VisualDSP++® 4.0 Loader Manual

Revision 1.0, January 2005

Part Number
82-000420-05

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

ANALOG DEVICES

CONTENTS

LOADER FOR ADSP-21161 SHARC PROCESSORS

PREFACE

Thank you for purchasing VisualDSP++® 4.0, Analog Devices, Inc. development software for digital signal processing (DSP) applications.

Purpose of This Manual

The *VisualDSP++ 4.0 Loader Manual* contains information about the loader/splitter program for the following Analog Devices, Inc. processors—SHARC® (ADSP-21xxx), TigerSHARC® (ADSP-TSxxx), and Blackfin® (ADSP-BFxxx).

The manual describes the loader/splitter operations for these processors and references information about related development software. It also provides information about the loader and splitter command-line interfaces.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe your target architecture.

Manual Contents

The manual contains:

- Chapter 1, "Introduction"
- Chapter 2, "Loader/Splitter for Blackfin Processors"
- Chapter 3, "Loader for ADSP-TSxxx TigerSHARC Processors"
- Chapter 4, "Loader for ADSP-2106x/21160 SHARC Processors"
- Chapter 5, "Loader for ADSP-21161 SHARC Processors"
- Chapter 6, "Loader for ADSP-2126x/2136x SHARC Processors"
- Chapter 7, "Splitter for SHARC and TigerSHARC Processors"
- Appendix A, "File Formats"

What's New in This Manual

Information in this *VisualDSP++ 4.0 Loader Manual* applies to all Analog Devices, Inc. processors listed in "Supported Processors".

Refer to *VisualDSP++ 4.0 Product Release Bulletin* for information on new and updated VisualDSP++ 4.0 features and other product release information.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at `http://www.analog.com/processors/technicalSupport`
- E-mail tools questions to `dsptools.support@analog.com`
- E-mail processor questions to `dsp.support@analog.com`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. 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 following is the list of Analog Devices, Inc. processors supported in VisualDSP++ 4.0.

Blackfin (ADSP-BFxxx) Processors

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

ADSP-BF531	ADSP-BF532 (formerly ADSP-21532)
ADSP-BF533	ADSP-BF534
ADSP-BF535 (formerly ADSP-21535)	ADSP-BF536
ADSP-BF537	ADSP-BF538
ADSP-BF539	ADSP-BF561
ADSP-BF566	AD6532

SHARC (ADSP-21xxx) Processors

The name "*SHARC*" refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC processors.

ADSP-21020	ADSP-21060	ADSP-21061	ADSP-21062
ADSP-21065L	ADSP-21160	ADSP-21161	ADSP-21261
ADSP-21262	ADSP-21266	ADSP-21267	ADSP-21363
ADSP-21364	ADSP-21365	ADSP-21366	ADSP-21367
ADSP-21368	ADSP-21369		

TigerSHARC (ADSP-TSxxx) Processors

The name "*TigerSHARC*" refers to a family of floating-point and fixed-point [8-bit, 16-bit, and 32-bit] processors. VisualDSP++ currently supports the following TigerSHARC processors.

ADSP-TS101	ADSP-TS201	ADSP-TS202	ADSP-TS203

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 Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages 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 a means to select the information you want to receive.

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

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at `www.analog.com/processors`, which provides access to technical publications, data sheets, 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.

- E-mail questions or requests for information to
  `dsp.support@analog.com`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49-89-76903-157 (Europe)
- Access the FTP Web site at
  `ftp ftp.analog.com` (or `ftp 137.71.25.69`)
  `ftp://ftp.analog.com`

Related Documents

For information on product related development software, see these publications:

- *VisualDSP++ 4.0 User's Guide*
- *VisualDSP++ 4.0 Getting Started Guide*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for SHARC Processors*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for TigerSHARC Processors*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors*

- *VisualDSP++ 4.0 Linker and Utilities Manual*
- *VisualDSP++ 4.0 Assembler and Preprocessor Manual*
- *VisualDSP++ 4.0 Product Release Bulletin*
- *VisualDSP++ 4.0 Kernel (VDK) User's Guide*
- *VisualDSP++ 4.0 Quick Installation Reference Card*

For hardware information, refer to your processors's hardware reference, programming reference, or data sheet. All documentation is available online. Most documentation is available in printed form.

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

`http://www.analog.com/processors/resources/technicalLibrary`

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, the Dinkum Abridged C++ library, and Flexible License Manager (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 of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
`.CHM`	Help system files and manuals in Help format
`.HTM` or `.HTML`	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the `.HTML` files requires a browser, such as Internet Explorer 4.0 (or higher).
`.PDF`	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the `.PDF` files requires 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 running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- 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).

Accessing Documentation From Windows

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

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

Using Windows Explorer

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

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.

Accessing Documentation From the Web

Download manuals at the following Web site:
http://www.analog.com/processors/resources/technicalLibrary/manuals

Select a processor 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

To purchase VisualDSP++ manuals, call **1-603-883-2430**. The manuals may be purchased only as a kit.

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

Hardware Tools Manuals

To purchase EZ-KIT Lite™ and In-Circuit Emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Notation Conventions

Text conventions used in this manual are identified and described as follows.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate 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`. One or the other is required.
`[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 ellipse; 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.
ⓘ	**Note:** For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
⚡	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
🚫	**Warning:** Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

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

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 an application program's development flow.

Most of this chapter applies to all 8-, 16-, and 32-bit data processors. Information applicable to a particular target processor, or to a particular processor family, is provided in the following chapters.

- Chapter 2, "Loader/Splitter for Blackfin Processors"
- Chapter 3, "Loader for ADSP-TSxxx TigerSHARC Processors"
- Chapter 4, "Loader for ADSP-2106x/21160 SHARC Processors"
- Chapter 5, "Loader for ADSP-21161 SHARC Processors"
- Chapter 6, "Loader for ADSP-2126x/2136x SHARC Processors"
- Chapter 7, "Splitter for SHARC and TigerSHARC Processors"

The code examples in this manual have been compiled using VisualDSP++ 4.0. The examples compiled with another version of VisualDSP++ may result in build errors or different output; although, the highlighted algorithms stand and should continue to stand in future releases of VisualDSP++.

Program Development Flow

Figure 1-1 is a simplified view of the application development flow.

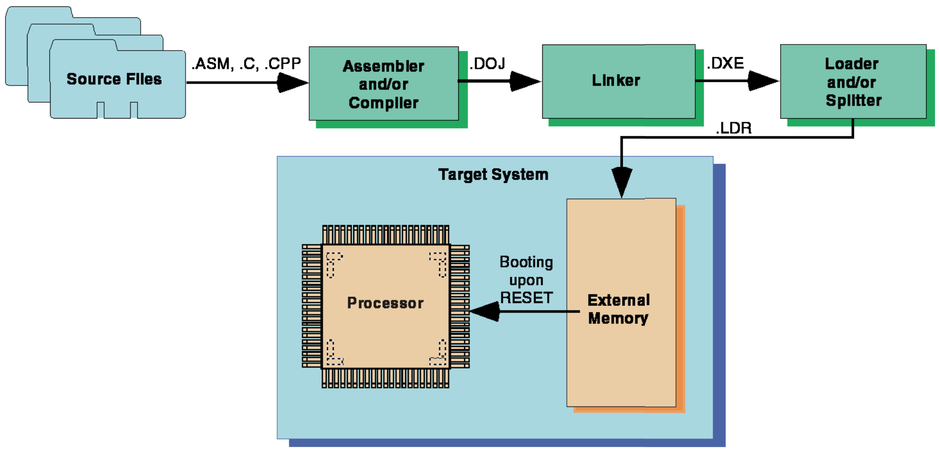


Figure 1-1. Program Development Flow and Booting Sequence

The development flow can be split into three phases:

1. "Compiling and Assembling"
2. "Linking"
3. "Loading, Splitting, or Both"

A brief description of each phase 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++ 4.0 Assembler and Preprocessor Manual* or the VisualDSP++ 4.0 C/C++ Compiler and Library Manual for your processor, and online help for information about the assembler and compiler.

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, the linker also produces the shared memory files and overlay files. The linker output (`.DXE` files) 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 (DWARF-2) used by the debugger.

These executable files are not readable by the processor hardware directly. They are neither supposed to be burned onto an EPROM or flash memory device. Executable files are intended for VisualDSP++ debugging targets, such as the simulator or emulator. Refer to the *VisualDSP++ 4.0 Linker and Utilities Manual* and online Help for information about linking and debugging.

Loading, Splitting, or Both

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

VisualDSP++ 4.0 includes these loader/splitter programs:

- `elfloader.exe` (loader) for Blackfin, TigerSHARC, and SHARC processors. The loader for Blackfin processors acts also as a ROM splitter when invoked with the corresponding option settings or switches.
- `elfspl21k.exe` (splitter) for TigerSHARC and SHARC processors.

The loader/splitter output is either a boot-loadable or non-bootable file. 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 PROM space on an EZ-KIT Lite® board via the Flash Programmer plug-in. Refer to VisualDSP++ Help 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.

Non-bootable Files Versus Boot-loadable Files

A non-bootable file executes from an external memory of the processor, while a boot-loadable file is transported into and executes from an internal memory of the processor. The boot-loadable file is then programmed (burned into EPROM) into an external memory device within your target system. The loader outputs loadable files in formats readable by most

EPROM burners, such as Intel hex-32 and Motorola S formats. For advanced usage, other file formats and boot modes are supported. (See "File Formats" on page A-1.)

A non-bootable EPROM image file executes from an external memory of the processor, bypassing the built-in boot mechanisms. Preparing a non-bootable EPROM image is called *splitting*. In most cases (except for Blackfin processors), developers working with floating- and fixed-point processors use the splitter instead of the loader to produce a non-bootable memory image file.

A booting sequence of the processor and application program design dictate the way loader/splitter program is called to consume and transform executable files:

- For Blackfin processors, splitter and loader operations are handled by the loader program, `elfloader.exe`. The splitter is invoked by a different set of command-line switches than the loader.
- For TigerSHARC and SHARC processors, splitter operations are handled by the splitter program, `elfspl21k.exe`.

Loader Operations

You can run the loader from the IDDE. In order to do so, change the project type from **DSP Executable** to **DSP Loader File**.

Loader operations depend on loader options, which control how the loader processes executable files into boot-loadable 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++ Integrated Development and Development Environment (IDDE) or on the loader command line. Option settings on the **Load** page correspond to switches typed on the `elfloader.exe` command line.

Splitter Operations

Splitter operations depend on splitter options, which control how the splitter processes executable files into non-bootable files:

- For Blackfin processor, the loader program includes the ROM splitter capabilities invoked through the **Load** page, the **ROM splitter options** category of the **Project Options** dialog box. Refer to "Using the ROM Splitter" on page 2-62. Option settings on the **Load** page correspond to switches typed on the `elfloader.exe` command line.

- For ADSP-21xxx SHARC and ADSP-TSxxx TigerSHARC processors, change the project type to **DSP splitter file**. The splitter options are set via the **Split** page of the **Project Options** dialog box. Refer to "Splitter for SHARC and TigerSHARC Processors" on page 7-1. Option settings on the **Splitter** page correspond to switches typed on the `elfspl21k.exe` command line.

Booting Modes

Once an executable file is fully debugged, the loader is ready to convert the executable file into a processor-loadable (or boot-loadable) file. The loadable file can be automatically downloaded (booted) to the processor after power-up or after a software reset. The way the loader creates a boot-loadable file depends upon how the loadable file is booted into the processor.

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

Analog Devices processors support different boot mechanisms. In general, the following schemes can be used to provide program instructions to the processors after reset.

- "No-Boot Mode"
- "PROM Boot Mode"
- "Host Boot Mode"

No-Boot Mode

After reset, the processor 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 generates a file that can be burned into the PROM memory.

PROM Boot Mode

After reset, the processor starts reading data from a 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*, *loader kernel*, or *boot kernel* (described on page 1-8) parses the boot stream and initializes memories accordingly. The loader kernel runs on the target processor. Depending on the architecture, the loader kernel may execute from on-chip boot RAM or may be preloaded from the PROM device into on-chip SRAM and execute from there.

The loader utility generates the boot stream from the linker output (an executable file) and stores it to file format that can be burned into the PROM.

Host Boot Mode

In this scheme, the target processor is a slave to a host system. After reset, the processor delays program execution until the slave 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. The host halts the target while 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 preloaded by the host devices into the processor's SRAM by any bootstrapping 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 differs from a non-bootable file in that it stores instruction code in a formatted manner in order to be processed by a boot kernel. A non-bootable file stores raw instruction code.

Boot Kernels

A boot kernel (or loader 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 preloaded from the boot source by a bootstrapping 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, an ADSP-2106x/2116x processor brings a 256-word program into internal memory for execution. This small program is a boot kernel.

The boot kernel then brings the rest of the application code into the processor's memory. Finally, the boot kernel overwrites itself with the final block of application code and jumps to the beginning of the application program.

Some of the newer Blackfin processors (ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF535, ADSP-BF536, ADSP-BF537, ADSP-BF538, and ADSP-BF539) do not require a boot kernel—the on-chip boot ROM allows the entire application program's body to be booted into the internal memory of the processor. The on-chip boot ROM for the former Blackfin processors behaves similar to the second-stage loader of the ADSP-BF535 processors. The boot ROM has the capability to parse address and count information for each bootable block.

Loader Tasks

Common tasks performed 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 "File Formats" on page A-1.
- Packing the code for a particular data format: 8-, 16- or 32-bit for some processors.
- Adding the code and data from a specified initialization executable file to the loader file, if applicable.
- Adding a boot kernel on top of the user code.

- If specified, preprogramming the location of the `.LDR` file in a specified PROM space.
- Specifying processor IDs for multiple input `.DXE` files for a multiprocessor system, if applicable.

Boot Streams

The loader output (`.LDR` file) is essentially the same executable code as in the input `.DXE` file; the loader simply repackages the executable as shown in Figure 1-2).

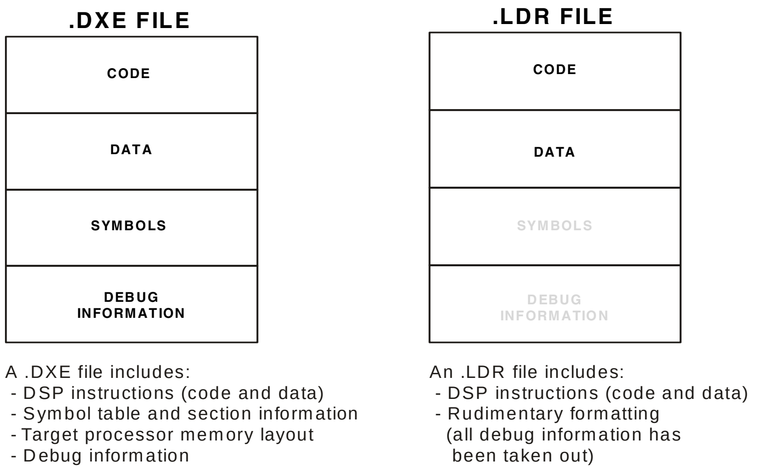


Figure 1-2. A .DXE File Versus an .LDR File

Processor code and data in a loader file (also called a boot stream) is split into blocks. Each code block is marked with a tag that contains information about the block, such as the number of words or destination in the 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); nonzero (code or data); and final (code or data). Depending on the processor family, there may be other block types.

Refer to the following chapters to learn more about boot streams.

File Searches

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

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

When providing an input or output file name as a loader/splitter command-line parameter, use these guidelines:

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

2 LOADER/SPLITTER FOR BLACKFIN PROCESSORS

This chapter explains how the loader/splitter program (`elfloader.exe`) is used to convert executable (`.DXE`) files 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 2-2
  - "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor Booting" on page 2-17
  - "ADSP-BF561 and ADSP-BF566 Processor Booting" on page 2-37

- "Blackfin Processor Loader Guide" on page 2-49

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

Blackfin Processor Booting

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. (Only the ADSP-BF531/BF532/BF533/BF534/BF535/BF536/BF537/BF538/BF539 processors support 24-bit addressable SPI memory booting.) There is also a no-boot option (bypass mode) in which execution occurs from a 16-bit external memory.

At power-up, 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 a 16-bit external memory (Async Bank 0); no-boot mode (bypass on-chip boot ROM)	`000`	`0x2000 0000`
Boot from an 8-bit/16-bit flash memory	`001`	`0xF000 0000`[1]
Boot from an 8-bit address SPI0 serial EEPROM	`010`	`0xF000 0000`[1]

Table 2-1. ADSP-BF535 Processor Boot Mode Selections (Cont'd)

Boot Source	BMODE[2:0]	Execution Start Address
Boot from a 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 2-3
- "ADSP-BF535 Processor Second-Stage Loader" on page 2-5
- "ADSP-BF535 Processor Boot Streams" on page 2-8
- "ADSP-BF535 Processor Memory Ranges" on page 2-15

ADSP-BF535 Processor On-Chip Boot ROM

The on-chip boot ROM for the ADSP-BF535 processor does the following (Figure 2-1).

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 System Reset Configuration Register (`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 2-2.

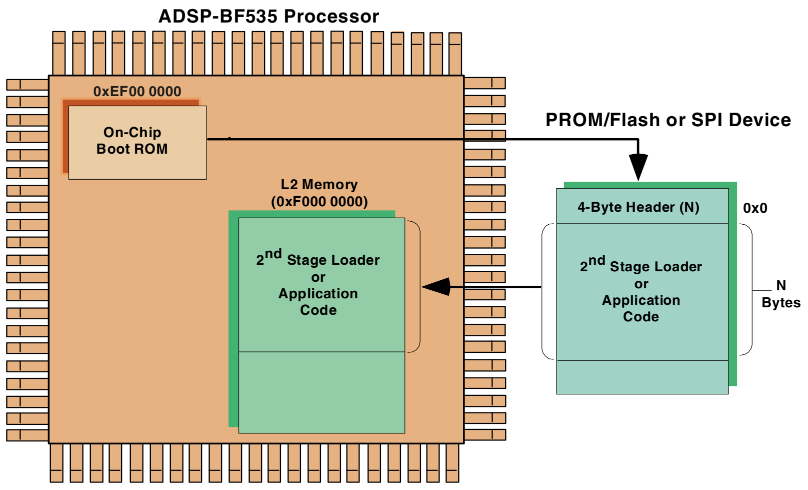
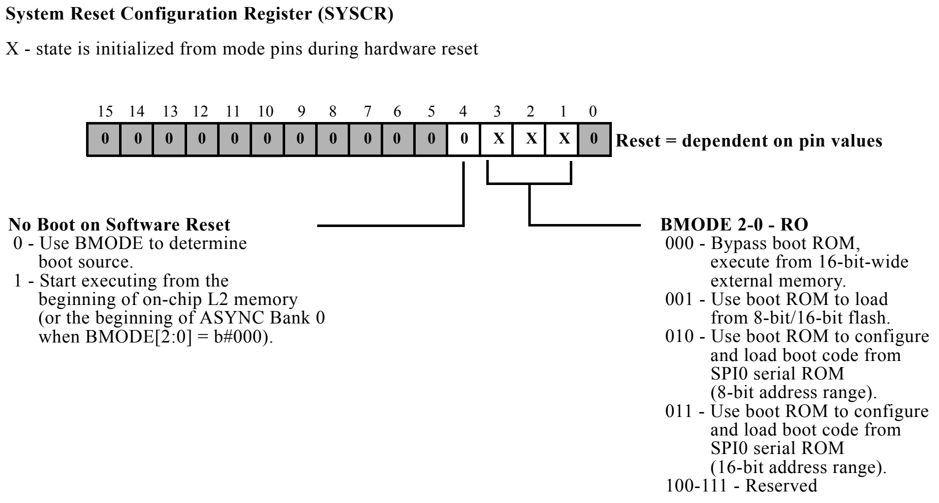


Figure 2-1. ADSP-BF535 Processors: On-Chip Boot ROM


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

3. Finally, if bit 4 of the `SYSCR` register is not set, 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 (called a 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 an SPI boot (see "Command-Line Switches" on page 2-51 for more information on these features).

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

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 2-3).

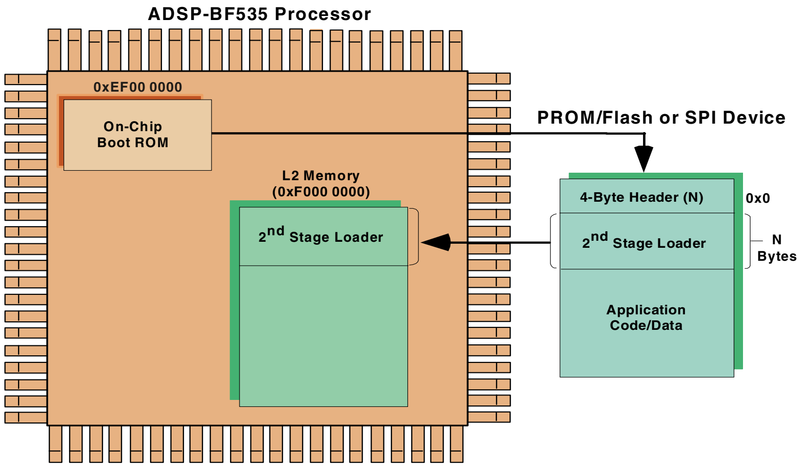


Figure 2-3. ADSP-BF535 Processors: Booting With Second-Stage Loader

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

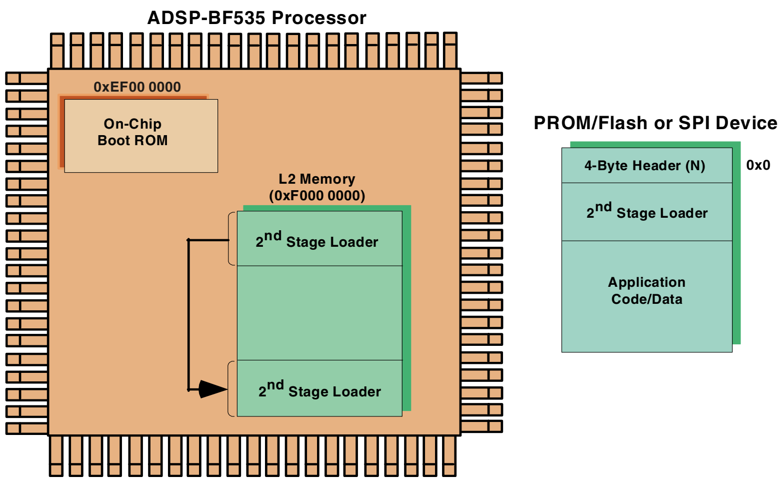


Figure 2-4. ADSP-BF535 Processors: Copying Second-Stage Loader

3. The second-stage loader boots in the application code/data into the various memories of the Blackfin processor (Figure 2-5).

