or the end-of-file record.

:`020000040000FA`	Extended linear address record
:`0402100000FE03F0F9`	Data record
:`00000001FF`	End-of-file record

Extended linear address records are used because data records have a 4-character (16-bit) address field, but in many cases, the required PROM size is greater than or equal to `0xFFFF` bytes. Extended linear address records specify bits 31–16 for the data records that follow.

Table A-2 shows an example of an extended linear address record.

Table A-3 shows the organization of an example data record.

Table A-4 shows an end-of-file record.

Table A-2. Extended Linear Address Record Example

Field	Purpose
`:020000040000FA`	Example record
:	Start character
`02`	Byte count (always 02)
`0000`	Address (always 0000)
`04`	Record type
`0000`	Offset address
`FA`	Checksum

Table A-3. Data Record Example

Field	Purpose
`:0402100000FE03F0F9`	Example record
:	Start character
`04`	Byte count of this record
`0210`	Address
`00`	Record type
`00`	First data byte
`F0`	Last data byte
`F9`	Checksum

Table A-4. End-of-File Record Example

Field	Purpose
`:00000001FF`	End-of-file record
:	Start character
`00`	Byte count (zero for this record)
`0000`	Address of first byte

Table A-4. End-of-File Record Example (Cont'd)

Field	Purpose
`01`	Record type
`FF`	Checksum

Splitter Output Files in ASCII Format

When the loader is invoked as a splitter, its output can be an ASCII-format file with the `.LDR` extension. ASCII-format files are text representations of ROM memory images that can be post-processed by users. For more information, refer to the chapter in this manual that is appropriate for your target processor.

Debugger Files

Debugger files provide input to the debugger to define support for simulation or emulation of your program. The debugger supports all the executable file types produced by the linker (`.DXE`, `.SM`, `.OVL`). To simulate I/O, the debugger also supports the assembler's data file format (`.DAT`) and the loader's loadable file formats (`.LDR`).

The standard hexadecimal format for a SPORT data file is one integer value per line. Hexadecimal numbers do not require an `0x` prefix. A value can have any number of digits but is read into the SPORT register as follows:

- The hexadecimal number is converted to binary.
- The number of binary bits read in matches the word size set for the SPORT register, which starts reading from the LSB. The SPORT register then fills with zero values shorter than the word size or conversely truncates bits beyond the word size on the MSB end.

In the following example (Table A-5), a SPORT register is set for 20-bit words and the data file contains hexadecimal numbers. The simulator converts the hex numbers to binary and then fills/truncates to match the SPORT word size. The `A5A5` is filled and `123456` is truncated.

Table A-5. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
`A5A5A`	`1010 0101 1010 0101 1010`	`1010 0101 1010 0101 1010`
`FFFF1`	`1111 1111 1111 1111 0001`	`1111 1111 1111 1111 0001`
`A5A5`	`1010 0101 1010 0101`	`0000 1010 0101 1010 0101`
`5A5A5`	`0101 1010 0101 1010 0101`	`0101 1010 0101 1010 0101`
`11111`	`0001 0001 0001 0001 0001`	`0001 0001 0001 0001 0001`
`123456`	`0001 0010 0011 0100 0101 0110`	`0010 0011 0100 0101 0110`

Format References

The following texts define industry-standard file formats supported by VisualDSP++.

- Gircys, G.R. (1988) *Understanding and Using COFF* by O’Reilly & Associates, Newton, MA
- (1993) *Executable and Linkable Format (ELF) V1.1* from the Portable Formats Specification V1.1, Tools Interface Standards (TIS) Committee.

  Go to: `http://developer.intel.com/vtune/tis.htm`.
- (1993) *Debugging Information Format (DWARF) V1.1* from the Portable Formats Specification V1.1, UNIX International, Inc.

  Go to: `http://developer.intel.com/vtune/tis.htm`.

I INDEX