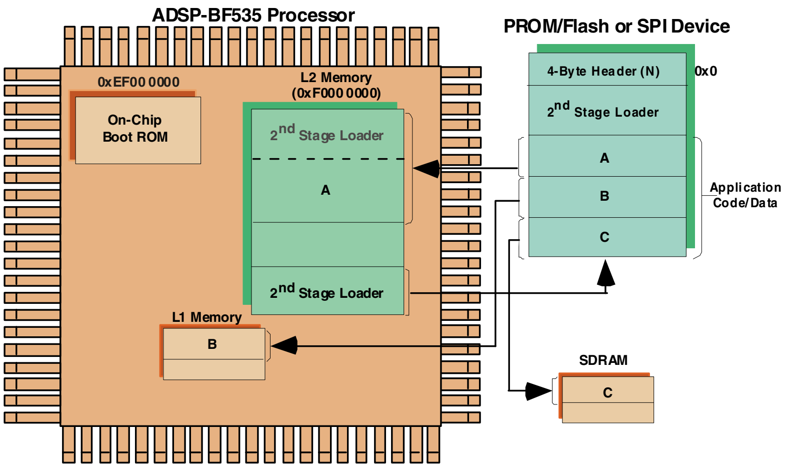


Figure 2-5. ADSP-BF535 Processors: Booting Application Code

4. Finally, after booting, the second-stage loader jumps to the start of L2 memory (`0xF000 0000`) for application code execution (Figure 2-6).

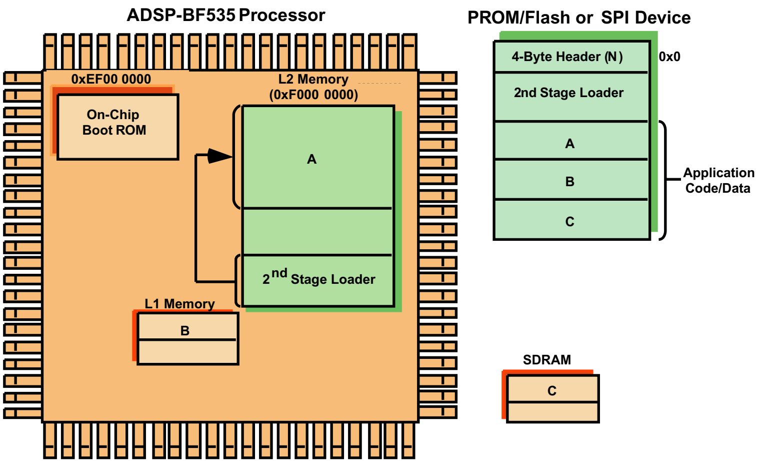


Figure 2-6. ADSP-BF535 Processors: Starting Application Code

ADSP-BF535 Processor Boot Streams

The loader generates the boot stream and places the boot stream in the output loader (`.LDR`) file. The loader prepares the boot stream in a way that enables the on-chip boot ROM and the second-stage loader to load the application code and data to the processor memory correctly. 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:

- "Loader Files Without a Second-Stage Loader" on page 2-9
- "Loader Files With a Second-Stage Loader" on page 2-11
- "Global Headers" on page 2-13
- "Blocks, Block Headers, and Flags" on page 2-14

Loader Files Without a Second-Stage Loader

Figure 2-7 is a graphical representation of an output loader file for 8-bit PROM/flash booting and 8-/16-bit addressable SPI booting without the second-stage loader (kernel).

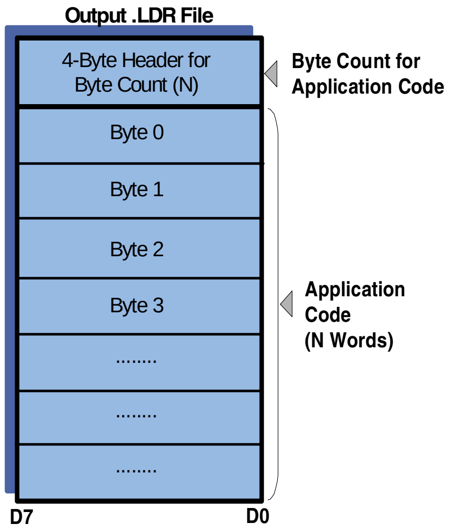

Output .LDR File
4-Byte Header for Byte Count (N)
Byte 0
Byte 1
Byte 2
Byte 3
....
....
....
D7
D0
Byte Count for Application Code
Application Code (N Words)

Figure 2-7. Loader File for 8-/16-bit PROM/Flash Booting Without Kernel

Figure 2-8 is a graphical representation of an output loader file for 16-bit PROM/flash booting without the second-stage loader (kernel).

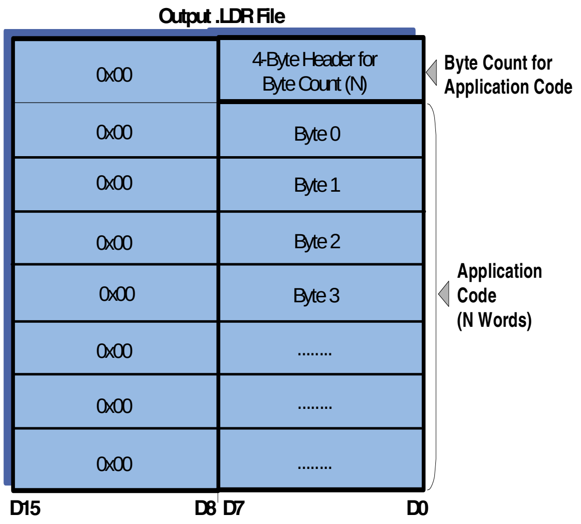


Figure 2-8. Loader File for 16-bit PROM/Flash Booting Without Kernel

Loader Files With a Second-Stage Loader

Figure 2-9 is a graphical representation of an output loader file for 8-bit PROM/flash booting and 8- or 16-bit addressable SPI booting with the second-stage loader (kernel).

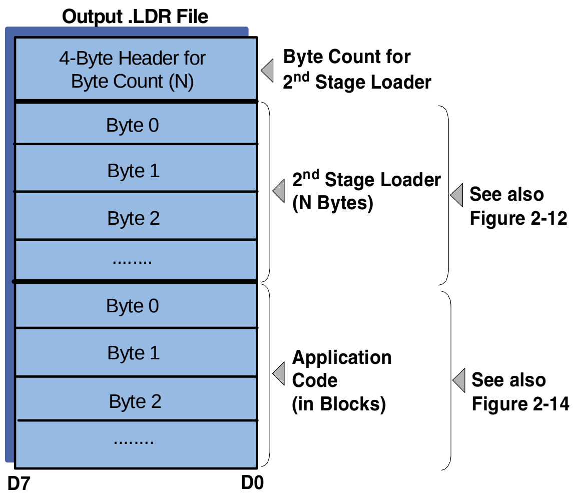

Output .LDR File
4-Byte Header for Byte Count (N)
Byte 0
Byte 1
Byte 2
....
Byte 0
Byte 1
Byte 2
....
D7
D0
Byte Count for 2nd Stage Loader
2nd Stage Loader (N Bytes)
See also Figure 2-12
Application Code (in Blocks)
See also Figure 2-14

Figure 2-9. Loader File for 8-/16-bit PROM/Flash/SPI Booting With Kernel

An output loader file for 16-bit PROM/flash booting with the second-stage loader is illustrated in Figure 2-10.

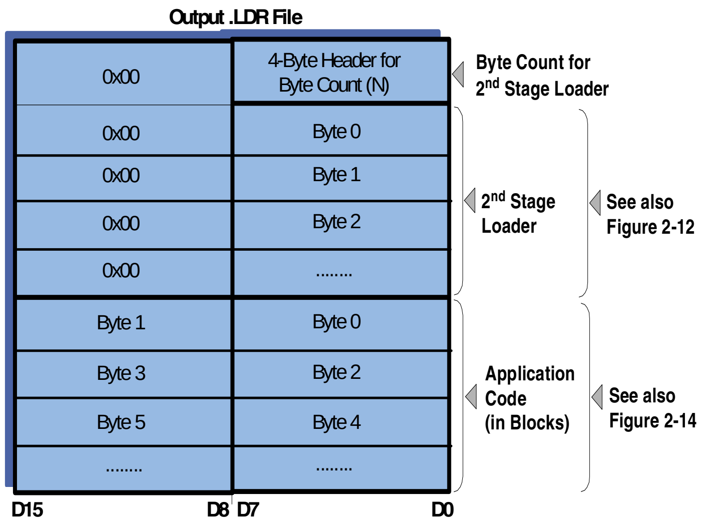


Figure 2-10. Loader File for 16-bit PROM/Flash Booting With Kernel for ADSP-BF531/BF532/BF533 Silicon Revision 0.2 and Below

Global Headers

Following the second-stage loader code and address in a loader file, there is a 4-byte global header. The header provides the global settings for a booting process (see Figure 2-11).

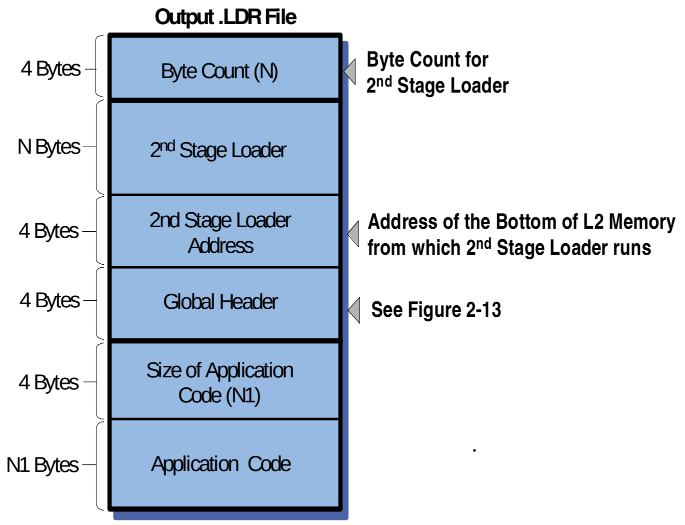


Figure 2-11. Global Header

A global header for 8- and 16-bit PROM/flash booting is illustrated in Figure 2-12.

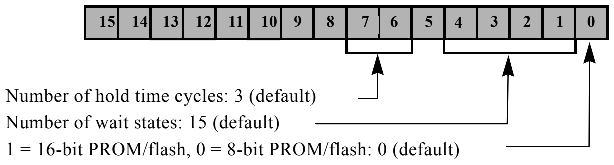

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Number of hold time cycles: 3 (default)
Number of wait states: 15 (default)
1 = 16-bit PROM/flash, 0 = 8-bit PROM/flash: 0 (default)

Figure 2-12. PROM/Flash Booting: Global Header Bits

A global header for 8- and 16-bit addressable SPI booting is illustrated in Figure 2-13.

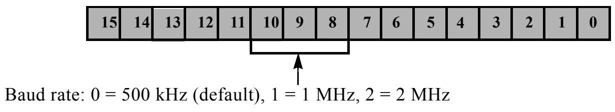


Figure 2-13. Addressable SPI Booting: Global Header Bits

Blocks, Block Headers, and Flags

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 and 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 in Figure 2-14.

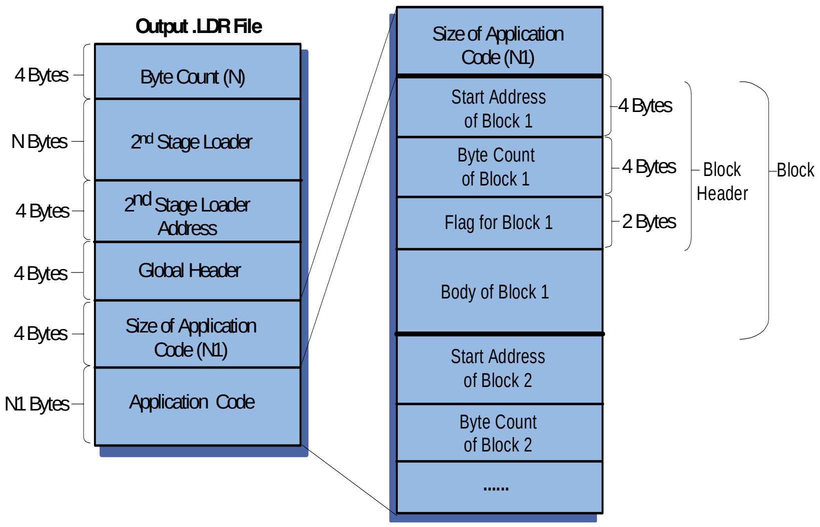


Figure 2-14. Application Block

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

The ADSP-BF535 block flag word's bits are illustrated in Figure 2-15.

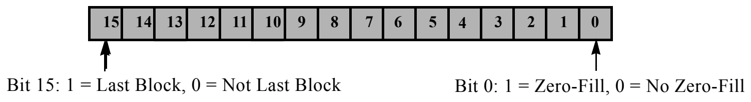

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Bit 15: 1 = Last Block, 0 = Not Last Block
Bit 0: 1 = Zero-Fill, 0 = No Zero-Fill

Figure 2-15. Block Flag Word Bits

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 the Second-Stage Loader" on page 2-59.

Second-Stage Loader Restrictions

Using the second-stage loader imposes the following restrictions.

- 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).
  - ✓ Use the `.ALIGN 8;` directives in the application code.

The two reasons for these restrictions 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/BF534/BF536/BF537/ BF538/BF539 Processor Booting

Upon reset, an ADSP-BF531/BF532/BF533/BF534/BF536/BF537/ BF538/BF539 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, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, and ADSP-BF539 processors.

Table 2-2. Processor Boot Mode Selections for ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processors

Boot Source	BMODE[1:0]	Execution Start Address	
		ADSP-BF531 ADSP-BF532 ADSP-BF538 Processors	ADSP-BF533 ADSP-BF534 ADSP-BF536 ADSP-BF537 ADSP-BF539 Processors
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`
Boot from SPI Host via SPI Slave Mode	`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/BF534/BF536/BF537/BF538/BF539 Processor On-Chip Boot ROM" on page 2-19
- "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor Boot Streams" on page 2-21

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor On-Chip Boot ROM

The on-chip boot ROM for ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539 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 sequence and jump to the start of L1 memory (`0xFFA0 0000` for ADSP-BF533/BF534/BF536/BF537/BF539 processors; `0xFFA0 8000` for ADSP-BF531/BF532/BF538) for execution. The on-chip boot ROM does this by checking bit 4 of the System Reset Configuration Register (`SYSCR`). See Figure 2-16. 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.

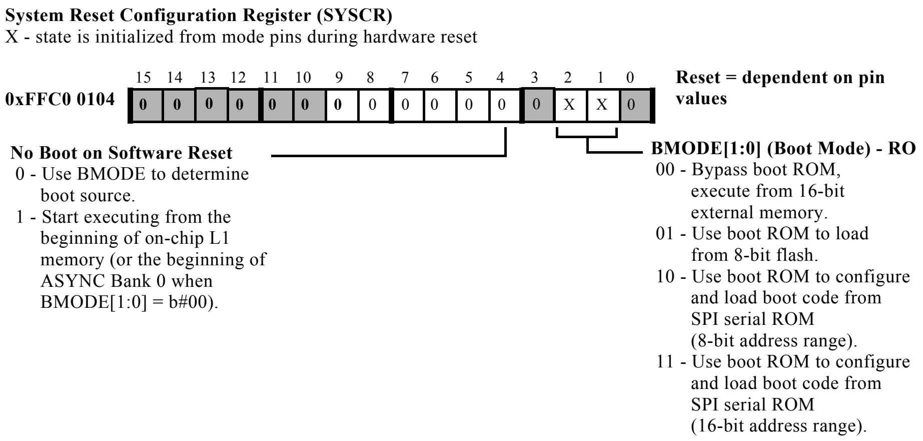

System Reset Configuration Register (SYSCR)
X - state is initialized from mode pins during hardware reset
0xFFC0 0104
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 X X 0
Reset = dependent on pin values
No Boot on Software Reset
0 - Use BMODE to determine boot source.
1 - Start executing 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 2-16. ADSP-BF533 Processors: SYSCR Register

3. Eventually, if bit 4 of the SYSCR register is not set, performs the full boot sequence (Figure 2-17).

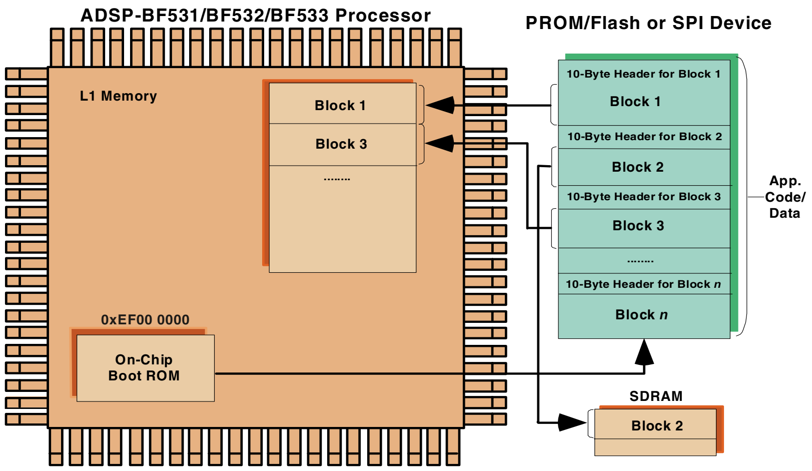


Figure 2-17. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processors: Booting Sequence

The booting sequence for ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 processors is quite different from that for 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/BF534/BF536/BF537/BF538/BF539 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 2-17 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/BF534/BF536/BF537/BF538/BF539 Processor Boot Streams

The following sections describe the boot stream, header, and flag framework for the ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538, and ADSP-BF539 processors.

- "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Blocks, Block Headers, and Flags" on page 2-21
- "Initialization Blocks" on page 2-24

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 "Loader Files With a Second-Stage Loader" on page 2-11). 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.

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Blocks, Block Headers, and Flags

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 2-18), followed by a block body (if it is a nonzero 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; the following text describes the flag structure.

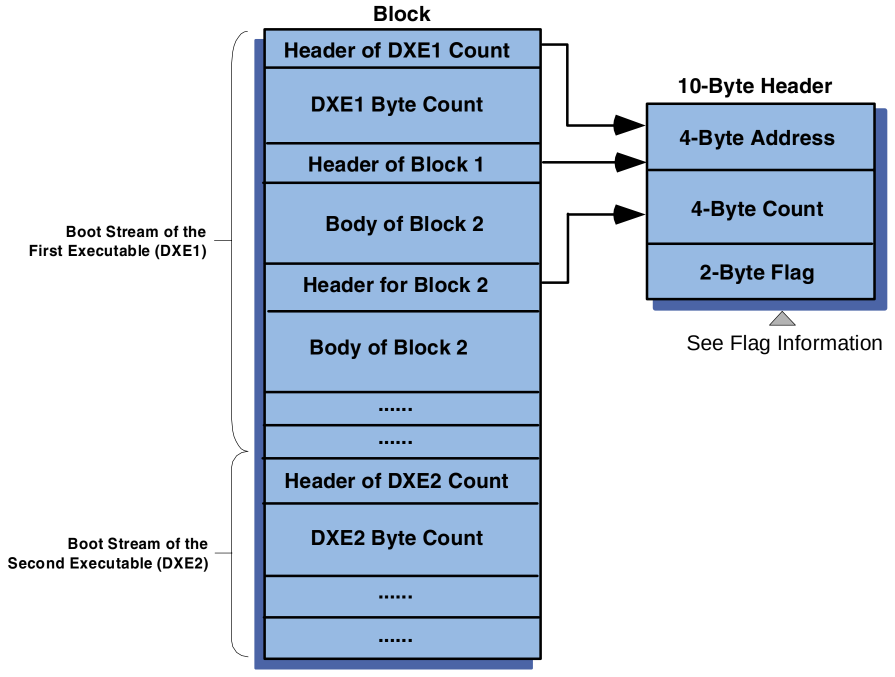


Figure 2-18. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processors: Boot Stream Structure

Refer to Figure 2-19 and Table 2-4 for the flag's bit descriptions.

Table 2-4. Flag Structure

Bit Field	Description
Zero-Fill Block	Indicates that the block is for a buffer filled with zeros. Zero block is not included within the 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 block body in the block.
Ignore Block	Indicates that the block is not to be booted into memory; skips the block and moves 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 an RTS at the end). This option allows the user to run initialization code (such as SDRAM initialization) before the full boot sequence proceeds. Figure 2-20 and Figure 2-21 illustrate the process. Initialization code can be included within the `.LDR` file by using the `-init` switch (see "-init filename" on page 2-52).
Processor Type	Indicates the processor, either ADSP-BF531/BF532/BF538 or ADSP-BF533/BF534/BF536/BF537/BF539. After booting is complete, the on-chip boot ROM jumps to `0xFFA0 0000` for an ADSP-BF533/BF536/BF537/BF539 processor and to `0xFFA0 8000` for an ADSP-BF531/BF532/BF538 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`).

Note that the ADSP-BF537 processor may have a special last block if the boot mode is TWI (Two Wire Interface). The loader will save all the data from `0xFF903F00` to `0xFF903FFF` and will make the last block with the data. The loader, however, will create a regular last block if no data is in that memory range. The space of `0xFF903F00` to `0xFF903FFF` is saved for the boot ROM to use as a data buffer during a booting process.

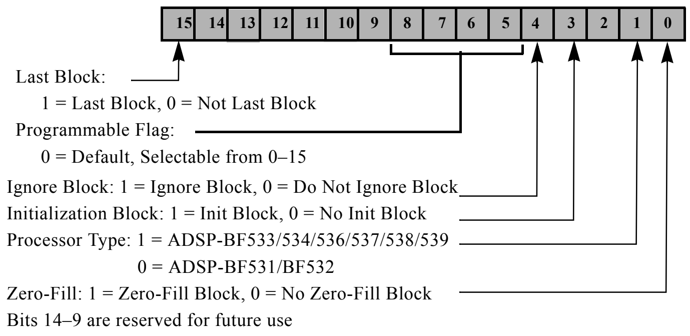

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Last Block:
1 = Last Block, 0 = Not Last Block
Programmable Flag:
0 = Default, Selectable from 0–15
Ignore Block: 1 = Ignore Block, 0 = Do Not Ignore Block
Initialization Block: 1 = Init Block, 0 = No Init Block
Processor Type: 1 = ADSP-BF533/534/536/537/538/539
0 = ADSP-BF531/BF532
Zero-Fill: 1 = Zero-Fill Block, 0 = No Zero-Fill Block
Bits 14–9 are reserved for future use

Figure 2-19. Flag Bit Assignments for the 2-Byte Block Flag Word

Initialization Blocks

The `-init` *`filename`* option directs the loader to produce the initialization block from the code of the initialization section of the named file. The initialization blocks are placed at the top of a loader file. They are executed before the rest of the code in the loader file is booted into the memory (see Figure 2-20).

Following execution of the initialization blocks, the booting process continues with the rest of data blocks until it encounters a final block (see Figure 2-21). The initialization code example follows in Listing 2-1.

Listing 2-1. Initialization Block Code Example

```
/*  This file contains 3 sections: */
/*  1) A Pre-Init Section-this section saves off all the
       processor 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.
```

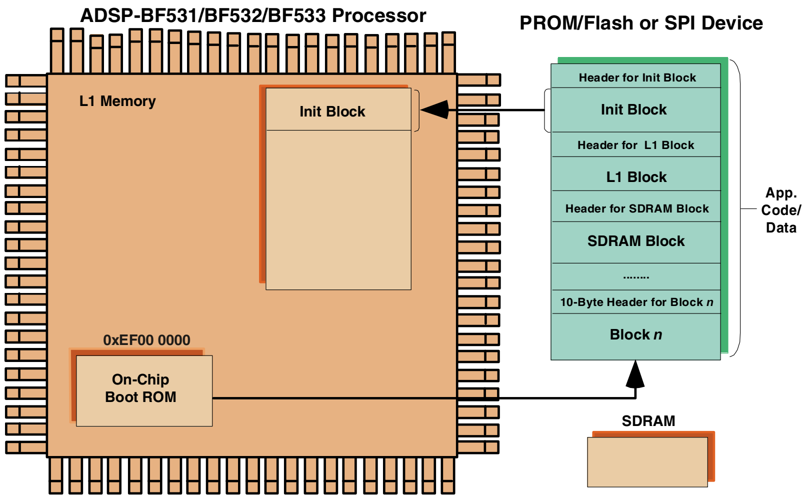


Figure 2-20. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Initialization Block Execution

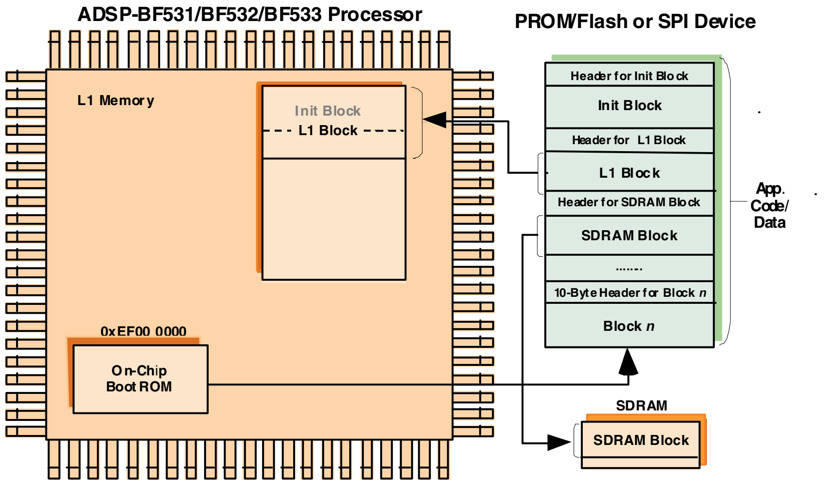


Figure 2-21. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Booting Application Code

```
    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/BF534/BF536/BF537/BF538/BF539 Processor Memory Ranges

The on-chip boot ROM on ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 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`–`0xFFA0 BFFF`)
    - ADSP-BF532 processor
        - ✓ Data Bank A SRAM (`0xFF80 4000`–`0xFF80 7FFF`)
        - ✓ Data Bank B SRAM (`0xFF90 4000`–`0xFF90 7FFF`)
        - ✓ Instruction SRAM (`0xFFA0 8000`–`0xFFA1 3FFF`)
    - ADSP-BF533 processor
        - ✓ Data Bank A SRAM (`0xFF80 0000`–`0xFF80 7FFF)`
        - ✓ Data Bank B SRAM (`0xFF90 000`–`0xFF90 7FFF)`
        - ✓ Instruction SRAM (`0xFFA0 0000`–`0xFFA1 3FFF`)
    - ADSP-BF534 processor
        - ✓ Data Bank A SRAM (`0xFF80 0000`–`0xFF80 7FFF`)
        - ✓ Data Bank B SRAM (`0xFF90 0000`–`0xFF90 7FFF`)
        - ✓ Instruction SRAM (`0xFFA0 0000`–`0xFFA1 3FFF`)

- ADSP-BF535 processor
  - ✓ Data Bank A SRAM (`0xFF80 0000`–`0xFF80 3FFF`)
  - ✓ Data Bank B SRAM (`0xFF90 0000`–`0xFF90 3FFF`)
  - ✓ Instruction SRAM (`0xFFA0 0000`–`0xFFA0 3FFF`)
- ADSP-BF536 processor
  - ✓ Data Bank A SRAM (`0xFF80 4000`–`0xFF80 7FFF`)
  - ✓ Data Bank B SRAM (`0xFF90 4000`–`0xFF90 7FFF`)
  - ✓ Instruction SRAM (`0xFFA0 0000`–`0xFFA1 3FFF`)
- ADSP-BF537 processor
  - ✓ Data Bank A SRAM (`0xFF80 0000`–`0xFF80 7FFF`)
  - ✓ Data Bank B SRAM (`0xFF90 0000`–`0xFF90 7FFF`)
  - ✓ Instruction SRAM (`0xFFA0 0000`–`0xFFA1 3FFF`)
- ADSP-BF538 processor
  - ✓ Data Bank A SRAM (`0xFF80 4000`–`0xFF80 7FFF`)
  - ✓ Data Bank B SRAM (`0xFF90 4000`–`0xFF90 7FFF`)
  - ✓ Instruction SRAM (`0xFFA0 8000`–`0xFFA1 3FFF`)
- ADSP-BF539 processor
  - ✓ Data Bank A SRAM (`0xFF80 0000`–`0xFF80 3FFF`)
  - ✓ Data Bank B SRAM (`0xFF90 2000`–`0xFF90 7FFF`)
  - ✓ Instruction SRAM (`0xFFA0 0000`–`0xFFA1 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/BF534/BF536/BF537/BF538/BF539 Processor SPI Slave Mode Boot via Master Host (BMODE = 10)

For SPI slave mode booting, the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 processor is configured as an SPI slave device and a host is used to boot the processor.

This boot mode is *not* supported in silicon revision 0.2 and earlier of the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 processors.

Figure 2-22 shows the pin-to-pin connections needed for SPI slave mode.

The host does not need any knowledge of the loader file stream to boot the Blackfin processor. It must be configured to send one byte at a time from the loader file (in ASCII format). In the above setup, `PFx` is the feedback strobe from the Blackfin processor to the master host device. This will be the signal used by the Blackfin processor to hold off the host during certain times within the boot process (specifically during init code execution and zero-fill blocks). When `PFx` is asserted (high), the master host device must discontinue sending bytes to the Blackfin processor. When `PFx` is de-asserted (low), the master host device will resume sending bytes from where it left off. Since the `PFx` pin is not driven by the slave until the first block has been processed, consider using a resistor to pull down the feedback strobe.

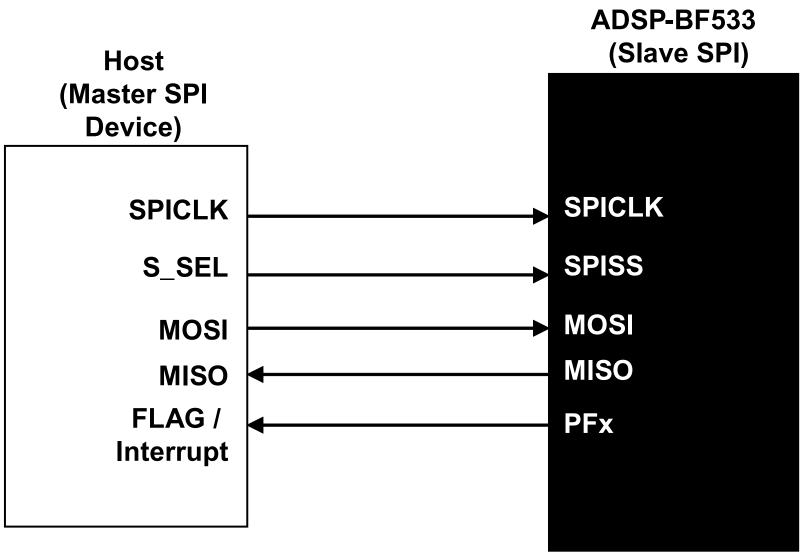


Figure 2-22. Pin-to-Pin Connections for ADSP-BF531/BF532/BF533 Processor SPI Slave Mode

This `PFx` number is going to be user-defined and will be embedded within the loader file. The elfloader utility will embed this number in bits [8:5] of the `FLAG` field within every 10-byte header. It does this by using the `-pflag` number command-line switch where number is the intended PF flag used by the Blackfin slave and has a value between 1 and 15.

If the `-pflag` number switch is not used, the default value placed within bits 8:5 of the `FLAG` will be `0`, indicating that `PF0` will be assumed as the feedback signal to the host. Since `PF0` is multiplexed with the `/SPISS` pin, which is mandatory for successful SPI slave boot, always use the `-pflag` switch and specify a value other than `0`.

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor SPI Master Mode Boot via SPI Memory (BMODE = 11)

For SPI master mode booting, the ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539 processor is configured as a SPI master connected to a SPI memory. The following shows the pin-to-pin connections needed for this mode.

Figure 2-23 shows the pin-to-pin connections needed for SPI master mode.

A pull-up resistor on `MISO` is *required* for this boot mode to work properly. For this reason, the ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/ BF539 processor reads a `0xFF` on the `MISO` pin if the SPI memory is not responding (that is, no data written on the `MISO` pin by the SPI memory).

Although the pull-up resistor on the `MISO` line is mandatory, additional pull-up resistors might also be worthwhile as well. Pull up the `PF2` signal to ensure the SPI memory is not activated while the Blackfin processor is in reset.

On silicon revision 0.2 and earlier, the `CPHA` and `CPOL` bits within the SPI Control (`SPICTL`) register were both set to 1. (Refer to the *ADSP-BF533 Blackfin Processor Hardware Reference* for information on these bits.) For this reason, the SPI memory may detect an erroneous rising edge on the clock signal when it recovers from three-state. If the boot process fails because of this situation, a pull-up resistor on the `SPICLK` signal will alleviate the problem. On silicon revision 0.3, this was fixed by setting `CPHA = CPOL = 0` within the SPI Control register.

The SPI memories supported by this interface are standard 8/16/24-bit addressable SPI memories (the read sequence is explained below) and the following Atmel SPI DataFlash devices: AT45DB041B, AT45DB081B, AT45DB161B.

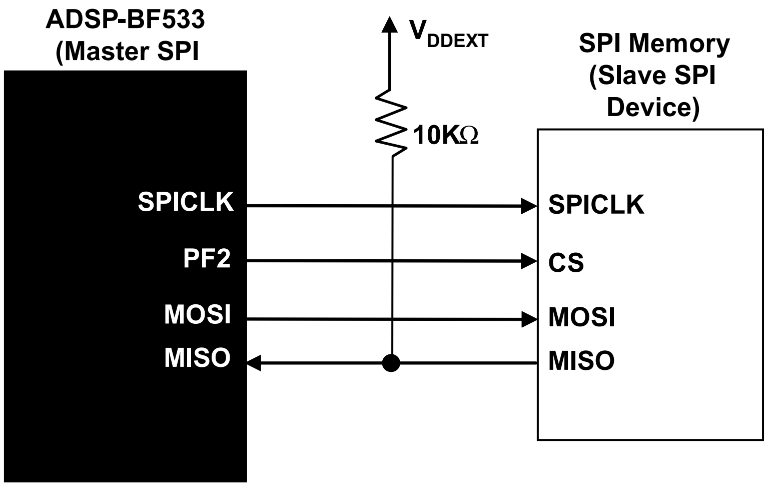


Figure 2-23. Pin-to-pin connections for ADSP-BF531/BF532/BF533 Processor SPI Master Mode

Standard 8/16/24-bit addressable SPI memories take in a read command byte of `0x03`, followed by one address byte (for 8-bit addressable SPI memories), two address bytes (for 16-bit addressable SPI memories), or three address bytes (for 24-bit addressable SPI memories). After the correct read command and address are sent, the data stored in the memory at the selected address is shifted out on the `MISO` pin. Data is sent out sequentially from that address with continuing clock pulses. Analog Devices has tested the following standard SPI memory devices.

- 8-bit addressable SPI memory: 25LC040 from Microchip
- 16-bit addressable SPI memory: 25LC640 from Microchip
- 24-bit addressable SPI memory: M25P80 from STMicroelectronics

SPI Memory Detection Routine

Since `BMODE` = 11 supports booting from various SPI memories, the on-chip boot ROM will detect what type of memory is connected. To determine the type of memory (8-, 16-, or 24-bit addressable) connected to the processor, the on-chip boot ROM sends the following sequence of bytes to the SPI memory until the memory responds back. The SPI memory does not respond back until it is properly addressed. The on-chip boot ROM does the following.

1. Sends the read command, `0x03`, on the `MOSI` pin then does a dummy read of the `MISO` pin.
2. Sends an address byte, `0x00`, on the `MOSI` pin then does a dummy read of the `MISO` pin.
3. Sends another byte, `0x00`, on the `MOSI` pin and checks whether the incoming byte on the `MISO` pin is anything other than `0xFF` (This is the value from the pull-up resistor. For more information, refer to the following note.) An incoming byte that is not `0xFF` means that the SPI memory has responded back after one address byte and an 8-bit addressable SPI memory device is assumed to be connected.
4. If the incoming byte is `0xFF`, the on-chip boot ROM sends another byte, `0x00`, on the `MOSI` pin and checks whether the incoming byte on the `MISO` pin is anything other than `0xFF`. An incoming byte other than `0xFF` means that the SPI memory has responded back after two address bytes and a 16-bit addressable SPI memory device is assumed to be connected.
5. If the incoming byte is `0xFF`, the on-chip boot ROM sends another byte, `0x00`, on the `MOSI` pin and checks whether the incoming byte on the `MISO` pin is anything other than `0xFF`. An incoming byte other than `0xFF` means that the SPI memory has responded back after three address bytes and a 24-bit addressable SPI memory device is assumed to be connected.

6. If an incoming byte is `0xFF` (meaning no devices have responded back), the on-chip boot ROM assumes that one of the following Atmel DataFlash devices are connected: AT45DB041B, AT45DB081B, or AT45DB161B. These DataFlash devices have a different read sequence than the one described above for standard SPI memories. The on-chip boot ROM determines which of the above Atmel DataFlash memories are connected by reading the status register.

For the SPI memory detection routine explained above, the on-chip boot ROM in silicon revision 0.2 and earlier checks whether the incoming data on the `MISO` pin is `0x00` (first byte of the loader file). The on-chip boot ROM in silicon revision 0.3 checks whether the incoming data on the `MISO` pin is anything *other* than `0xFF`. For this reason, SPI loader files built for silicon revision 0.2 and earlier must have the first byte as `0x00`. For silicon revision 0.3, the first byte of the loader file is set to `0x40`.

The SPI baud rate register is set to 133, which, when based on a 54 MHz system clock, results in a 54 MHz/(2*133) = 203 kHz baud rate. On the ADSP-BF533 EZ-KIT Lite board, the default system clock frequency is 54 MHz.

ADSP-BF534/BF536/BF537 Processor Booting

The ADSP-BF534/BF536/BF537 processors support the boot modes listed in Table 2-5. They include all the boot modes supported in the ADSP-BF531/BF532/BF533 processors in addition to booting from a TWI (Two Wire Interface) serial device, a TWI host, and a UART host.

Table 2-5. ADSP-BF534/BF536/BF537 Processor Boot Modes

BMODE[2:0]	**Description**
000	Executes from external 16-bit memory connected to ASYNC Bank0 (bypass boot ROM)
001	Boots from 8/16-bit flash/PROM
010	Reserved
011	Boots from a 8/16/24-bit addressable SPI memory in SPI Master mode with support for Atmel AT45DB041B, AT45DB081B, and AT45DB161B DataFlash® devices
100	Boots from a SPI host in SPI Slave mode
101	Boots from an TWI serial device
110	Boots from an TWI Host
111	Boots from a UART Host

ADSP-BF561 and ADSP-BF566 Processor Booting

The booting sequence for the ADSP-BF561 and ADSP-BF566 dual-core processors is similar to the ADSP-BF531/BF532/BF533 processor booting sequence (described on page 2-17). 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 Blackfin Processor Hardware Reference* manual for information about the processor's operating modes and states. Please refer to the "System Reset and Power-up Configuration" section 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 2-6 summarizes the boot modes and execution start addresses for ADSP-BF561 processors.

Table 2-6. 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 Interrupt Service Routine. 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 code is run 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. A bit-by-bit description of the global header is presented in Table 2-7. 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 2-7. 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 block header structure is the same as that of the ADSP-BF531/BF532/BF533 processors (described in "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Blocks, Block Headers, and Flags" on page 2-21). 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 "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Blocks, Block Headers, and Flags" on page 2-21.

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 2-8. 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 2-8. 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

Table 2-8. ADSP-BF561 Processor Boot Stream Structure (Cont'd)

Bit Field	Description
112–119	LSB of the first 1st .DXE byte count
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

Table 2-8. ADSP-BF561 Processor Boot Stream Structure (Cont'd)

Bit Field	Description
…	LSB of the byte count field of the nth block of 1st `.DXE`
…	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`
…	And so on …
…	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/BF566 Processor Memory Ranges

The on-chip boot ROM of the ADSP-BF561/BF566 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`)
- 128K of Shared L2 Memory (`FEB0 0000` – `FEB1 FFFF`)
- 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/BF566 Processor Initialization Blocks

The initialization block or a second-stage loader must be used to initialize the SDRAM memory of the ADSP-BF561/BF566 processor before any instructions or data are loaded into it.

The initialization blocks are identified by a bit in the flag word of the 10-byte block header. When the boot ROM encounters the initialization blocks in the boot stream, it loads the blocks and executes them immediately. The initialization blocks 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 "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Blocks, Block Headers, and Flags" on page 2-21.

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 execution of the initialization blocks, 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-BF53x and ADSP-BF561/BF566 Multiple .DXE Booting" on page 2-46.

ADSP-BF561/BF566 Multiple .DXE Booting

A typical dual-core application is separated into two executable files; one for each core. The default linker description file (LDF) for the ADSP-BF561/BF566 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 run-time libraries. When using shared memory and/or C run-time 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 `.DXE` files are loaded, a second-stage loader is 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 blocks, 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` files 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-BF53x and ADSP-BF561/BF566 Multiple .DXE Booting

This section describes how to boot more than one `.DXE` file into an ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539 and ADSP-BF561/BF566 processor. The information presented in this section applies to all of the named processors. For additional information on the ADSP-BF561/BF566 processor, refer to "ADSP-BF561/BF566 Multiple .DXE Booting" on page 2-45.

The ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539 and ADSP-BF561/BF566 loader file structure and the silicon revision of 0.1 and higher allow the booting of multiple `.DXE` files into a single processor from external memory. As illustrated in Figure 2-24, 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 "ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Blocks, Block Headers, and Flags" on page 2-21.

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

- Use the second-stage loader switch, "-l userkernel". This option allows the use of 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` files from external memory.

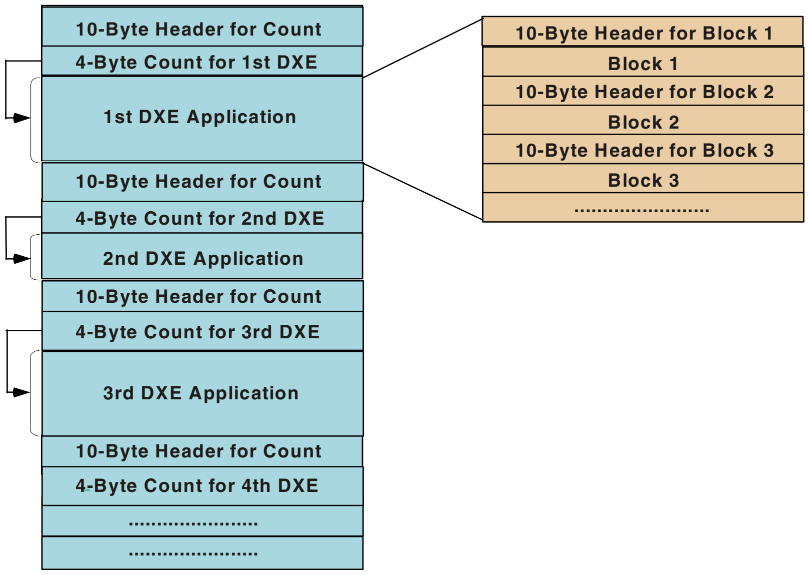


Figure 2-24. ADSP-BF531/BF32/BF33/BF534/ BF536/BF537/BF538/ BF539/BF561/BF566: Multi-Application Booting

- Use the initialization block switch, "-init filename", where "filename" is the name of the executable file containing the initialization code. This option allows you to change the external memory pointer and boot a specific .DXE file 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. You select features such as boot mode, boot kernel, and output file format via the loader options. 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 the ADSP-BF5xx Blackfin Loader Command Line" on page 2-49
- "Using the Base Loader" on page 2-57
- "Using the Second-Stage Loader" on page 2-59
- "Using the ROM Splitter" on page 2-62

Using the ADSP-BF5xx Blackfin Loader Command Line

The ADSP-BF5xx Blackfin loader uses the following command-line syntax.

For a single input file:

```
elfloader inputfile -proc processor [-switch ...]
```

For multiple input files:

```
elfloader inputfile1 inputfile2 … -proc processor [-switch ...]
```

where:

- *inputfile*—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 multi-input systems, specify multiple input .DXE files. Put the input file names 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 built. Provide a processor part number for every input .DXE if designing multiprocessor systems.

- *-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 multi-input system. For a multi-input system, the loader processes the input files in the order presented on the command line.

File Searches

File searches are important in loader processing. The loader supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-11.

File Extensions

Some loader switches take a file name as an optional parameter. Table 2-9 lists the expected file types, names, and extensions.

Table 2-9. 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

In some cases the loader expects the overlay input files with the file extension of `.OVL`, shared memory input files with the extension of `.SM`, or both, but does not expect those files to appear in a command line or on the **Load** property page. The loader will find them in the directory of the associated `.DXE` files, in the current working directory, or in the directory specified in the `.LDF` file.

Command-Line Switches

A summary of the loader command-line switches appears in Table 2-10.

Table 2-10. Blackfin Loader Command-Line Switches

Switch	Description

`-b prom`
`-b flash`
`-b spi`
`-b spislave`
`-b UART`
`-b TWI`
	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, SPI, SPI Slave, UART, and TWI. SPI Slave, UART, and TWI are for the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539/ processors only.
	If `-b` does not appear on the command line, the default is `-b flash`.

Table 2-10. Blackfin Loader Command-Line Switches (Cont'd)

Switch	Description
`-baudrate #`	Accepts a baud rate for SPI booting only. **Note:** Currently supports only 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 input `.DXE` files with the encryption algorithms in the dynamic library file `dll_filename`. If the *`dll_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 format of a boot-loadable file (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, SPI Slave, UART, and TWI.

`-ghc #`	Specifies a 4-bit value (global header cookie) for bits 31–28 of the global header. This switch is for ADSP-BF561/BF566 processors only.

`-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.

`-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 supports only ADSP-BF535 processors.
`-init` *`filename`*	Directs the loader to include the initialization code 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 code and then calls it. It is the responsibility of the code to save/restore state/registers and then perform a RTS back to the kernel. **Note:** This switch cannot be applied to ADSP-BF535 processors.

Table 2-10. Blackfin Loader Command-Line Switches (Cont'd)

Switch	Description

`-kb prom`
`-kb flash`
`-kb spi`
`-kb spislave`
`-kb UART`
`-kb TWI`
	Specifies the boot mode (PROM, Flash, SPI, SPI Slave, UART, or TWI) for the boot kernel output file if you generate two output files from the loader: one for boot kernel and another for user application code. SPI Slave, UART, and TWI are for the ADSP-BF531/BF532/ BF533/BF534/ BF536/BF537/BF538/BF539/ processors only.
	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 program.

`-kf hex`
`-kf ascii`
`-kf binary`
	Specifies the output file format (hex, ASCII, or binary) for the boot kernel if you output two files from the loader: one for boot kernel and one 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.

`-kenc` *`dll_filename`*	Specifies the user encryption dynamic library file for the encryption of 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.
`-kp #`	Specifies a hex PROM/flash output start address for kernel code. A valid value is between [`0x0`, `0xFFFFFFFF`]. The specified value is not 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 is used in conjunction with the `-o2` switch.

Table 2-10. Blackfin Loader Command-Line Switches (Cont'd)

Switch	Description
`-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 is 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 multiple 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.
`-Mt` *`filename`*	Specifies the make dependencies target output file. 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 supports only ADSP-BF535 processors.
`-o` *`filename`*	Directs the loader to use the specified *`filename`* as the name of the loader output file. If the *`filename`* is absent, the default name is the name of the input file with an `.LDR` extension.

Table 2-10. Blackfin Loader Command-Line Switches (Cont'd)

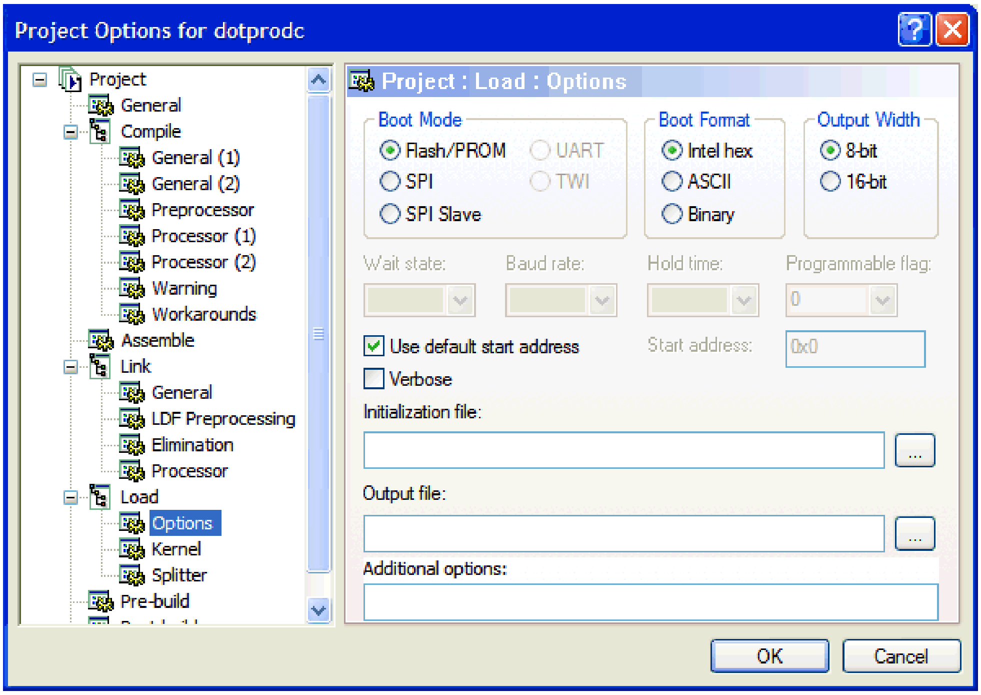
Switch	Description
`-o2`	Produces two output files: one for the Init block (if present) and boot kernel and one for user application code. To have a different format, boot mode, or output width from the application code output file, use the `-kb -kf -kwidth` switches to specify the boot mode, the boot format, and the boot width for the output kernel file. If you intend use the `-o2` switch, do not combine it with: `-nokernel` on ADSP-BF535 processors You must combine it with: `-l` *`filename`* and/or `-init` *`filename`* on ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539/BF561/BF566 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 (a single output file).
`-pFlag #`	Specifies a 4-bit hex value for strobe (programmable flag). The default value is zero (0). This switch is for ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539/ BF561/BF566 processors only.
`-proc` *`processor`*	Specifies the target processor. The *`processor`* can be one of the following: `ADSP-BF531`, `ADSP-BF532`, `ADSP-BF533`, `ADSP-BF534`, `ADSP-BF535`, `ADSP-BF536`, `ADSP-BF537`, `ADSP-BF538`, `ADSP-BF539`, `ADSP-BF561` and `ADSP-BF566`.
`-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 'split' 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.
`-ShowEncryptionMessage`	Displays a message returned from the encryption function.

Table 2-10. Blackfin Loader Command-Line Switches (Cont'd)

Switch	Description
`-si-revision #|none`	Provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` switch. The parameter takes one of two forms: • The `none` value indicates that the VDSP++ tool should ignore silicon errata. • The `#` value indicates 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`. Revision `0.1` is distinct from and "lower" than revision `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`. 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 switch parameter (a valid revision 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 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 supports only 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 the Base Loader

The **Load** page of the **Project Options** dialog consists of multiple panes and is shared with 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 2-25 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 2-51 for more information on the switches.



Boot Mode: [✓ Flash/PROM] UART SPI TWI SPI Slave

Boot Format: [✓ Intel hex] ASCII Binary

Output Width: [✓ 8-bit] 16-bit

Wait state:

Baud rate:

Hold time:

Programmable flag: 0

[✓ Use default start address]

Start address: 0x0

Verbose

Initialization file:

Output file:

Additional options:

Figure 2-25. Loader File Options Page for Blackfin Processors

Using the page controls, select or modify the loader settings. Table 2-11 describes each loader control and corresponding setting. When satisfied with default settings, click **OK** to complete the loader setup.

Table 2-11. Base Loader Page Settings

Setting	Description
Load	Selections for the loader. The options are: • **Options** – default booting options (this section) • **Kernel** – specification for a second-stage loader (see on page 2-59) • **Splitter** – specification for the no-boot mode (see on page 2-62) If you do not use the boot kernel for ADSP-BF535 processors, the **Kernel** page 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 2-54. For ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539/BF561/ BF566 processors, which do not have software boot kernels by default, select the boot kernel to use one.
Boot mode	Specifies PROM, Flash, SPI, SPI Slave, UART, or TWI 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 wait states for external access (`0`–`15`). The selection is active for ADSP-BF535 processors. For ADSP-BF531/BF532/ BF533/BF534/BF536/BF537/BF538/BF539/BF561/BF566 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/BF532/ BF533/BF534/BF536/BF537/BF538/BF539/BF561/BF566 processors, the field is grayed out.
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/BF532/ BF533/BF534/BF536/BF537/BF538/BF539/BF561/BF566 processors, the field is grayed out.

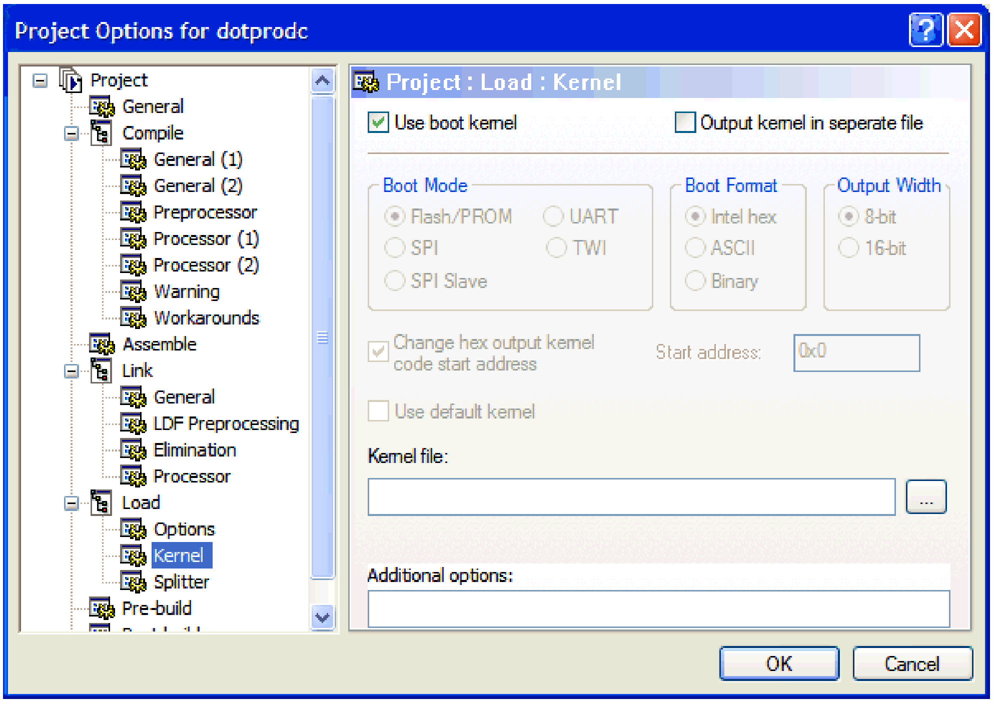
Table 2-11. Base Loader Page Settings (Cont'd)

Setting	Description
Programmable flag	Selects a programmable flag number (from 0 to 15) as strobe. This box is active if **SPI Slave** is selected for ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 processors.
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 multi-input system. Type the file names with the paths (they are not in the current working directory) separated with a space between two names, and the loader will retrieve these input files. **Note:** The loader processes the input files in the order in which the files appear on the command line generated form the property page.

Using the Second-Stage Loader

If you use a second-stage loader, select **Kernel** (under **Load** in the **Project Options** tree control). The page shows how to configure the loader for boot loading and to output file generation using the boot kernel.

Figure 2-26 shows a sample **Kernel** page, with boot options for a Blackfin processor.



Project : Load : Kernel

[✓] Use boot kernel
[ ] Output kernel in seperate file

Boot Mode: [● Flash/PROM] UART SPI TWI SPI Slave
Boot Format: [● Intel hex] ASCII Binary
Output Width: [● 8-bit] 16-bit

[✓] Change hex output kernel code start address
Start address: 0x0

[ ] Use default kernel

Kernel file:

Additional options:

Figure 2-26. Kernel Page – Boot Options for an ADSP-BF53x Blackfin Processor

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

1. Select **Options** (under **Load**) to set up base booting options (see "Using the Base Loader" on page 2-57).

2. Select **Kernel** (under **Load**) to open the **Kernel** page for the second-stage loader settings, shown in Figure 2-26.

3. Select **Use boot kernel**. By default, this option is selected for ADSP-BF535 and grayed out for ADSP-BF531/BF532/BF533/ BF534/ BF536/BF537/BF538/BF539/BF561/BF566 processors.

4. To produce two output files (Init code and/or 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 an ADSP-BF535 processor) or enter a kernel's file name 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/BF532/BF533/BF534/ BF536/BF537/BF538 /BF539/BF561/BF566 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 the ROM Splitter

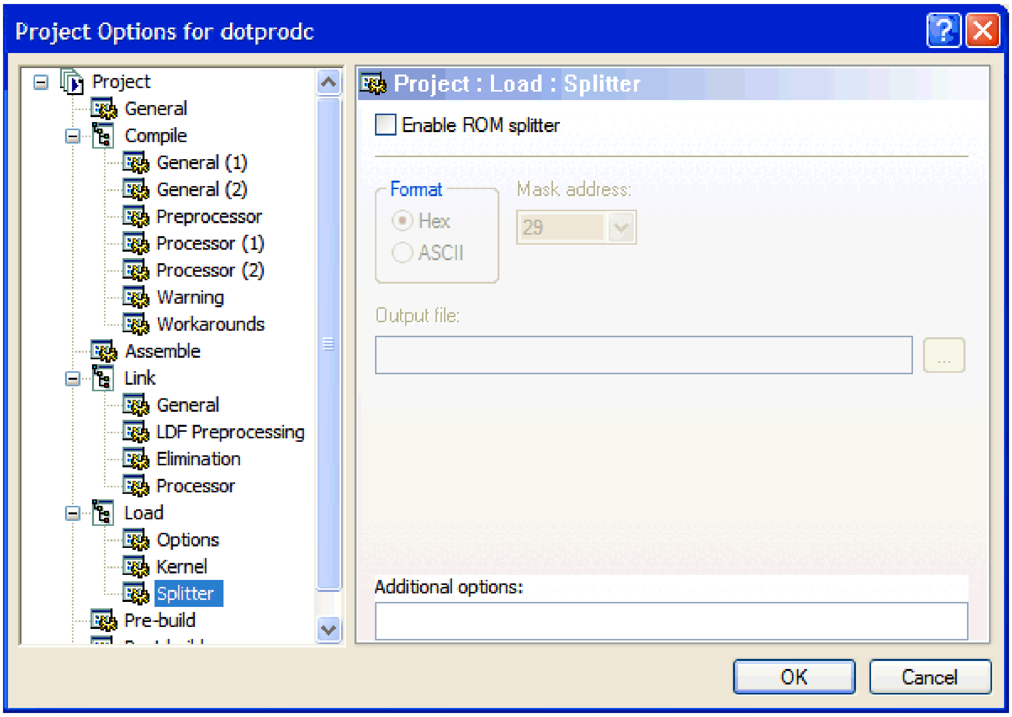
Unlike the loader utility, the splitter does not format the application data when transforming an .DXE file to an .LDR file. It emits raw data only. Whether data and/or instruction segments are processed by the loader or by the splitter utility depends upon the LDF's TYPE() command. Segments declared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

Figure 2-27 shows a sample **Splitter** page, with ROM splitter options. 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.

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/BF532/BF533/BF534/ BF536/BF537/ BF538/BF539 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.



Enable ROM splitter [ ]

Format: Hex [✓], ASCII

Mask address: 29

Output file:

Additional options:

Figure 2-27. Splitter Page – ROM Splitter Options for Blackfin Processors

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 for the reset vector to be located accordingly. The following code fragments (Listing 2-3 and Listing 2-4) 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;
               1 = 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 LOADER FOR ADSP-TSXXX TIGERSHARC PROCESSORS

This chapter explains how the loader/splitter program (`elfloader.exe`) is used to convert executable (`.DXE`) files into boot-loadable or non-bootable files for ADSP-TSxxx TigerSHARC 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-TSxxx TigerSHARC processors are detailed in the following sections.

- "ADSP-TSxxx TigerSHARC Processor Booting" on page 3-2
  Provides general information on various booting modes, including information on boot kernels.
- "TigerSHARC Loader Guide" on page 3-5
  Provides reference information on the loader's command-line syntax and switches.

Refer to the processor's Data Sheet and Hardware Reference for more information on system configuration, peripherals, registers, and operating modes.

ADSP-TSxxx TigerSHARC Processor Booting

At chip reset, a TigerSHARC processor loads (bootstraps) a 256-instruction program (called boot kernel or loader kernel) into the processor's internal memory. The boot kernel program may be stored on an external PROM, a host processor, or another TigerSHARC processor. The boot type is selected via the processor's Boot Mode Select ($\overline{\texttt{BMS}}$) pin as described in "Boot Type Selection" on page 3-3. After the boot kernel loads, it executes itself and then loads the rest of the application program and data into the processor. The combination of the boot kernel and the application program comprises a boot-loadable file.

TigerSHARC processors support three booting modes: EPROM/flash, host, and link. The boot-loadable files for each of these modes pack the boot data into 32-bit instructions and use a DMA channel of the processor's DMA controller to boot-load the instructions.

Additionally, there are several no-boot modes, which do not require kernels.

- In EPROM/flash boot mode, the loader generates a PROM image that contains all project data and loader code. The project data is then stored in an 8-bit wide external EPROM. After reset, the processor performs a special booting scenario, reading the EPROM content through the processor's external port and initializing on-chip and off-chip memories.
- In host boot mode, the processor accepts boot data from a 32- or 64-bit synchronous microprocessor (host). The host writes a boot-loadable file to the processor's `AUTODMA` register through the processor's external port, one 32-bit word at a time. Once the last word is written, the processor takes over and runs the user code.

- In link port boot mode, the processor receives boot data via its link port from another TigerSHARC processor.

EE-174: ADSP-TS101S TigerSHARC Processor Boot Loader Kernels Operation and *E-200: ADSP-TS20x TigerSHARC Processor Boot Loader Kernels Operation* provide additional information about the loader. These EE notes are available from the Analog Devices Web site:

`http://www.analog.com/processors/processors/tigersharc/technicalLibrary/index.html.`

Boot Type Selection

To determine the boot mode, a TigerSHARC processor samples its Boot Mode Select ($\overline{\text{BMS}}$) pin. While the processor is held in reset, the $\overline{\text{BMS}}$ pin is an active input.

If $\overline{\text{BMS}}$ is sampled low a certain number of clock cycles after reset, EPROM/flash boot is selected and, after RESET goes high, $\overline{\text{BMS}}$ becomes an output, acting as EPROM chip select.

If $\overline{\text{BMS}}$ is sampled high after reset, the TigerSHARC processor is at an IDLE state, waiting for a host or link boot.

The 100K Ohm internal pull-down on $\overline{\text{BMS}}$ may not suffice, depending on the line loading. Thus, an additional external pull-down resistor may be necessary for the EPROM boot mode. If host or link boot is desired, $\overline{\text{BMS}}$ must be high and may be tied directly to the system power bus.

Boot Kernels

Upon completion of the DMA, in all boot modes, the boot-loading process continues by downloading the boot kernel into the processor memory. The boot kernel sets up and initializes the processor's memory. After initializing the rest of the system, the boot kernel overwrites itself.

You can build an `.LDR` file that includes or does not include a kernel. Use the `-NoKernel` command-line switch or the **No Kernel** option on the **Load** page to build without a kernel.

VisualDSP++ includes distinct kernel programs for each TigerSHARC processor. A boot kernel is loaded at reset into a memory segment, `seg_ldr`, which is defined in the `ADSP-TSxxx_Loader.ldf` file. The provided files are located in the `...\VisualDSP\TS\ldr` directory.

Table 3-1. TigerSHARC Boot Kernel Source Files

PROM Boot Kernel	Host Boot Kernel	Link Boot Kernel
Ts101_prom.asm	Ts101_host.asm	Ts101_link.asm
Ts201_prom.asm	Ts201_host.asm	Ts201_link.asm
Ts202_prom.asm	Ts202_host.asm	Ts202_link.asm
Ts203_prom.asm	Ts203_host.asm	Ts203_link.asm

Boot Kernel Modification

For most systems, some customization of the boot kernel is required. The operation of other tools (notably the C/C++ compiler) is influenced by loader usage.

For more information on boot kernel operations, refer to the comments in the corresponding boot kernel source files and application notes *EE-174: ADSP-TS101S TigerSHARC® Processor Boot Loader Kernels Operation* and *EE-200: ADSP-TS20x TigerSHARC Processor Boot Loader Kernels Operation*. The notes are at:

```
http://www.analog.com/processors/processors/tigersharc/
technicalLibrary/index.html
```

TigerSHARC Loader Guide

Loader operations depend on the loader options, which control how the loader processes executable files. You select features such as boot mode, boot kernel, and output file format via the loader options. 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. 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 TigerSHARC Loader Command Line" on page 3-6
- "Using VisualDSP++ Interface (Load Page)" on page 3-12

Using TigerSHARC Loader Command Line

The TigerSHARC loader uses the following command-line syntax.

For a single input file:

```
elfloader inputfile -proc processor [-switch …]
```

For multiple input files:

```
elfloader id2exe=inputfile1 id2exe=inputfile2 … -proc processor
  [-switch ...]
```

where:

- `inputfile`—Name of the executable file (`.DXE`) to be processed into a single boot-loadable file. An input file name can include the drive and directory.

  For multiprocessor or multi-input systems, specify multiple input `.DXE` files. Use the `id#exe=` switch, where `#` is the ID number (from 0 to 7) of the processor. Enclose long file names within straight quotes, "`long file name`".

- `-proc processor`—Part number of the processor (for example, `ADSP-TS101`) for which the loadable file is built.

- `-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. For a multi-input system, the loader processes the input files in the ascending order of the executable files from the `-exe#=` switch presented on the command line.

```
elfloader p0.dxe -proc ADSP-TS101 -bprom -fhex -l Ts101_prom.dxe
```

In the above example, the command line runs the loader with:

- `p0.dxe` – Identifies the executable file to process into a boot-loadable file. Note the absence of the `-o` switch causes the output file name to default to `p0.ldr`.
- `-proc ADSP-TS101` – Specifies ADSP-TS101 as the processor type.
- `-bprom` – Specifies EPROM booting as the boot type for the boot-loadable file.
- `-fhex` – Specifies Intel Hex-32 format for the boot-loadable file.
- `-l Ts101_prom.exe` – Specifies the boot kernel file to be used for the boot-loadable file.

```
elfloader id2exe=p0.dxe id2exe=p1.dxe -proc ADSP-TS101 -bprom
   -fhex -l Ts101_prom.dxe
```

In the above example, the command line runs the loader with:

- `p0.dxe` – Identifies the executable file for the processor with ID 0 to process into a boot-loadable file. Note the absence of the `-o` switch causes the output file name to default to `p0.ldr`.
- `p1.dxe` – Identifies the executable file for the processor with ID 1 to process into a boot-loadable file.
- `-proc ADSP-TS101` – Specifies ADSP-TS101 as the processor type.
- `-bprom` – Specifies EPROM booting as the boot type for the boot-loadable file.
- `-fhex` – Specifies Intel Hex-32 format for the boot-loadable file.
- `-l Ts101_prom.exe` – Specifies the boot kernel file to be used for the boot-loadable file.

File Searches

File searches are important in loader processing. The loader supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-11.

File Extensions

Some loader switches take a file name as an optional parameter. Table 3-2 lists the expected file types, names, and extensions. The loader takes files with extensions of `.DXE`, `.OVL`, and `.SM` but expects only those with extension `.DXE` in a command line on the **Load** page. The loader finds files with extensions of `.OVL` and `.SM` as it processes the associated `.DXE` file. The loader searches for `.OVL` and `.SM` files in the directory holding the `.DXE` files, in the directory specified in the `.LDF` file, or in the current directory. The `.OVL` extension is for overlaying files, and the `.SM` extension is for shared memory files.

Table 3-2. TigerSHARC File Extensions

Extension	File Description
`.DXE`	Loader input files, boot kernel files, and initialization files
`.LDR`	Loader output file
`.OVL`	Overlay files. The loader does not expect them on a command line.
`.SM`	Shared memory files. The loader does not expect them on a command line.

Command-Line Switches

A summary of the loader command-line switches appears in Table 3-3.

Table 3-3. TigerSHARC Loader Command-Line Switches

Switch	Description

`-bprom`
`-bhost`
`-blink`
	Prepares a boot-loadable file for the specified boot mode. Valid boot types include PROM, host, and link. If the `-b` switch does not appear on the command line, the default setting is `-bprom`. To use a custom kernel, the boot type selected with the `-b` switch must correspond to the boot kernel selected with the `-l` switch.

`-fhex`
`-fASCII`
`-fbinary`
`-fS1`
`-fS2`
`-fS3`
	Prepares a boot-loadable file in the specified format. Available format selections are: hex (Intel Hex-32), S1, S2, S3 (Motorola S-records), include, ASCII, and binary. Valid formats depend on the `-b` switch boot type selection.
	• For a PROM boot type, use a hex, S1, S2, S3, include, binary, or ASCII format.
	• For host or link booting, use ASCII or binary formats.
	If the `-f` switch does not appear on the command line, the default boot type format is hex for PROM, and ASCII for host or link.

`-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 target TigerSHARC processor, add the `-proc` switch to the command line. For example, type `elfloader-proc ADSP-TS101 -h` to obtain help for the ADSP-TS101S processor.

`-id#exe=`*`filename`*	Directs the loader to use the processor ID number for the corresponding executable file when producing a boot-loadable file for a EPROM- or host-boot multiprocessor system. Use this switch only to produce a boot-loadable file that boots multiple processors from a single EPROM. Valid # are `0`, `1`, `2`, `3`, `4`, `5`, `6`, and `7`. Do not use this switch for single-processor systems. For single-processor systems, use the executable file name as a parameter without a switch.
`-l` *`userkernele`*	Directs the loader to use the specified *`userkernel`* and to ignore the default boot kernel for the boot-loading routine in the output boot-loadable file. Note: The boot kernel file selected with this switch must correspond to the boot type selected with the `-b` switch). If the `-l` switch does not appear on the command line, the loader searches for a default boot kernel file in the installation directory (see "Boot Kernels" on page 3-4).

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

Switch	Description
`-nokernel`	Supports internal boot mode. The `-nokernel` switch directs the loader not to include the boot kernel code into the loader (`.LDR`) file.
`-o` *`filename`*	Directs the loader to use the specified *`filename`* as the name of the loader output file. If the *`filename`* is absent, the default name is the name of the input file with an `.LDR` extension.
`-p #`	Specifies the EPROM start address (hex format) for the boot-loadable file. If the `-p` switch does not appear on the command line, the loader starts the EPROM file at address `0x0` in the EPROM; this EPROM address corresponds to address `0x4000000` in a TigerSHARC processor.
`-proc` *`processor`*	Specifies the target processor. The *`processor`* can be one of the following: `ADSP-TS101`, `ADSP-TS201`, `ADSP-TS202`, or `ADSP-TS203`.
`-t #`	Sets the number of timeout cycles (#) as a maximum number of cycles the processor spends initializing external memory. Valid values range from `3` to `32765` cycles; `32765` is the default value. The timeout value is directly related to the number of cycles the processor locks the bus for boot-loading, instructing the processor to lock the bus for no more than 2X timeout number of cycles. When working with a fast host that cannot tolerate being locked out of the bus, use a relatively small timeout value.
`-v`	Outputs verbose loader messages and status information as the loader processes files.

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

Switch	Description
`-version`	Directs the loader to display its version information. Type `elf-loader -version` to display the version of the loader drive. Add the `-proc` switch, such as in `elfloader -proc ADSP-TS201 -version` to display version information for the loader drive and TigerSHARC loader.
`-si-revision #|none`	Provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` switch. The parameter 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`. Revision `0.1` is distinct from and "lower" than revision `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` 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 switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader generates an error.

Using VisualDSP++ Interface (Load Page)

When developing a **DSP loader file** project in VisualDSP++, modify the default option settings on the **Load** page of the **Projects Options** dialog box. For information specific to your target processor, refer to the VisualDSP++ online help for that processor.

VisualDSP++ invokes the `elfloader` utility to build the output file. Dialog box buttons and fields correspond to command-line switches and parameters (see Table 3-3 on page 3-9). Use the **Additional Options** box to enter options that have no dialog box equivalent.

4 LOADER FOR ADSP-2106X/21160 SHARC PROCESSORS

This chapter explains how the loader program (`elfloader.exe`) is used to convert executable (`.DXE`) files into boot-loadable files for ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, and ADSP-21160 SHARC processors.

Refer to "Introduction" on page 1-1 for the loader overview; the introductory material applies to all processor families. Refer to "Loader for ADSP-21161 SHARC Processors" on page 5-1 for information about ADSP-21161 processors. Refer to "Loader for ADSP-2126x/2136x SHARC Processors" on page 6-1 for information about ADSP-2126x and ADSP-2136x processors.

Loader operations specific to ADSP-2106x/21160 SHARC processors are detailed in the following sections.

- "ADSP-2106x/21160 Processor Booting" on page 4-2
  Provides general information about various booting modes, including information about boot kernels.
- "ADSP-2106x/21160 Processor Loader Guide" on page 4-25
  Provides reference information about the loader graphical user interface, command-line syntax, and switches.

ADSP-2106x/21160 Processor Booting

ADSP-2106x/21160 processors support four boot types (modes): EPROM, host, link port, and no-boot, described in Table 4-3 and Table 4-4 on page 4-5. Boot-loadable files for these modes pack boot data into 48-bit instructions and use an appropriate DMA channel of the processor's DMA controller to boot-load the instructions.

ADSP-2106x processors use DMA channel 6 (`DMAC6`) for booting. ADSP-21160 processors use `DMAC8` for link port booting and `DMAC10` for the host and EPROM booting.

- When booting from an EPROM through the external port, the ADSP-2106x/21160 processor reads boot data from an 8-bit external EPROM.
- When booting from a host processor through the external port, the ADSP-2106x/21160 processor accepts boot data from a 8- or 16-bit host microprocessor.
- When booting through the link port, the ADSP-2106x/21160 processor receives boot data as 4-bit wide data in link buffer 4.
- In no-boot mode, the ADSP-2106x/21160 processor begins executing instructions from external memory.

Software developers who use the loader should be familiar with the following operations.

- "Power-Up Booting Process" on page 4-3
- "Boot Mode Selection" on page 4-5
- "Boot Types" on page 4-7
- "Boot Kernels" on page 4-16
- "Interrupt Vector Table" on page 4-22

- "Multiprocessor EPROM Booting" on page 4-23
- "Processor ID Numbers" on page 4-24

Power-Up Booting Process

ADSP-2106x and ADSP-21160 processors include a hardware feature that boot-loads a small, 256-instruction program into the processor's internal memory after power-up or after the chip reset. These instructions come from a program called the boot kernel or the loader kernel. When executed, the boot kernel facilitates booting of user application code. The combination of the boot kernel and application code comprise the boot-loadable (`.LDR`) file.

At power-up, after the chip reset, the booting process includes the following steps.

1. Based on the boot type, an appropriate DMA channel is automatically configured for a 256-instruction (48-bit) transfer. This transfer boot-loads the boot kernel program into the processor memory.

   DMA channels used by the various processor models are shown in Table 4-1.

Table 4-1. ADSP-2106x/21160 Processor DMA Channels

Processor Model	PROM Booting	Host Booting	Link Booting
ADSP-21060	6 (See Table 4-8.)	6 (See Table 4-8.)	6
ADSP-21061			Not supported
ADSP-21062			6
ADSP-21065L	8 (`DMAC0` programs DMA channel 8; see Table 4-8)	8 (`DMAC0` programs DMA channel 8; see Table 4-8)	Not supported
ADSP-21160	10 (See Table 4-9.)	10 (See Table 4-9.)	8

2. The boot kernel runs and loads the application executable code and data.
3. The boot kernel overwrites itself with the first 256 words of the application at the end of the booting process. After that, the application executable code begins to execute the user application code from locations `0x20000` (ADSP-21060/61/62), `0x8000` (ADSP-21065L), and `0x40000` (ADSP-21160).

   The start addresses and reset vector addresses are summarized in Table 4-2.

Table 4-2. ADSP-2106x/21160 Processor Start Addresses

Processor Model	**Block 0 Start Address**	**Reset Vector Address**[1]
ADSP-21060	`0x20000`	`0x20004`
ADSP-21061	`0x20000`	`0x20004`
ADSP-21062	`0x20000`	`0x20004`
ADSP-21065L	`0x8000`	`0x8004`
ADSP-21160	`0x40000`	`0x40004`

1 The reset vector address must not contain a valid instruction since it is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location.

The boot type selection directs the system to prepare the appropriate boot kernel.

Boot Mode Selection

The state of various pins selects the processor boot mode (boot type). For ADSP-21060, ADSP-21061, ADSP-21062, and ADSP-21160 processors, refer to Table 4-3 and Table 4-4. For ADSP-21065L processors, refer to Table 4-5 and Table 4-6.

Table 4-3. ADSP-21060/061/062, and ADSP-21160 Boot Mode Pins

Pin	Type	Description
`EBOOT`	I	EPROM Boot. When `EBOOT` is high, the processor boot-loads from an 8-bit EPROM through the processor's external port. When `EBOOT` is low, the `LBOOT` and $\overline{\text{BMS}}$ pins determine the booting mode.
`LBOOT`	I	Link Boot. When `LBOOT` is high and `EBOOT` is low, the processor boots from another SHARC through the link port. When `LBOOT` is low and `EBOOT` is low, the processor boots from a host processor through the processor's external port.
`BMS`	I/O/T[1]	Boot Memory Select. When boot-loading from an EPROM (`EBOOT`=1 and `LBOOT`=0), this pin is an *output* and serves as the chip select for the EPROM. In a multiprocessor system, $\overline{\text{BMS}}$ is output by the bus master. When host-booting or link-booting (`EBOOT`=0), $\overline{\text{BMS}}$ is an *input* and must be high.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 4-4. ADSP-21060/061/062 and ADSP-21160 Boot Modes

EBOOT	LBOOT	`BMS`	Boot Mode
0	0	0 (Input)	No boot (processor executes from external memory)
0	0	1 (Input)	Host processor
0	1	0 (Input)	Reserved
0	1	1 (Input)	Link port
1	0	Output	EPROM ($\overline{\text{BMS}}$ is chip select)
1	1	x (Input)	Reserved

Table 4-5. ADSP-21065L Boot Mode Pins

Pin	Type	Description
BMS	I/O/T[1]	Boot Memory Select When BSEL is low, $\overline{\text{BMS}}$ is an input pin and selects between host boot mode and no boot mode. In no boot mode, the processor executes from external memory. For no boot mode, connect $\overline{\text{BMS}}$ to ground. For host boot mode, connect $\overline{\text{BMS}}$ to VDD. When BSEL is high, $\overline{\text{BMS}}$ is an output pin and the processor starts up in EPROM boot mode. Connect $\overline{\text{BMS}}$ to the EPROM's chip select.
BSEL	I	EPROM Boot Select Hardwire this signal; it is used for system configuration. When BSEL is high, the processor starts up in EPROM boot mode. The processor assumes the EPROM data bus is 8 bits wide. Connect BSEL to the processor data bus in LSB alignment. When BSEL is low, $\overline{\text{BMS}}$ determines the booting mode. Connect BSEL to ground.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 4-6. ADSP-21065L Boot Modes

BSEL	$\overline{\text{BMS}}$	Description
0	1	No boot mode. The processor executes from external memory at location 0x20004.
0	1	Host boot mode. The processor defaults to an 8-bit host bus width.
1	Output	EPROM boot mode. The processor assumes an 8-bit EPROM data bus width. Connect to the data bus in LSB alignment.

Boot Types

ADSP-2106x/21160 processors support these booting modes: EPROM, host, and link. The following sections describe each of the booting modes.

- "EPROM Booting" on page 4-7
- "Host Booting" on page 4-11
- "Link Booting" on page 4-15
- "No-Boot Mode" on page 4-16

For multiprocessor booting, refer to "Multiprocessor EPROM Booting" on page 4-23.

EPROM Booting

The ADSP-2106x/21160 processor is configured for EPROM booting through the external port when the `EBOOT` pin is high and the `LBOOT` pin is low. These settings cause the $\overline{\texttt{BMS}}$ pin to become an output, serving as chip select for the EPROM. Table 4-7 lists all PROM-to-processor connections.

Table 4-7. PROM Connections to ADSP-2106x/21160 Processors

Processor Model	Connection
ADSP-21060/61/62	PROM/EPROM connected to `DATA23–16` pins
ADSP-21065L	PROM/EPROM connected to `DATA0–7` pins
ADSP-21160	PROM/EPROM connected to `DATA32–39` pins
ADSP-21xxx	Address pins of PROM connect to lowest address pins of any processor
ADSP-21xxx	Chip select connect to the $\overline{\text{BMS}}$ pin
ADSP-21060/61/62/65L	Output enable connect to the $\overline{\text{RD}}$ pin
ADSP-21160	Output enable connect to $\overline{\text{RDH}}$ pin

During reset, the ACK line is pulled high internally with a 2K ohm equivalent resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the ACK line during booting or at any other time.

The DMA channel parameter registers are initialized at reset for EPROM booting as shown in Table 4-8 and Table 4-9. The count is initialized to 0x0100 to transfer 256 words to internal memory. The external count register (ECx), which is used when external addresses (BMS space) are generated by the DMA controller, is initialized to 0x0600 (0x100 words at six bytes per word).

Table 4-8. DMA Settings for ADSP-2106x EPROM Booting

DMA Setting		**Processor Model**	
		ADSP-21060/61/62	**ADSP-21065L**
BMS space		4M x 8-bit	8M x 8-bit
DMA Channel		DMAC6 = 0x2A1	DMAC0 = 0x2A1
II6	IIEP0	0x20000	0x8000
IM6	IMEP0	0x1 (Implied)	0x1 (Implied)
C6	CEP0	0x100	0x100
EI6	EIEP0	0x80 0000	0x40 0000
EM6	EMEP0	0x1 (Implied)	0x1 (Implied)
EC6	ECEP0	0x600	0x600
IRQ Vector		0x20040	0x8040

Table 4-9. DMA Settings for ADSP-21160EPROM Booting

DMA Setting	**ADSP-21160 Processor**
BMS space	8M x 8-bit
DMA Channel	DMAC10 = 0x4A1

Table 4-9. DMA Settings for ADSP-21160EPROM Booting (Cont'd)

DMA Setting	ADSP-21160 Processor
II10	0x40000
IM10	0x1 (Implied)
C10	0x100
EI10	0x800000
EM10	0x1 (Implied)
EC10	0x600
IRQ Vector	0x40050

After the processor's RESET pin goes inactive on start-up, a SHARC system configured for EPROM booting undergoes the following boot-loading sequence.

1. The processor $\overline{\text{BMS}}$ pin becomes the boot EPROM chip select.

2. The processor goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to the processor reset vector address (refer to Table 4-2 on page 4-4).

3. The DMA controller reads 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them into internal memory (low-to-high byte packing order) until the 256 words are loaded.

4. The DMA parameter registers for appropriate DMA channels are initialized, as shown in Table 4-8 and Table 4-9. The external port DMA channel (6 or 10) becomes active following reset; it is initialized to set external port DMA enable and selects DTYPE for instruction words. The packing mode bits (PMODE) are ignored, and 48- to 8-bit packing is forced with least significant word first. The UBWS and UBWM fields of the WAIT register are initialized to generate six wait states for the EPROM access in unbanked external memory space.

5. The processor begins 8-bit DMA transfers from the EPROM to internal memory using the following external port data bus lines:

   `D23–D16` for ADSP-21060/61/62 processors

   `D0–D7` for ADSP-21065L processors

   `D32–D39` for ADSP-21160 processors

6. Data transfers begin and increment after each access. The external address lines (`ADDR 31–0`), start at:

   `0x40 0000` for ADSP-21060/61/62 processors

   `0x00 0000` for ADSP-21065L processors

   `0x80 0000` for ADSP-21160 processors

7. The processor $\overline{\mathtt{RD}}$ pin asserts as in a normal memory access, with six wait states (seven cycles).

8. After finishing DMA transfers to load the boot kernel into the processor, the `BS0` bit is cleared in the `SYSCON` register, deactivating $\overline{\mathtt{BMS}}$ and activating normal external memory select.

   The boot kernel uses three copies of `SYSCON`—one that contains the original value of `SYSCON`, a second that contains `SYSCON` with the `BS0` bit set (allowing the processor to gain access to the boot EPROM), and a third with the `BS0` bit cleared.

   When `BS0=1`, the EPROM packing mode bits in the `DMACx` control register are ignored and 8- to 48-bit packing is forced. (8-bit packing is available only during EPROM booting or when `BS0` is set.) When an external port DMA channel is being used in conjunction with the `BS0` bit, none of the other three channels may be used. In this mode, $\overline{\mathtt{BMS}}$ is not asserted by a core processor access but only by a DMA transfer. This allows the boot kernel to perform other external accesses to non-boot memory.

The EPROM is automatically selected by the $\overline{\text{BMS}}$ pin after reset, and other memory select pins are disabled. The processor's DMA controller reads the 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them to internal memory until 256 words have been loaded. The Master DMA internal and external count registers (CX and ECX) decrement after each EPROM transfer. When both counters reach zero, DMA transfer has stopped and RTI returns the program counter to the address where kernel begins.

To EPROM boot a single-processor system, include the executable on the command-line without a switch. Do not use the -id#exe switch with ID=0 (see "Processor ID Numbers" on page 4-24).

The WAIT register UBWM (used for EPROM booting) is initialized at reset to both internal wait and external acknowledge required. The internal keeper-latch on the ACK pin initially holds acknowledge high (asserted). If acknowledge is driven low by another device during an EPROM boot, the keeper latch may latch acknowledge low.

The processor views the deasserted (low) acknowledge as a hold off from the EPROM. In this condition, wait states are continually inserted, preventing completion of the EPROM boot. When writing a custom boot kernel, change the WAIT register early within the boot kernel so UBWM is set to internal wait mode (01).

Host Booting

ADSP-2106x/21160 processors accept data from a 8- and 16-bit host microprocessor (or other external device) through the external port EPB0 and pack boot data into 48-bit instructions using an appropriate DMA channel. The host is selected when the EBOOT and LBOOT inputs are low and $\overline{\text{BMS}}$ is high. Configured for host booting, the processor enters the slave mode after reset and waits for the host to download the boot program. Table 4-10 lists host connections to processors.

Table 4-10. Host Connections to ADSP-2106x/21160 Processors

Processor	Connection/Data Bus Pins
ADSP-21060/61/62	Host connected to DATA47–16 or DATA31–16 pins (based on HPM bits)
ADSP-21065L	Host connected to DATA31–0 or DATA15–0 or DATA7–0 pins (based on HBW-bits)
ADSP-21160	Host connected to DATA63–32 or DATA47–31 pins (based on HPM-bits)
ADSP-21060/61/62/65L	ADSP-21065L host address to IOP registers only
ADSP-21160	ADSP-21160 host address to IOP registers and internal memory

After reset, the processor goes into an idle stage with:

- PC set to address 0x20004 (ADSP-21060/61/62 processors)
- PC set to address 0x8004 (ADSP-21065L processors)
- PC set to address 0x40004 (ADSP-21160 processors)

The parameter registers for the external port DMA channel (0, 8, or 10) are initialized as shown in Table 4-8 and Table 4-9, except that registers EIx, EMx and ECx are not initialized and no DMA transfers start.

The DMA Channel Control register (DMAC6 for ADSP-21060/61/62 processors, DMAC0 for ADSP-21065L processors, or DMAC10 for ADSP-21160 processors) is initialized, which allows external port DMA enable and selects DTYPE for instruction words, PMODE for 16- to 48-bit word packing (8- to 48-bit for ADSP-21065L processors), and least significant word first.

Because the host processor is accessing the EPB0 external port buffer, the HPM host packing mode bits of the SYSCON register must correspond to the external bus width specified by the PMODE bits of DMACx control register.

For a different packing mode, the host must write to `DMACx` and `SYSCON` to change the `PMODE` and `HBW` (`HPW` for ADSP-21065L processors) setting. The host boot file created by the loader requires the host processor to perform the following sequence of actions.

1. The host initiates the synchronous booting operation (synchronous not valid for ADSP-21065L processors) by asserting the processor $\overline{\texttt{HBR}}$ input pin, informing the processor that the default 8-/16-bit bus width is used. The host may optionally assert the $\overline{\texttt{CS}}$ chip select input to allow asynchronous transfers.

2. After the host receives the $\overline{\texttt{HBG}}$ signal (and `ACK` for synchronous operation or `READY` for asynchronous operation) from the processor, the host can start downloading instructions by writing directly to the external port DMA buffer 0 or the host can change the reset initialization conditions of the processor by writing to any of the `IOP` control registers. The host must use data bus pins as shown in Table 4-10.

3. The host continues to write 16-bit words (8-bit for ADSP-21065L) to `EPB0` until the entire program is boot-loaded. The host must wait between each host write to external port DMA buffer 0.

After the host boot-loads the first 256 instructions (boot kernel), the initial DMA transfers stop, and the boot kernel:

1. Activates external port DMA channel interrupt (`EP0I`), stores the `DMACx` control setting in `R2` for later restore, clears `DMACx` for new setting, and sets the `BUSLCK` bit in the `MODE2` register to lock out the host.

2. Stores the `SYSCON` register value in `R12` for restore.

3. Enables interrupts and nesting for DMA transfer, sets up the `IMASK` register to allow DMA interrupts, and sets up the `MODE1` register to enable interrupts and allow nesting.

4. Loads the DMA Control register with `0x00A1` and sets up its parameters to read the data word by word from external buffer 0.

   Each word is read into the reset vector address (refer to Table 4-2 on page 4-4) for dispatching. The data through this buffer has a structure of boot section which could include more than one initialization block.

5. Clears the `BUSLCK` bit in the `MODE2` register to let the host write in the external buffer 0 right after the appropriate DMA channel is activated.

   For information on the data structure of the boot section and initialization, see "Blocks and Block Headers" on page 4-17.

6. Loads the first 256 words of target the executable file during the final initialization stage, and then the kernel overwrites itself.

The final initialization works the same way as with EPROM booting, except that the `BUSLCK` bit in the `MODE2` register is cleared to allow the host to write to the external port buffer.

The default boot kernel for host booting assumes `IMDW` is set to `0` during boot-loading, except during the final initialization stage. When using any power-up booting mode, the reset vector address (refer to Table 4-2 on page 4-4) must not contain a valid instruction because it is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location.

If the boot kernel initializes external memory, create a custom boot kernel that sets appropriate values in the `SYSCON` and `WAIT` register. Be aware that the value in the DMA channel register is non-zero, and `IMASK` is set to allow DMA channel register interrupts. Because the DMA interrupt remains enabled in `IMASK`, this interrupt must be cleared before using the DMA channel again. Otherwise, unintended interrupts may occur.

A master SHARC processor may boot a slave SHARC processor by writing to its DMACx control register and setting the packing mode (PMODE) to 00. This allows instructions to be downloaded directly without packing. The wait state setting of 6 on the slave processor does not affect the speed of the download since wait states affect bus master operation only.

Link Booting

Link booting is supported on all SHARC processors except ADSP-21061 and ADSP-21065L processors.

When you link-boot ADSP-2106x/21160 SHARC processors, the processor receives data from 4-bit link buffer 4 and packs boot data into 48-bit instructions using the appropriate DMA channels (DMA channel 6 for ADSP-2106x processors, DMA channel 8 for ADSP-21160 processors).

Link mode is selected when the EBOOT is low and LBOOT and $\overline{\text{BMS}}$ are high. The external device must provide a clock signal to the link port assigned to link buffer 4. The clock can be any frequency, up to a maximum of the processor clock frequency. The clock falling edges strobe the data into the link port. The most significant 4-bit nibble of the 48-bit instruction must be downloaded first. The link port acknowledge signal generated by the processor can be ignored during booting since the link port cannot be preempted by another DMA channel.

Link booting is similar to host booting—the parameter registers (IIx and Cx) for DMA channels are initialized to the same values. The DMA channel 6 control register (DMAC6) is initialized to 0x00A0, and the DMA channel 10 control register (DMAC10) is initialized to 0x100000. This disables external port DMA and selects DTYPE for instruction words. The LCTL and LCOM link port control registers are overridden during link booting to allow link buffer 4 to receive 48-bit data.

After booting completes, the IMASK remains set, allowing DMA channel interrupts. This interrupt must be cleared before link buffer 4 is again enabled; otherwise, unintended link interrupts may occur.

Refer to "Host Booting" on page 4-11 for more information on booting process.

No-Boot Mode

No-boot mode causes the processor to start fetching and executing instructions at address `0x400004` (ADSP-2106x), `0x20004` (ADSP-21065L), and `0x800004` (ADSP-21160) in external memory space. All DMA control and parameter registers are set to their default initialization values. The loader is not intended to support no-boot mode.

Boot Kernels

The boot-loading process starts with a transfer of the boot kernel program into the processor memory. The boot kernel sets up the processor and loads boot data. After the boot kernel finishes initializing the rest of the system, the boot kernel loads boot data over itself with a final DMA transfer.

Boot kernels are loaded at reset into program segment `seg_ldr`, which is defined in `06x_ldr.ldf` for ADSP-2106x processors, `065L_ldr.ldf` for ADSP-21065L processors, or in `160_ldr.ldf` for ADSP-21160 processors. This file is stored in the processor tools installation directory in ...`\21k\ldr` for ADSP-2106x processors and ...`\211xx\ldr` for ADSP-21160 processors.

The default boot kernel files shipped with VisualDSP++ are listed in Table 4-11.

Table 4-11. ADSP-2106x/21160 Default Boot Kernel Files

Processor Model	PROM Booting	Link Booting	Host Booting
ADSP-21060	`060_prom.asm`	`060_link.asm`	`060_host.asm`
ADSP-21065L	`065L_prom.asm`		`065L_host.asm`
ADSP-21160	`160_prom.asm`	`160_link.asm`	`160_host.asm`

Once the kernel has been loaded successfully into the processor, the kernel follows the following sequence.

1. Each boot kernel begins with general initializations for the `DAG` registers, appropriate interrupts, processor ID information, and various `SDRAM` or `WAIT` state initializations.

2. Once the boot kernel has finished the task of initializing the processor, the kernel initializes processor memory, both internal and external, with user application code.

Blocks and Block Headers

The structure of a loader file enables the boot kernel to load code and data, block by block. In the loader file, each block of code or data is preceded by a block header, which describes the block —length, placement, and data or instruction type. After the block header, the loader outputs the block body, which includes the actual data or instructions for placement in the processor memory. The loader, however, does not output a block body if the actual data or instructions are all zeros in value. This type of block is called a zero block. Table 4-12 describes the block header and block body format.

Table 4-12. Boot Block Format

Block Header	First word	Bits 16–47 are not used. Bits 0–15 define the type of data block (tag).
	Second word	Bits 16–47 are the start address of the block. Bits 0–15 are the word count for the block.
Block Body (if not a zero block)		Word 1 (48 bits) Word 2 (48 bits)

The loader identifies the data type in the block header with a 16-bit tag. The tag precedes the block. Each type of initialization has a unique tag number (tag number—initialization type) as shown in Table 4-13.

Table 4-13. ADSP-2106x/21160 Processor Loader Block Tags

Tag Number	Block Type	Tag Number	Block Type
0x0000	final init	0x000A	zero pm48
0x0001	zero dm16	0x000B	init pm16
0x0002	zero dm32	0x000C	init pm32
0x0003	zero dm40	0x000E	init pm48
0x0004	init dm16	0x000F	zero dm64 (21160 only)
0x0005	init dm32	0x0010	init dm64 (21160 only)
0x0007	zero pm16	0x0011	zero pm64 (21160 only)
0x0008	zero pm32	0x0012	init pm64 (21160 only)
0x0009	zero pm40		

The kernel enables the boot port (external or link) to read the block header. After reading information from the block header, the kernel places the body of the block in the appropriate place in memory if the block has a block body, or initializes in the appropriate place with zero values in the memory if it is a zero block.

The final section, which is identified by a tag of `0x0`, is called the finial initialization section. This section has self-modifying code that, when executed, facilitates a DMA over the kernel, replacing it with the user application code that actually belongs in that space at run time. The final initialization code also takes care of interrupts and returns the processor registers, such as `SYSCON` and `DMAC` or `LCTL`, to their default values.

When the loader detects the final initialization tag, it reads the next 48-bit word. This word indicates the instruction to load into the 48-bit `PX` register after the boot kernel finishes initializing memory.

The boot kernel requires that the interrupt, External Port (or Link Port address, depending on the boot type) contains an `RTI` instruction. This `RTI` is inserted automatically by the loader to guarantee that the kernel executes from the reset vector, once the DMA that overwrites the kernel, is complete. A last remnant of the kernel code is left at the reset vector location to replace the `RTI` with the user's intended code. Because of this last kernel remnant, user application code should not use the first location of the reset vector. This first location should be a `NOP` or `IDLE` instruction. The kernel automatically completes, and the Program Controller begins sequencing the user application code at the second location in the processor reset vector space.

When the boot process is complete, the processor automatically executes the user application code. The only remaining evidence of the boot kernel is at the first location of the interrupt vector. Almost no memory is sacrificed to the boot code.

Boot Kernel Modification and Loader Issues

Some systems require boot kernel customization. The operation of other tools (such as the C/C++ compiler) is influenced by whether the loader is used.

When producing a boot-loadable file, the loader reads a processor executable file and uses information in it to initialize the memory. However, the loader cannot determine how the processor `SYSCON` and `WAIT` registers are to be configured for external memory loading in the system.

If you modify the boot kernel by inserting values for your system, you must rebuild it before generating the boot-loadable file. The boot kernel contains default values for `SYSCON`. The initialization code can be found in the comments in the boot kernel source file.

After modifying the boot kernel source file, rebuild the boot kernel (`.DXE`) file. Do this from the VisualDSP++ IDDE (refer to VisualDSP++ online Help for details), or rebuild the boot kernel file from the command line.

When using VisualDSP++, specify the name of the modified kernel executable in the **Kernel file** box on the **Load** page of the **Project Options** dialog box.

If you modify the boot kernel for EPROM, host, or link booting modes, ensure that the `seg_ldr` memory segment is defined in the `.LDF` file. Refer to the source of this segment in the `.LDF` file located in the `...\21k\ldr\` or (`...\211xx\ldr\`) directory.

The loader generates a warning when vector address (`0x20004` for ADSP-21060/61/62 processors, `0x40004` for ADSP-21160 processors, or `0x8004` for ADSP-21065L processors) does not contain `NOP` or `IDLE`. Because the boot kernel uses this address for the first location of the reset vector during the boot-load process, avoid placing code at this address. When using any of the processor's power-up booting modes, ensure that this address does not contain a critical instruction, because this address is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location.

The boot kernel project can be rebuilt from the VisualDSP++ IDDE. The command-line can also be used to rebuild various default boot kernels for ADSP-2106x/21160 processors.

EPROM Booting. The default boot kernel source file for the ADSP-2106x EPROM booting is `060_prom.asm`. Copy this file to `my_prom.asm` and modify it to suit your system. Then, use the following commands to rebuild the boot kernel.

```
easm21k -21060 my_prom.asm
```

or

```
easm21k -proc ADSP-21060 my_prom.asm
linker -T 060_ldr.ldf my_prom.doj
```

Host Booting. The default boot kernel source file for the ADSP-2106x host booting is `060_host.asm`. Copy this file to `my_host.asm` and modify it to suit your system. Then use the following commands to rebuild the boot kernel.

```
easm21k -21060 my_host.asm
```

or

```
easm21k -proc ADSP-21060 my_host.asm
linker -T 060_ldr.ldf my_host.doj
```

Link Port Booting. The default boot kernel source file for the ADSP-2106x link port booting is `060_link.asm`. Copy this file to `my_link.asm` and modify it to suit your system. Then use the following commands to rebuild the boot kernel.

```
easm21k -21060 my_link.asm
```

or

```
easm21k -proc ADSP-21060 my_link.asm
linker -T 060_ldr.ldf my_link.doj
```

Rebuilding Boot Kernels

To rebuild the PROM boot kernel for ADSP-21065L processors, use these commands:

```
easm21k -21065L my_prom.asm
```

or

```
easm21k -proc ADSP-21065L my_prom.asm
linker -T 065L_ldr.ldf my_prom.doj
```

To rebuild the PROM boot kernel for ADSP-21160 processors, use these commands:

```
easm21k -21160 my_prom.asm
```

or

```
easm21k -proc ADSP-21160 my_prom.asm
linker -T 160_ldr.ldf my_prom.doj
```

Interrupt Vector Table

If a ADSP-2106x/21160 SHARC processor is booted from an external source (EPROM, host, or another SHARC processor), the interrupt vector table is located in internal memory. If, however, the processor is not booted and executes from external memory, the vector table must be located in external memory.

The `IIVT` bit of the `SYSCON` control register can be used to override the booting mode in determining where the interrupt vector table is located. If the processor is not booted (no-boot mode), setting `IIVT` to `1` selects an internal vector table, and setting `IIVT` to `0` selects an external vector table. If the processor is booted from an external source (any mode other than no-boot mode), `IIVT` has no effect. The `IIVT` default initialization value is `0`.

Refer to *EE-56: Tips & Tricks on the ADSP-2106x EPROM and HOST bootloader*, *EE-189: Link Port Tips and Tricks for ADSP-2106x and ADSP-2116x*, and *EE-77: SHARC Link Port Booting* on the Analog Devices Web site for more information.

Multiprocessor EPROM Booting

Currently, the loader generates single-processor loader files for host and link port booting. As a result, the loader supports multiprocessor EPROM booting only. The application code must be modified to properly set up multiprocessor booting in host and link port booting modes.

The loader can produce boot-loadable files that permit the ADSP-2106x/21160 SHARC processors in a multiprocessor system to boot from a single EPROM. In such a system, the $\overline{\text{BMS}}$ signals from each SHARC processor are OR'ed together to drive the chip select pin of the EPROM. Each processor boots in turn, according to its priority. When the last processor finishes booting, it must inform the processors to begin program execution.

Besides taking turns when booting, EPROM booting of multiple processors is similar to single-processor EPROM booting.

When booting a multiprocessor system through a single EPROM:

- Connect all $\overline{\text{BMS}}$ pins to EPROM.
- Processor with `ID#1` boots first. The other processors follow.
- The EPROM boot kernel accepts multiple `.DXE` files and reads the `ID` field in `SYSTAT` to determine which area of EPROM to read.
- All processors require a software flag or hardware signal (`FLAG` pins) to indicate that booting is complete.

When booting a multiprocessor system through an external port:

- The host can use the host interface.
- A SHARC processor that is EPROM-, host-, or link-booted can boot the other processors through the external port (host boot mode).

For multiprocessor EPROM booting, select the **Multiprocessor** check box on the **Load** page of the **Project Options** dialog box or specify the `-id#exe` switch on the loader command line. These options specify the executable file targeted for a specific processor.

Do not use the `-id#exe` switch to EPROM-boot a single processor whose `ID` is `0`. Instead, name the executable file on the command line without a switch. For a single processor with `ID=1`, use the `-id1exe=` switch.

Processor ID Numbers

A single-processor system requires only one input (`.DXE`) file without any prefix and suffix to the input file name, for example:

```
elfloader -proc ADSP-21060 -bprom Input.dxe
```

A multiprocessor system requires a distinct processor ID number for each input file on the command line. A processor ID is provided via the `-id#exe=`*`filename`*`.dxe` switch, where `#` is `1` to `6`.

In the following example, the loader processes the input file `Input1.dxe` for the processor with an ID of `1` and the input file `Input2.dxe` for the processor with an ID of `2`.

```
elfloader -proc ADSP-21060 -bprom -id1exe=Input1.dxe
          -id2exe=Input2.dxe
```

If the executable for the `#` processor is identical to the executable of the *`N`* processor, the output loader file contains only one copy of the code from the input file.

```
elfloader -proc ADSP-21060 -bprom -id1exe=Input.dxe -id2exe=1
```

The loader points the `id(2)exe` loader jump table entry to the `id(1)exe` image, effectively reducing the size of the loader file.

ADSP-2106x/21160 Processor Loader Guide

Loader operations depend on the loader options, which control how the loader processes executable files. You select features such as boot mode, boot kernel, and output file format via the loader options. 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.

For information specific to the ADSP-2106x/21160 processor, refer to the VisualDSP++ online help for that processor. When you open the **Load** page, the default loader settings for the selected processor are already set. Use the **Additional Options** box to enter options that have no dialog box equivalent.

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

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

- "Using the ADSP-2106x/21160 Loader Command Line" on page 4-26
- "Using the VisualDSP++ Interface (Load Page)" on page 4-31

Using the ADSP-2106x/21160 Loader Command Line

Use the following syntax for the SHARC loader command line.

```
elfloader inputfile -proc processor -switch [-switch …]
```

where:

- `inputfile` – Name of the executable file (`.DXE`) to be processed into a single boot-loadable file. An input file name can include the drive and directory. Enclose long file names within straight quotes, “`long file name`”.
- `-proc processor` – Part number of the processor (for example, `-proc ADSP-21062`) for which the loadable file is built. The `-proc` switch is mandatory.
- `-switch …` – One or more optional switches to process. Switches select operations and boot modes for the loader. A list of all switches and their descriptions appear in Table 4-15 on page 4-28.

Command-line switches are not case-sensitive and can be placed on the command line in any order.

The following command line,

```
elfloader p0.dxe -bprom -fhex -l 060_prom.dxe -proc ADSP-21060
```

runs the loader with:

- `p0.dxe` – Identifies the executable file to process into a boot-loadable file. The absence of the `-o` switch causes the output file name to default to `p0.ldr`.
- `-bprom` – Specifies EPROM booting as the boot type for the boot-loadable file.
- `-fhex` – Specifies Intel hex-32 format for the boot-loadable file.

- `-l 060_prom.exe` – Specifies `060_prom.exe` as the boot kernel file to be used in the boot-loadable file.
- `-proc ADSP-21060` – Identifies the processor model as ADSP-21060.

File Searches

File searches are important in loader processing. The loader supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-11.

File Extensions

Some loader switches take a file name as an optional parameter. Table 4-14 lists the expected file types, names, and extensions.

Table 4-14. File Extensions

Extension	File Description
`.DXE`	Input executable files and boot kernel files. The loader recognizes overlay memory files (`.OVL`) and shared memory files (`.SM`), but does not expect these files on the command line. Place `.OVL` and `.SM` files in the same directory as the `.DXE` file that refers to them. The loader finds the files when processing the `.DXE` file. The `.OVL` and `.SM` files may also be placed in the `.OVL` and `.SM` file output directory specified in the `.LDF` file or current working directory.
`.LDR`	Loader output file.

Loader Command-Line Switches

Table Table 4-15 is a summary of the ADSP-2106x and ADSP-21160 loader switches.

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description

`-bprom`
`-bhost`
`-blink`
`-bJTAG`
	Specifies the boot mode. The `-b` switch directs the loader to prepare a boot-loadable file for the specified boot mode. Valid boot modes (boot types) include PROM, host, and link.
	For ADSP-21020 processors, JTAG is the only permitted boot mode.
	If `-b` does not appear on the command line, the default is `-bprom`.
	To use a custom boot kernel, the boot type selected with the `-b` switch must correspond to the boot kernel selected with the `-l` switch. Otherwise, the loader automatically selects a default boot kernel based on the selected boot type (see "Boot Kernels" on page 4-16).

`-c`*`address`*	Custom option. This switch directs the loader to use the specified address. Valid addresses are: • `20004` and `20040` for ADSP-2106x processors • `8004` and `8040` for ADSP-21065L processors • `40000` and `40050` for ADSP-21160 processors The loader obtains the proper address even when this switch is absent from the command line.
`-e` *`filename`*	Except shared memory. The `-e` switch omits the specified shared memory (`.SM`) file from the output loader file. Use this option to omit the shared parts of the executable file from multiprocessor boot files. To omit multiple `.SM` files, repeat the switch and parameter multiple times on the command line. For example, to omit two files, use: `-e fileA.SM -e fileB.SM`.

`-fhex`
`-fASCII`
`-fbinary`
`-finclude`
`-fS1`
`-fS2`
`-fS3`
	Specifies the format of the boot-loadable file (Intel hex-32, ASCII, S1, S2, S3, binary, and include). If the `-f` switch does not appear on the command line, the default boot file format is Intel hex-32 for PROM, and ASCII for host or link.
	Available formats depend on the boot type selection (`-b` switch):
	• For PROM boot type, select a hex, ASCII, S1, S2, S3, or include format.
	• For host or link boot type, select an ASCII, binary, or include format.

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description

`-h`
or
`-help`
	Command line help. Outputs the list of command-line switches to standard output and exits. Type `elfloader -proc ADSP-21xxx` -h, where `xxx` is `060`, `061`, `062`, `065L`, or `160` to obtain help for SHARC processors. By default, the `-h` switch alone provides help for the loader driver.

`-id#exe=`*`filename`*	Specifies the processor ID. The `-id#exe` switch directs the loader to use the processor ID (#) for the corresponding executable file (*`filename`*) when producing a boot-loadable file for EPROM booting of a multiprocessor system. This switch is used only to produce a boot-loadable file that boots multiple processors from a single EPROM. Valid values for # are `1`, `2`, `3`, `4`, `5`, and `6`. Do not use this switch for single-processor systems. For single-processor systems, use *`filename`* as a parameter without a switch. For more information, refer to "Processor ID Numbers" on page 4-24.
`-id#ref=`*`N`*	Points the processor ID (*`N`*) loader jump table entry to the ID (#) image. If the executable file for the (#) processor is identical to the executable of the (*`N`*) processor, the switch can be used to set the PROM start address of the processor with ID of # to be the same as for the processor with ID of *`N`*. This effectively reduces the size of the loader file by providing a single copy of an executable to two or more processors in a multiprocessor system. For more information, refer to "Processor ID Numbers" on page 4-24.
`-l` *`kernelfile`*	Directs the loader to use the specified *`kernelfile`* as the boot-loading routine in the output boot-loadable file. The boot kernel selected with this switch must correspond to the boot type selected with the `-b` switch. If the `-l` switch does not appear on the command line, the loader searches for a default boot kernel file. Based on the boot type (`-b` switch), the loader searches in the processor-specific loader directory for the boot kernel file as described in "Boot Kernels" on page 4-16.
`-o` *`filename`*	Directs the loader to use the specified *`filename`* as the name for the loader output file. If not specified, the default name is *`inputfile`*`.ldr`.

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description
`-p`*`address`*	PROM start address. Places the boot-loadable file at the specified address in the EPROM. If the `-p` switch does not appear on the command line, the loader starts the EPROM file at address `0x0`; this EPROM address corresponds to `0x800000` on ADSP-21060/21061/21062 processors, `0x400000` on ADSP-21064 processors, and `0x800000` for ADSP-21160 processors.
`-proc` *`processor`*	Specifies the processor. This a mandatory switch. The *`processor`* is one of the following: `ADSP-21060`, `ADSP-21061`, `ADSP-21062`, `ADSP-21065L`, `ADSP-21160`
`-si-revision #|none`	Provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` switch. The parameter 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`. Revision `0.1` is distinct from and “lower” than revision `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` 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 switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader generates an error.
`-t#`	(Host boot only) Specifies timeout cycles; for example, `-t100`. Limits the number of cycles that the processor spends initializing external memory with zeros. Valid timeout values # range from `3` to `32765` cycles; `32765` is the default. The # is directly related to the number of cycles the processor locks the bus for boot-loading, instructing the processor to lock the bus for no more than two times the timeout number of cycles. When working with a fast host that cannot tolerate being locked out of the bus, use a relatively small timeout value.

Table 4-15. ADSP-2106x/21160 Loader Command-Line Switches

Switch	Description
`-use32bit Tags for External Memory Block`	Directs the loader to treat the external memory sections as 32-bit sections as specified in the `.LDF` file and does not pack them into 48-bit sections before processing. This option is useful if the external memory sections are packed by the linker and do not need the loader to pack them again.
`-v`	Outputs verbose loader messages and status information as the loader processes files.
`-version`	Directs the loader to show its version information. Type `elfloader -version` to display the version of the loader drive. Add the `-proc` switch, for example, `elfloader -proc ADSP-21062 -version` to display version information of both loader drive and SHARC loader.

Using the VisualDSP++ Interface (Load Page)

After selecting a **Loader file** as the target **Type** from the **Project** page in VisualDSP++, modify the default options from the **Load** page (also called **Load** property page) and then click **OK** to save your selections. Selecting **Build Project** from the **Project** menu generates a loader file. For information relative to a specific processor, refer to the VisualDSP++ online help for that processor.

VisualDSP++ invokes the `elfloader` utility to build the output file. The **Load** page buttons and fields correspond to loader command line switches and parameters (see Table 4-15 on page 4-28). Use the **Additional Options** box to enter options that do not have dialog box equivalents.

For ADSP-21020 DSPs, the only permitted boot mode is JTAG: **-bJTAG** is automatically entered in the **Additional Options** box.

5 LOADER FOR ADSP-21161 SHARC PROCESSORS

This chapter explains how the loader program (`elfloader.exe`) is used to convert executable (`.DXE`) files into boot-loadable files for ADSP-21161 SHARC processors.

Refer to "Introduction" on page 1-1 for the loader overview; the introductory material applies to all processor families. Refer to "Loader for ADSP-2106x/21160 SHARC Processors" on page 4-1 for information about ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, and ADSP-21160 processors. Refer to "Loader for ADSP-2126x/2136x SHARC Processors" on page 6-1 for information about ADSP-2126x and ADSP-2136x processors.

Loader operations specific to ADSP-21161 SHARC processors are detailed in the following sections.

- "ADSP-21161 Processor Booting" on page 5-2
  Provides general information about various booting modes, including information about boot kernels.

- "ADSP-21161 Processor Loader Guide" on page 5-24
  Provides reference information about the loader graphical user interface, command-line syntax, and switches.

Refer to *EE-177 SHARC SPI Booting*, *EE-199 Link Port Booting on the ADSP-21161 SHARC DSP*, *EE-209 Asynchronous Host Interface on the ADSP-21161 SHARC DSP* on the Analog Devices Processor Web site for related information.

ADSP-21161 Processor Booting

ADSP-21161 processors support five boot types (modes): EPROM, host, link port, SPI port, and no-boot, described in Table 5-8 and Table 5-3 on page 5-7. Boot-loadable files for these modes pack boot data into words of appropriate widths and use an appropriate DMA channel of the processor's DMA controller to boot-load the words.

- When booting from an EPROM through the external port, the ADSP-21161 processor reads boot data from an 8-bit external EPROM.
- When booting from a host processor through the external port, the ADSP-21161 processor accepts boot data from 8- or 16-bit host microprocessor.
- When booting through the link port, the ADSP-21161 processor receives boot data through the link port as 4-bit wide data in link buffer 4.
- When booting through the SPI port, the ADSP-21161 processor uses DMA channel 8 of the IO processor to transfer instructions to internal memory. In this boot mode, the processor receives data in the `SPIRX` register.
- In no-boot mode, ADSP-21161 processors begin executing instructions from external memory.

Software developers who use the loader should be familiar with the following operations.

- "Power-Up Booting Process" on page 5-3
- "Boot Mode Selection" on page 5-3
- "Boot Types" on page 5-4
- "Boot Kernels" on page 5-16

- "Boot Kernel Modification and Loader Issues" on page 5-18
- "Interrupt Vector Table" on page 5-21
- "Multiprocessor EPROM Booting" on page 5-21

Power-Up Booting Process

ADSP-21161 processors include a hardware feature that boot-loads a small, 256-instruction program into the processor's internal memory after power-up or after the chip reset. These instructions come from a program called the boot kernel or the loader kernel. When executed, the boot kernel facilitates booting of user application code. The combination of the boot kernel and application code comprise the boot-loadable (`.LDR`) file.

At power-up, after the chip reset, the booting process includes the following steps.

1. Based on the boot type, an appropriate DMA channel is automatically configured for a 256-instruction transfer. This transfer boot-loads the boot kernel program into the processor memory.
2. The boot kernel runs and loads the application executable code and data.
3. The boot kernel overwrites itself with the first 256 words of the application at the end of the booting process. After that, the application executable code starts running.

The boot type selection directs the system to prepare the appropriate boot kernel.

Boot Mode Selection

The state of the `LBOOT`, `EBOOT`, and $\overline{\texttt{BMS}}$ pins selects the ADSP-21161 processor's boot mode (boot type). Table 5-1 and Table 5-2 show how the pin states correspond to the boot type.

Table 5-1. ADSP-21161 Boot Mode Pins

Pin	Type	Description
`EBOOT`	I	EPROM Boot – When `EBOOT` is high, the processor boot-loads from an 8-bit EPROM through the processor's external port. When `EBOOT` is low, the `LBOOT` and $\overline{\text{BMS}}$ pins determine booting mode.
`LBOOT`	I	Link Boot – When `LBOOT` is high (and `EBOOT` is low), the processor boots from another SHARC processor through the processor's link port. When `LBOOT` is low (and `EBOOT` is low), the processor boots from a host processor through the processor's external port.
`BMS`	I/O/T[1]	Boot Memory Select – When boot-loading from EPROM (`EBOOT`=1 and `LBOOT`=0), this pin is an *output* and serves as the chip select for the EPROM. In a multiprocessor system, $\overline{\text{BMS}}$ is output by the bus master. When host-booting, link-booting, or SPI-booting, (`EBOOT`=0), $\overline{\text{BMS}}$ is an input and must be high.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 5-2. ADSP-21161 Boot Mode Pin States

EBOOT	LBOOT	`BMS`	Booting Mode
1	0	Output	EPROM (connect $\overline{\text{BMS}}$ to EPROM chip select)
0	0	1 (Input)	Host processor
0	1	1 (Input)	Link port
0	1	0 (Input)	Serial port (SPI)
0	0	0 (Input)	No-boot (processor executes from external memory)

Boot Types

ADSP-21161 processors support these booting types: EPROM, host, link, and SPI. The following section describe each of the booting types.

- "EPROM Booting" on page 5-5
- "Host Booting" on page 5-9

- "Link Port Booting" on page 5-12
- "SPI Port Booting" on page 5-14
- "No-Boot Mode" on page 5-16

For multiprocessor booting, refer to "Multiprocessor EPROM Booting" on page 5-21.

EPROM Booting

EPROM booting through the external port is selected when the `EBOOT` input is high and the `LBOOT` input is low. These settings cause the $\overline{\texttt{BMS}}$ pin to become an output, serving as chip select for the EPROM.

The `DMAC10` control register is initialized for booting packing boot data into 48-bit instructions. EPROM boot mode uses channel 10 of the IO processor's DMA controller to transfer the instructions to internal memory. For EPROM booting, the processor reads data from an 8-bit external EPROM.

After the boot process loads 256 words into memory locations `0x40000` through `0x400FF`, the processor begins to execute instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (loader kernels) that can load entire programs; see "Boot Kernels" on page 5-16 for more information.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.

Be aware that DMA channel differences between the ADSP-21161 and previous SHARC processors (ADSP-2106x) account for booting differences. Even with these differences, the ADSP-21161 processor supports the same boot capability and configuration as the ADSP-2106x processors. The `DMAC` register default values differ because the ADSP-21161 processor has additional parameters and different DMA channel assignments. EPROM boot mode uses EPB0, DMA channel 10. Similar to ADSP-2106x processors, the ADSP-21161processor boots from DATA23–16.

The processor determines the booting mode at reset from the EBOOT, LBOOT, and $\overline{\text{BMS}}$ pin inputs. When EBOOT=1 and LBOOT=0, the processor boots from an EPROM through the external port and uses $\overline{\text{BMS}}$ as the memory select output. For information on boot mode selection, see the Boot Memory Select pin descriptions in Table 5-8 and Table 5-2 on page 5-4.

When any of the power-up booting modes is used, address 0x40004 should not contain a valid instruction since it is not executed during the booting sequence. Place a NOP or IDLE instruction at this location.

EPROM booting (boot space 8M x 8-bit) through the external port requires that an 8-bit wide boot EPROM be connected to the processor data bus pins 23–16 (DATA23–16). The processor's lowest address pins should be connected to the EPROM address lines. The EPROM's chip select should be connected to $\overline{\text{BMS}}$, and its output enable should be connected to $\overline{\text{RD}}$.

In a multiprocessor system, the $\overline{\text{BMS}}$ output is driven by the ADSP-21161 processor bus master only. This allows the wired OR of multiple $\overline{\text{BMS}}$ signals for a single common boot EPROM.

Systems can boot up to six ADSP-21161 processors from a single EPROM using the same code for each processor or differing code for each processor.

During reset, the ACK line is internally pulled high with the equivalent of an internal 20K ohm resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the ACK line during booting or at any other time.

The RBWS and RBAM fields of the WAIT register are initialized to perform asynchronous access and generate seven wait states (eight cycles total) for the EPROM access in external memory space. Note that wait states defined for boot memory are applied to $\overline{\text{BMS}}$ asserted accesses.

Table 5-3 shows how DMA channel 10 parameter registers are initialized at reset. The count register (CEP0) is initialized to 0x0100 to transfer 256 words to internal memory. The external count register (ECEP0), used when external addresses (BMS space) are generated by the DMA controller, is initialized to 0x0600 (0x0100 words at six bytes per word). The DMAC10 control register is initialized to 0x00 0561.

The default value sets up external port transfers as follows.

- DEN = 1, external port enabled
- MSWF = 0, LSB first
- PMODE = 101, 8-bit to 48-bit packing, Master = 1
- DTYPE = 1, three column data

Table 5-3. DMA Channel 10 Parameter Registers for EPROM Booting

Parameter Register	Initialization Value
IIEP0	0x40000
IMEP0	Uninitialized (increment by 1 is automatic)
CEP0	0x100 (256 instruction words)
CPEP0	Uninitialized
GPEP0	Uninitialized
EIEP0	0x800000
EMEP0	Uninitialized (increment by 1 is automatic)
ECEP0	0x600 (256 words x 6 bytes/word)

The following sequence occurs at system start-up, when the processor $\overline{\texttt{RESET}}$ input goes inactive.

1. The processor goes into an idle state, identical to that caused by the `IDLE` instruction. The program counter (`PC`) is set to address `0x40004`.
2. The DMA parameter registers for channel 10 are initialized as shown in Table 5-3.
3. $\overline{\texttt{BMS}}$ becomes the boot EPROM chip select.
4. 8-bit Master Mode DMA transfers from EPROM to the first internal memory address on the external port data bus lines 23–16.
5. The external address lines (`ADDR23–0`) start at `0x800000` and increment after each access.
6. The $\overline{\texttt{RD}}$ strobe asserts as in a normal memory access with seven wait states (eight cycles).

The processor's DMA controller reads the 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them to internal memory until 256 words have been loaded. The EPROM is automatically selected by the $\overline{\texttt{BMS}}$ pin; other memory select pins are disabled.

The Master DMA internal and external count registers (`ECEP0/CEP0`) decrements after each EPROM transfer. When both counters reach zero, the following wake-up sequence occurs.

1. DMA transfers stop.
2. External port DMA channel 10 interrupt (`EP0I`) is activated.
3. $\overline{\texttt{BMS}}$ is deactivated, and normal external memory selects are activated.
4. The processor vectors to the `EP0I` interrupt vector at `0x40050`.

At this point the processor has completed its booting mode and is executing instructions normally. The first instruction at the `EP0I` interrupt vector location, address `0x40050`, should be an `RTI` (return from interrupt). This process returns execution to the reset routine at location `0x40005` where normal program execution can resume. After reaching this point, a program can write a different service routine at the `EP0I` vector location `0x40050`.

Host Booting

The processor can boot from a host processor through the external port. Host booting is selected when the `EBOOT` and `LBOOT` inputs are low and $\overline{\texttt{BMS}}$ is high. Configured for host booting, the processor enters the slave mode after reset and waits for the host to download the boot program.

The `DMAC10` control register is initialized for booting, packing boot data into 48-bit instructions. Channel 10 of the IO processor's DMA controller is used to transfer instructions to internal memory. Processors accept data from 8- or 16-bit host microprocessor (or other external devices).

After the boot process loads 256 words into memory locations `0x40000` through `0x400FF`, the processor begins executing instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (loader kernels) that can load entire programs; refer to "Boot Kernels" on page 5-16 for more information.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.

ⓘ DMA channel differences between ADSP-21161 and previous SHARC family processors (ADSP-2106x) account for booting differences. Even with these differences, ADSP-21161 processors support the same boot capability and configuration as ADSP-2106x processors. The `DMAC10` register default values differ because the ADSP-21161processor has additional parameters and different DMA channel assignments. Host boot mode uses `EPB0`, DMA channel 10.

The processor determines the boot mode at reset from the `EBOOT`, `LBOOT`, and $\overline{\texttt{BMS}}$ pin inputs. When `EBOOT=0`, `LBOOT=0`, and `BMS=1`, the processor boots from a host through the external port. Refer to Table 5-1 and Table 5-2 on page 5-4 for boot mode selection.

When any of the power-up booting modes is used, address `0x40004` should not contain a valid instruction since it is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location.

During reset, the processor `ACK` line is internally pulled high with an equivalent 20K ohm resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the `ACK` line during booting or at any other time.

Table 5-4 shows how the DMA channel 10 parameter registers are initialized at reset for host booting. The internal count register (`CEP0`) is initialized to `0x0100` to transfer 256 words to internal memory. The `DMAC10` control register is initialized to `0000 0161`.

The default value sets up external port transfers as follows.

- `DEN = 1`, external port enabled
- `MSWF = 0`, LSB first
- `PMODE = 101`, 8-bit to 48-bit packing
- `DTYPE = 1`, three column data

Table 5-4. DMA Channel 10 Parameter Register for Host Booting

Parameter Register	Initialization Value
IIEP0	0x0004 0000
IMEP0	Uninitialized (increment by 1 is automatic)
CEP0	0x0100 (256 instruction words)
CPEP0	Uninitialized
GPEP0	Uninitialized
EIEP0	Uninitialized
EMEP0	Uninitialized
ECEP0	Uninitialized

At system start-up, when the processor $\overline{\text{RESET}}$ input goes inactive, the following sequence occurs.

1. The processor goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to address 0x40004.
2. The DMA parameter registers for channel 10 are initialized as shown in Table 5-4.
3. The host uses HBR and CS to arbitrate for the bus.
4. The host can write to SYSCON (if HBG and READY are returned) to change boot width from default.
5. The host writes boot information to external port buffer 0.

The slave DMA internal count register (CEP0) decrements after each transfer. When CEP0 reaches zero, the following wake-up sequence occurs.

1. The DMA transfers stop.
2. The external port DMA channel 10 interrupt (EP0I) is activated.
3. The processor vectors to the EP0I interrupt vector at 0x40050.

At this point the processor has completed its booting mode and is executing instructions normally. The first instruction at the EP0I interrupt vector location, address 0x40050, should be an RTI (return from interrupt). This process returns execution to the reset routine at location 0x40005 where normal program execution can resume. After reaching this point, a program can write a different service routine at the EP0I vector location 0x40050.

Link Port Booting

Link port booting uses DMA channel 8 of the IO processor to transfer instructions to internal memory. In this boot mode, the processor receives 4-bit wide data in link buffer 0.

After the boot process loads 256 words into memory locations 0x40000 through 0x400FF, the processor begins to execute instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (loader kernels) that load an entire program through the selected port; refer to "Boot Kernels" on page 5-16 for more information.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.

ⓘ DMA channel differences between ADSP-21161 and previous SHARC family processors (ADSP-2106x) account for booting differences. Even with these differences, ADSP-21161 processors support the same boot capability and configuration as ADSP-2106x processors.

The processor determines the booting mode at reset from the `EBOOT`, `LBOOT` and $\overline{\texttt{BMS}}$ pin inputs. When `EBOOT=0`, `LBOOT=1`, and `BMS=1`, the processor boots through the link port. For information on boot mode selection, see Table 5-1 and Table 5-2 on page 5-4.

ⓘ When any of the power-up booting modes is used, address `0x40004` should not contain a valid instruction since it is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location.

In link port booting, the processor gets boot data from another processor link port or 4-bit wide external device after system power-up.

The external device must provide a clock signal to the link port assigned to link buffer 0. The clock can be any frequency up to the processor clock frequency. The clock falling edges strobe the data into the link port. The most significant 4-bit nibble of the 48-bit instruction must be downloaded first.

Table 5-5 shows how the DMA channel 8 parameter registers are initialized at reset. The count register (`CLB0`) is initialized to `0x0100` to transfer 256 words to internal memory. The `LCTL` register is overridden during link port booting to allow link buffer 0 to receive 48-bit data.

In systems where multiple processors are not connected by the parallel external bus, booting can be accomplished from a single source through the link ports. To simultaneously boot all the processors, make a parallel common connection to link buffer 0 on each of the processors. If a daisy

Table 5-5. DMA Channel 8 Parameter Register for Link Port Booting

Parameter Register	Initialization Value
`IILB0`	`0x0004 0000`
`IMLB0`	Uninitialized (increment by 1 is automatic)
`CLB0`	`0x0100` (256 instruction words)
`CPLB0`	Uninitialized
`GPLB0`	Uninitialized

chain connection exists between the processors' link ports, each processor can boot the next processor in turn. Link buffer 0 must always be used for booting.

SPI Port Booting

Serial Peripheral Interface (SPI) port booting uses DMA channel 8 of the IO processor to transfer instructions to internal memory. In this boot mode, the processor receives 8-bit wide data in the `SPIRX` register.

During the booting process, the program loads 256 words into memory locations `0x40000` through `0x400FF`. The processor subsequently begins executing instructions. Because most processor programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (loader kernels) that load an entire program through the selected port. See "Boot Kernels" on page 5-16 for more information.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations. For information about SPI slave booting, refer to *EE-177: SHARC SPI Booting*, located on the Analog Devices processor Web site.

The processor determines the booting mode at reset from the `EBOOT`, `LBOOT`, and $\overline{\texttt{BMS}}$ pin inputs. When `EBOOT=0`, `LBOOT=1`, and `BMS=0`, the processor boots through its SPI port. For information on the boot mode selection, see Table 5-1 and Table 5-2 on page 5-4.

When any of the power-up booting modes is used, address `0x40004` should not contain a valid instruction because it is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction placed at this location.

For SPI port booting, the processor gets boot data after system power-up from another processor's SPI port or another SPI compatible device.

Table 5-6 shows how the DMA channel 8 parameter registers are initialized at reset. The SPI Control Register (`SPICTL`) is configured to `0x0A001F81` upon reset during SPI boot.

This configuration sets up the `SPIRX` register for 32-bit serial transfers. The `SPIRX` DMA channel 8 parameter registers are configured to DMA in `0x180` 32-bit words into internal memory normal word address space starting at `0x40000`. Once the 32-bit DMA transfer completes, the data is accessed as 3 column, 48-bit instructions. The processor executes a 256 word (`0x100`) loader kernel upon completion of the 32-bit, `0x180` word DMA.

For 16-bit SPI hosts, two words are shifted into the 32-bit receive shift register before a DMA transfer to internal memory occurs. For 8-bit SPI hosts, four words are shifted into the 32-bit receive shift register before a DMA transfer to internal memory occurs.

Table 5-6. DMA Channel 8 Parameter Register for SPI Port Booting

Parameter Register	Initialization Value
`IISRX`	`0x0004 0000`
`IMSRX`	Uninitialized (increment by 1 is automatic)

Table 5-6. DMA Channel 8 Parameter Register for SPI Port Booting

Parameter Register	Initialization Value
`CSRX`	`0x0180` (256 instruction words)
`GPSRX`	Uninitialized

No-Boot Mode

No-boot mode causes the processor to start fetching and executing instructions at address `0x200004` in external memory space. In no-boot mode, the processor does not boot-load and all DMA control and parameter registers are set to their default initialization values. The loader is not used to produce the code for no-boot execution.

Boot Kernels

The boot-loading process starts with a transfer of the boot kernel program into the processor memory. The boot kernel sets up the processor and loads boot data. After the boot kernel finishes initializing the rest of the system, the boot kernel loads boot data over itself with a final DMA transfer.

Four boot kernels ship with VisualDSP++; refer to Table 5-7.

Table 5-7. ADSP-21161 Default Boot Kernel Files

PROM Booting	Link Booting	Host Booting	SPI Booting
`161_prom.asm`	`161_link.asm`	`161_host.asm`	`161_spi.asm`

Boot kernels are loaded at processor reset into the `seg_ldr` memory segment, which is defined in `161_ldr.ldf`. The file is stored in the processor tools installation directory, ...`\211xx\ldr`.

Blocks and Block Headers

The loader produces the boot stream in blocks and inserts header words at the beginning of data blocks in the loader (`.LDR`) file. The boot kernel uses header words to properly place data and instruction blocks into processor memory. The header format for PROM, host, and link boot-loader files is as follows.

```
0x00000000DDDD
0xAAAAAAAALLLL
```

In the above example, `D` is a data block type tag, `A` is a block start address, and `L` is a block word length.

For single-processor systems, the data block header has three 32-bit words in SPI boot type, as follows.

`0x0000LLLL`	First word. Data word length or data word count of the data block.
`0xAAAAAAAA`	Second word. Data block start address.
`0x000000DD`	Third word. Tag of data block type.

The loader kernel examines the tag to determine the type of data or instruction being loaded. Table 5-8 lists ADSP-21161N processor data tags.

Table 5-8. ADSP-21161 Data Tags

Tag Number	Block Type	Tag Number	Block Type
`0x0000`	final init	`0x000E`	init pm48
`0x0001`	zero dm16	`0x000F`	zero dm64
`0x0002`	zero dm32	`0x0010`	init dm64
`0x0003`	zero dm40	`0x0012`	init pm64
`0x0004`	init dm16	`0x0013`	init pm8 ext

Table 5-8. ADSP-21161 Data Tags (Cont'd)

Tag Number	Block Type	Tag Number	Block Type
0x0005	init dm32	0x0014	init pm16 ext
0x0007	zero pm16	0x0015	init pm32 ext
0x0008	zero pm32	0x0016	init pm48 ext
0x0009	zero pm40	0x0017	zero pm8 ext
0x000A	zero pm48	0x0018	zero pm16 ext
0x000B	init pm16	0x0019	zero pm32 ext
0x000C	init pm32	0x001A	zero pm48 ext
0x0011	zero pm64		

Boot Kernel Modification and Loader Issues

Some systems require boot kernel customization. In addition, the operation of other tools (such as the C/C++ compiler) is influenced by whether the loader is used.

If you do not specify a boot kernel file via the **Load** page of the **Project Options** dialog box in VisualDSP++ (or via the `-l` command-line switch), the loader places a default boot kernel in the loader output file (see "Boot Kernels" on page 5-16) based on the specified boot type.

Rebuilding a Boot Kernel File

If you modify the boot kernel source (`.ASM`) file by inserting correct values for your system, you must rebuild the boot kernel (`.DXE`) before generating the boot-loadable (`.LDR`) file. The boot kernel source file contains default values for the `SYSCON` register. The `WAIT`, `SDCTL`, and `SDRDIV` initialization code are in boot kernel file comments.

To modify a boot kernel source file:

1. Copy the applicable boot kernel source file (`161_link.asm`, `161_host.asm`, `161_prom.asm`, or `161_spi.asm`).
2. Apply the appropriate initializations of the `SYSCON` and `WAIT` registers.

After modifying the boot kernel source file, rebuild the boot kernel (`.DXE`) file. Do this from the VisualDSP++ IDDE (refer to VisualDSP++ online Help for details), or rebuild the boot kernel file from the command line.

Rebuilding a Boot Kernel Using Command Lines

Rebuild a boot kernel by using command lines as follows.

EPROM Booting. The default boot kernel source file for EPROM booting is `161_prom.asm`. After copying the default file to `my_prom.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21161 my_prom.asm
linker -T 161_ldr.ldf my_prom.doj
```

Host Booting. The default boot kernel source file for host booting is `161_host.asm`. After copying the default file to `my_host.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21161 my_host.asm
linker -T 161_ldr.ldf my_host.doj
```

Link Booting. The default boot kernel source file for link booting is `161_link.asm`. After copying the default file to `my_link.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21161 my_link.asm
linker -T 161_ldr.ldf my_link.doj
```

SPI Booting. The default boot kernel source file for link booting is `161_SPI.asm`. After copying the default file to `my_SPI.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21161 my_SPI.asm
linker -T 161_ldr.ldf my_SPI.doj
```

Loader File Issues

If you modify the boot kernel for the EPROM, host, SPI, or link booting modes, ensure that the `seg_ldr` memory segment is defined in the `.LDF` file. Refer to the source of this memory segment in the `.LDF` file located in the ...`\ldr\` installation directory of the target processor.

Because the loader uses this address for the first location of the reset vector during the boot-load process, avoid placing code at this address. When using any of the processor's power-up booting modes, ensure that this address does not contain a critical instruction, because this address is not executed during the booting sequence. Place a `NOP` or `IDLE` in this location. The loader generates a warning if the vector address `0x40004` does not contain `NOP` or `IDLE`.

When using VisualDSP++ to create the loader file, specify the name of the customized boot kernel executable in the **Kernel file** box on the **Load** page of the **Project Options** dialog box.

Interrupt Vector Table

If the ADSP-21161 processor is booted from an external source (EPROM, host, link port, or SPI), the interrupt vector table is located in internal memory. If the processor is not booted and executes from external memory (no-boot mode), the vector table must be located in external memory.

The `IIVT` bit in the `SYSCON` control register can be used to override the booting mode in determining where the interrupt vector table is located. If the processor is not booted (no-boot mode), setting `IIVT` to `1` selects an internal vector table, and setting `IIVT` to zero selects an external vector table. If the processor is booted from an external source (any boot mode other than no-boot), `IIVT` has no effect. The default initialization value of `IIVT` is zero.

Multiprocessor EPROM Booting

Currently, the loader generates single-processor loader files for host, link, and SPI port booting, and supports multiprocessor EPROM booting only. The application code must be modified to properly set up multiprocessor booting in host, link, and SPI port booting modes.

There are two methods by which a multiprocessor system can be booted:

- "Booting From a Single EPROM"
- "Sequential EPROM Booting"

Regardless of the method, processors perform the following steps.

1. Arbitrate for the bus.
2. Upon becoming bus master, DMA the 256 word boot stream.
3. Release the bus.
4. Execute the loaded instructions.

Booting From a Single EPROM

The $\overline{\texttt{BMS}}$ signals from each processor may be wire ORed together to drive the EPROM's chip select pin. Each processor can boot in turn, according to its priority. When the last processor has finished booting, it must inform the other processors (which may be in the idle state) that program execution can begin (if all processors are to begin executing instructions simultaneously).

When multiple processors boot from a single EPROM, the processors can boot identical code or different code from the EPROM. If the processors load differing code, use a jump table in the loader file (based on processor ID) to select the code for each processor.

Sequential EPROM Booting

Set the `EBOOT` pin of the processor with ID of 1 high for EPROM booting. The other processors should be configured for host booting (`EBOOT=0`, `LBOOT=0`, and `BMS=1`), leaving them in the idle state at startup and allowing the processor with `ID=1` to become bus master and boot itself. Connect the $\overline{\texttt{BMS}}$ pin of processor #1 only to the EPROM's chip select pin. When processor #1 has finished booting, it can boot the remaining processors by writing to their external port DMA buffer 0 (`EPB0`) via the multiprocessor memory space.

The loader can produce boot-loadable files that permit SHARC processors in a multiprocessor system to boot from a single EPROM. In such a system, the processors $\overline{\texttt{BMS}}$ signals are ORed together to drive the EPROM's chip select pin. Each processor boots in turn, according to its priority. When the last processor has finished booting, it must inform the other processors to begin program execution.

Processor ID Numbers

A single-processor system requires only one input (`.DXE`) file without any prefix and suffix to the input file name, for example:

```
elfloader -proc ADSP-21161 -bprom Input.dxe
```

A multiprocessor system requires a distinct processor ID number for each input file on the command line. A processor ID is provided via the `-id#exe=`*`filename`*`.dxe` switch, where `#` is `1` to `6`.

In the following example, the loader processes the input file `Input1.dxe` for the processor with an ID of `1` and the input file `Input2.dxe` for the processor with an ID of `2`.

```
elfloader -proc ADSP-21261 -bprom -id1exe=Input1.dxe
          -id2exe=Input2.dxe
```

If the executable for the `#` processor is identical to the executable of the *`N`* processor, the output loader file contains only one copy of the code from the input file, as the command-line switch `-id#ref=N` is used in the example

```
elfloader -proc ADSP-21161 -bprom -id1exe=Input.dxe -id2ref=1
```

where `2` is the processor ID and `1` is another processor ID referred to by processor `2`.

The loader points the `id(2)exe` loader jump table entry to the `id(1)exe` image, effectively reducing the size of the loader file.

ADSP-21161 Processor Loader Guide

Loader operations depend on the loader options, which control how the loader processes executable files. You select features such as boot mode, boot kernel, and output file format via the loader options. 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. For information specific to the ADSP-21161 processor, refer to the VisualDSP++ online help for that processor. When you open the **Load** page, the default loader settings for the selected processor are already set. Use the **Additional Options** box to enter options that have no dialog box equivalent.

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

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

- "Using ADSP-21161 Loader Command Line" on page 5-24
- "Using VisualDSP++ Interface (Load Page)" on page 5-27

Using ADSP-21161 Loader Command Line

Use the following syntax for the ADSP-21161 loader command line.

```
elfloader inputfile -proc ADSP-21161 -switch [-switch …]
```

where:

- `inputfile`—Name of the executable file (`.DXE`) to be processed into a single boot-loadable file. An input file name can include the drive and directory. Enclose long file names within straight quotes, “`long file name`”.
- `-proc` ADSP-21161—Part number of the processor for which the loadable file is built. The `-proc` switch is mandatory.
- `-switch` …—One or more optional switches to process. Switches select operations and boot modes for the loader. A list of all switches and their descriptions appear in Table 5-10 on page 5-28.

Command-line switches are not case-sensitive and placed on the command line in any order.

Single-Processor Systems

The following command line,

```
elfloader Input.dxe -bSPI -proc ADSP-21161
```

runs the loader with:

- `Input.dxe`—Identifies the executable file to process into a boot-loadable file for a single-processor system. Note that the absence of the `-o` switch causes the output file name to default to `Input.ldr`.
- `-bSPI`—Specifies SPI port booting as the boot type for the boot-loadable file.
- `-proc ADSP-21161`—Specifies ADSP-21161 as the target processor.

Multiprocessor Systems

The following command line,

```
elfloader -proc ADSP-21161 -bprom -id1exe=Input1.dxe
          -id2exe=Input2.dxe
```

runs the loader with:

- `-proc ADSP-21161`—Specifies ADSP-21161 as the target processor.
- `-bprom`—Specifies EPROM booting as the boot type for the boot-loadable file.
- `id1exe=Input1.dxe`—Identifies `Input1.dxe` as the executable file to process into a boot-loadable file for a processor with ID #1 (see "Processor ID Numbers" on page 5-23).
- `id2exe=Input2.dxe`—Identifies `Input2.dxe`. as the executable file to process into a boot-loadable file for a processor with ID #2 (see "Processor ID Numbers" on page 5-23).

File Searches

File searches are important in loader processing. The loader supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-11.

File Extensions

Some loader switches take a file name as an optional parameter. Table 5-9 lists the expected file types, names, and extensions.

Table 5-9. File Extensions

Extension	File Description
`.DXE`	Executable files and boot kernel files. The loader recognizes overlay memory files (`.OVL`) and shared memory files (`.sm`), but does not expect these files on the command line. Place `.OVL` and `.SM` files in the same directory as the `.DXE` file that refers to them so the loader can find them when processing the `.LDR` file. The `.OVL` and `.SM` files may also be placed in the `.OVL` and `.SM` file output directory specified in the `.LDF` file or the current working directory.
`.LDR`	Loader output file

Loader Command-Line Switches

Table 5-10 is a summary of the ADSP-21161 loader switches.

Using VisualDSP++ Interface (Load Page)

When developing a Loader file project from VisualDSP++, modify the default options from the **Load** page (also called **Load** property page) of the **Projects Options** dialog box. For information relative to a specific processor, refer to the VisualDSP++ online help for that processor.

VisualDSP++ invokes the `elfloader` utility to build the output file. The **Load** page buttons and fields correspond to loader command line switches and parameters (see Table 5-10 on page 5-28). Use the **Additional Options** box to enter options that do not have dialog box equivalents.

Table 5-10. ADSP-21161 Loader Command-Line Switches

Switch	Description

`-bprom`
`-bhost`
`-blink`
`-bspi`
	Specifies the boot mode. The `-b` switch directs the loader to prepare a boot-loadable file for the specified boot mode. The valid modes (boot types) are PROM, host, link, and SPI.
	If `-b` does not appear on the command line, the default is `-bprom`.
	To use a custom boot kernel, the boot type selected with the `-b` switch must correspond with the boot kernel selected with the `-l` switch. Otherwise, the loader automatically selects a default boot kernel based on the selected boot type (see "Boot Kernels" on page 5-16).

`-e`*`filename`*	Except shared memory. The `-e` switch omits the specified shared memory (`.SM`) file from the output loader file. Use this option to omit the shared parts of the executable file from multiprocessor boot files. To omit multiple `.SM` files, repeat the switch and parameter multiple times on the command line. For example, to omit two files, use: `-efileA.SM -efileB.SM`.

`-fhex`
`-fASCII`
`-fbinary`
`-finclude`
`-fS1`
`-fS2`
`-fS3`
	Species the format of the boot-loadable file (Intel hex-32, ASCII, include, binary, S1, S2, and S3 (Motorola S-records). If the `-f` switch does not appear on the command line, the default boot file format is hex for PROM, and ASCII for host, link, or `SPI`.
	Available formats depend on the boot type selection (`-b` switch):
	• For a PROM boot type, select a hex, S1, S2, S3, ASCII, or include format.
	• For host or link booting, select an ASCII, binary, or include format.
	• For SPI booting, select an ASCII or binary format.

`-h`
or
`-help`
	Command line help. Outputs the list of command-line switches to standard output and exits.
	Combining the `-h` switch with `-proc ADSP-21161`; for example, `elfloader -proc ADSP-21161 -h`, yields the loader syntax and switches for ADSP-21161 processors. By default, the `-h` switch alone provides help for the loader driver.

Table 5-10. ADSP-21161 Loader Command-Line Switches (Cont'd)

Switch	Description
`-hostwidth #`	Sets up the `.LDR` file word width. By default, the word width for PROM and host is 8, for link is 16, and for SPI is 32. The valid word widths for the various boot modes are: • PROM—8 for hex or ASCII format, 8 or 16 for include format • host—8 or 16 for ASCII or binary format, 16 for include format • link—16 for ASCII, binary, or include format • SPI—8, 16, or 32 for Intel hex-32 or ASCII format
`-id#exe=`*`filename`*	Specifies the processor ID. The `-id#exe` switch directs the loader to use the processor ID (#) for the corresponding executable file (*`filename`*) when producing a boot-loadable file for EPROM booting of a multiprocessor system. This switch is used only to produce a boot-loadable file that boots multiple processors from a single EPROM. Valid values for # are `1`, `2`, `3`, `4`, `5`, and `6`. Do not use this switch for single-processor systems. For single-processor systems, use *`filename`* as a parameter without a switch. For more information, refer to "Processor ID Numbers" on page 5-23.
`-id#ref=`*`N`*	Points the processor ID (*`N`*) loader jump table entry to the ID (#) image. If the executable file for the (#) processor is identical to the executable of the (*`N`*) processor, the switch can be used to set the PROM start address of the processor with ID of # to be the same as for the processor with ID of *`N`*. This effectively reduces the size of the loader file by providing a single copy of an executable to two or more processors in a multiprocessor system. For more information, refer to "Processor ID Numbers" on page 5-23.
`-l` *`kernelfile`*	Directs the loader to use the specified *`kernelfile`* as the boot-loading routine in the output boot-loadable file. The boot kernel selected with this switch must correspond to the boot type selected with the `-b` switch. If the `-l` switch does not appear on the command line, the loader searches for a default boot kernel file. Based on the boot type (`-b` switch), the loader searches in the processor-specific loader directory for the boot kernel file as described in "Boot Kernels" on page 5-16.

Table 5-10. ADSP-21161 Loader Command-Line Switches (Cont'd)

Switch	Description
`-o` *`filename`*	Directs the loader to use the specified *`filename`* as the name for the loader output file. If not specified, the default name is *`inputfile`*`.ldr`.
`-p`*`address`*	Directs the loader to start the boot-loadable file at the specified address in the EPROM. This EPROM address corresponds to `0x8000000` on the ADSP-21161 processor. If the `-p` switch does not appear on the command line, the loader starts the EPROM file at address `0x0`.
`-proc ADSP-21161`	Specifies the processor. This is a mandatory switch.
`-si-revision #|none`	Provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` switch. The parameter 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`. Revision `0.1` is distinct from and "lower" than revision `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` 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 switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader generates an error.

Table 5-10. ADSP-21161 Loader Command-Line Switches (Cont'd)

Switch	Description
`-t#`	(Host boot type only) Specifies timeout cycles. The `-t` switch (for example, `-t100`) limits the number of cycles that the processor spends initializing external memory with zeros. Valid values range from `3` to `32765` cycles; `32765` is the default value. The timeout value `#` is directly related to the number of cycles the processor locks the bus for boot-loading, instructing the processor to lock the bus for no more than two times the timeout number of cycles. When working with a fast host that cannot tolerate being locked out of the bus, use a relatively small timeout value.
`-v`	Outputs verbose loader messages and status information as the loader processes files.
`-version`	Directs the loader to show its version information. Type `elfloader -version` to display the version of the loader drive. Add the `-proc` switch, for example, `elfloader -proc ADSP-21161 -version` to display version information of both loader drive and SHARC loader.

6 LOADER FOR ADSP-2126X/2136X SHARC PROCESSORS

This chapter explains how the loader program (`elfloader.exe`) is used to convert executable (`.DXE`) files into boot-loadable files for ADSP-2126x and ADSP-2136x SHARC processors.

Refer to "Introduction" on page 1-1 for the loader overview; the introductory material applies to all processor families. Refer to "Loader for ADSP-2106x/21160 SHARC Processors" on page 4-1 for information about ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, and ADSP-21160 processors. Refer to "Loader for ADSP-21161 SHARC Processors" on page 5-1 for information about ADSP-21161 processors.

Loader operations specific to ADSP-2126x/2136x SHARC processors are detailed in the following sections.

- "ADSP-2126x/2136x Processor Booting"
  Provides general information about various booting modes, including information about boot kernels.

- "ADSP-2126x/2136x Processor Loader Guide"
  Provides reference information about the loader graphical user interface, command-line syntax, and switches.

ADSP-2126x/2136x Processor Booting

ADSP-2126x and ADSP-2136x processors can be booted from an external PROM memory device via the parallel port (PROM mode) or via the Serial Peripheral Interface port (SPI slave, SPI flash, or SPI master mode). In no-boot mode, the processor is booted from the internal ROM (only available on some processors).

- In parallel port boot mode, the loader output file (`.LDR`) is stored in an 8-bit wide parallel PROM device and fetched by the processor.

The ADSP-2126x/2136x processors do not support multiprocessing. This means that in PROM boot mode, there should be no ID lookup table between the kernel and the rest of the application.

- In SPI slave boot mode, the loader file is transmitted to the processor by a host processor configured as an SPI master.
- There are three cases for the SPI master boot mode: SPI master (no address), SPI PROM (16-bit address), and SPI flash (24-bit address). The difference between the these modes is the way the slave device sends the first word of the `.LDR` file. In SPI PROM and SPI flash boot modes, the `.LDR` file is stored in a passive memory device and fetched by the processor. In SPI master, the `.LDR` file is transmitted to the processor by a host processor configured as an SPI slave.
- In no-boot mode, the processor fetches and executes instructions directly from the internal memory, bypassing the boot loader (boot kernel) entirely. The loader is not used to produce a file supporting no-boot mode.

Software developers who use the loader should be familiar with the following operations.

- "Power-Up Booting Process" on page 6-3
- "Boot Type Selection" on page 6-4
- "Boot Types" on page 6-4
- "Boot Kernels" on page 6-18
- "Interrupt Vector Table" on page 6-21
- "Loader File Section Header" on page 6-22

Power-Up Booting Process

ADSP-2126x and ADSP-2136x processors include a hardware feature that boot-loads a small, 256-instruction, program into the processor's internal memory after power-up or after the chip reset. These instructions come from a program called the boot kernel or the loader kernel. When executed, the boot kernel facilitates booting of user application code. The combination of the boot kernel and application code comprise the boot-loadable (`.LDR`) file.

At power-up, after the chip reset, the booting process includes the following steps.

1. Based on the boot type, an appropriate DMA channel is automatically configured for a 384-word (32-bit) transfer. This transfer boot-loads the boot kernel program into the processor memory.

2. The boot kernel runs and loads the application executable code and data.

3. The boot kernel overwrites itself with the first 256 words of the application at the end of the booting process. After that, the application executable code starts running.

The boot type selection directs the system to prepare the appropriate boot kernel.

Boot Type Selection

Unlike previous SHARC processors, ADSP-2126x/2136x processors do not have a Boot Memory Select ($\overline{\text{BMS}}$) pin. On the ADSP-2126x/2136x processor, the boot type is determined by sampling the state of the `BOOTCFGx` pins, as described in Table 6-1. A description of each boot type follows in "Boot Types".

Table 6-1. ADSP-2126x/2136x Boot Mode Pins

BOOT_CFG[1–0]	Boot Type	Boot Type Selection
`00`	SPI slave	`-bspislave`

`01`	SPI master (SPI flash, SPI PROM, or a host processor via SPI master mode)	`-bspiflash`
		`-bspiprom`
		`-bspimaster`

`10`	EPROM boot via the parallel port	`-bprom`
`11`	Internal boot. (Not available on all ADSP-2126x processors).	Does not use the loader

Boot Types

The following sections describe the ADSP-2126x/2136x processor boot types.

- "PROM Boot Mode" on page 6-5
- "SPI Port Boot Modes" on page 6-8
- "Internal Boot Mode" on page 6-17

PROM Boot Mode

ADSP-2126x/2136x processors support an 8-bit boot mode through the parallel port. This mode is used to boot from external 8-bit-wide memory devices. The processor is configured for 8-bit boot mode when the `BOOT_CFG1-0` pins = `10`. When configured for parallel booting, the parallel port transfers occur with the default bit settings for the `PPCTL` register (shown in Table 6-2).

Table 6-2. PPCTL Register Settings for PROM Boot Mode

Bit	Setting
PPALEPL	= 0; ALE is active high
PPEN	= 1
PPDUR	= 10111; (23 core clock cycles per data transfer cycle)
PPBHC	= 1; insert a bus hold cycle on every access
PP16	= 0; external data width = 8 bits
PPDEN	= 1; use DMA
PPTRAN	= 0; receive (read) DMA
PPBHD	= 0; buffer hang enabled

The parallel port DMA channel is used when downloading the boot kernel information to the processor. At reset, the DMA parameter registers are initialized to the values listed in Table 6-3.

Table 6-3. Parameter Register Settings for PROM Boot Mode

Parameter Register	Initialization Value	Comment
PPCTL	0x0000 016F	See Table 6-2.
IIPP	0 for ADSP-2126x processors 0x10000 for ADSP-2136x processors	This is the offset from internal memory normal word starting address of 0x80000.

Table 6-3. Parameter Register Settings for PROM Boot Mode (Cont'd)

Parameter Register	Initialization Value	Comment
ICPP	0x180 (384)	This is the number of 32-bit words that are equivalent to 256 instructions (48-bit).
IMPP	0x01	
EIPP	0x00	
ECPP	0x600	This is the number of bytes in 0x100 48-bit instructions.
EMPP	0x01	

Packing Options for External Memory

For the ADSP-2126x/2136x processor, the external memory address ranges are 0x1000000-0x2FFFFFF. The parallel port automatically packs internal 32-bit words to either 8-bit or 16-bit words for external memory. These are the only widths supported. The WIDTH() command in the linker specifies which packing mode should be used to initialize the external memory: WIDTH(8) for 8-bit memory, and WIDTH(16) for 16-bit memory.

The linker packs the external memory data in the .DXE file according to the WIDTH() and PACKING() commands. This is the only valid way to specify external memory. The correct physical address is in the .DXE file (selected with WIDTH() and PACKING()) because each 8-bit or 16-bit word occupies one external address, and there is no logical addressing of external memory. The loader unpacks the data from the .DXE file and packs the data again into 32-bit words in the loader file (see Table 6-4). In the loader file, tag INIT_EXT8 is used for 8-bit external packed sections, and tag INIT_EXT16 is used for 16-bit external packed sections.

ZERO_INIT sections are treated like 32-bit ZERO_INIT sections, meaning the count contains the number of 32-bit zeros.

Table 6-4. External Packed Sections in .DXE Files Vs. Repacked Blocks in .LDR Files

External Width	Address in .DXE	Packed in .DXE	Block Address in .LDR	Repacked to 32-bit in .LDR
`WIDTH(8)`	0x01000000	0x0000001100		
	0x01000001	0x0000002200		
	0x01000002	0x0000003300		
	0x01000003	0x0000004400	0x01000000	0x44332211
	0x01000004	0x0000005500		
	0x01000005	0x0000006600		
	0x01000006	0x0000007700		
	0x01000007	0x0000008800	0x01000004	0x88776655
`WIDTH(16)`	0x02000000	0x0000112200		
	0x02000001	0x0000334400	0x02000000	0x33441122
	0x02000002	0x0000556600		
	0x02000003	0x0000778800	0x02000002	0x77885566

Packing and Padding Details

For `ZERO_INIT` sections in a `.DXE` file, no data packing or padding in the `.LDR` file is required because only the header itself is included in the `.LDR` file. However, for other section types, additional data manipulation is required. It is important to note that in *all* cases, the word count placed into the block header in the loader file is the original number of words. That is, the word count does *not* include the padded word.

SPI Port Boot Modes

The ADSP-2126x/2136x processor supports booting from a host processor via Serial Peripheral Interface slave mode (`BOOT_CFG1-0 = 00`), and booting from an SPI flash, SPI PROM, or a host processor via SPI master mode (`BOOT_CFG1-0 = 01`). SPI slave booting is discussed on page 6-9, and SPI master bootings are discussed on page 6-10.

Both SPI boot modes support booting from 8-, 16-, or 32-bit SPI devices. In all SPI boot types, the data word size in the Shift register is hardwired to 32 bits. Therefore, for 8 or 16-bit devices, data words are packed into the Shift register (`RXSPI`) to generate 32-bit words least significant bit (LSB) first, which are then shifted into internal memory.

For 16-bit SPI devices, two words shift into the 32-bit receive Shift register (`RXSR`) before a DMA transfer to internal memory occurs. For 8-bit SPI devices, four words shift into the 32-bit receive Shift register before a DMA transfer to internal memory occurs.

When booting, the ADSP-2126x/2136x SHARC processor expects to receive words into the `RXSPI` register seamlessly. This means that bits are received continuously without breaks in the `CS` link. For different SPI host sizes, the processor expects to receive instructions and data packed in a least significant word (LSW) format.

See the manual for the target SHARC processor peripherals for information on how data is packed into internal memory during SPI booting for SPI devices with widths of 32, 16, or 8 bits.

SPI Slave Boot Mode

In SPI slave boot mode, the host processor initiates the booting operation by activating the SPICLK signal and asserting the $\overline{\text{SPIDS}}$ signal to the active low state. The 256-word boot kernel is loaded 32 bits at a time, via the SPI Receive Shift register. To receive 256 instructions (48-bit words) properly, the SPI DMA initially loads a DMA count of 384 32-bit words, which is equivalent to 256 48-bit words.

The processor's $\overline{\text{SPIDS}}$ pin should not be tied low. When in SPI slave mode, including booting, the $\overline{\text{SPIDS}}$ signal is required to transition from high to low. SPI slave booting uses the default bit settings shown in Table 6-5.

Table 6-5. SPI Slave Boot Bit Settings

Bit	Setting	Comment
SPIEN	Set (= 1)	SPI enabled
MS	Cleared (= 0)	Slave device
MSBF	Cleared (= 0)	LSB first
WL	10, 32-bit SPI	Receive Shift register word length
DMISO	Set (= 1) MISO	MISO disabled
SENDZ	Cleared (= 0)	Send last word
SPIRCV	Set (= 1)	Receive DMA enabled
CLKPL	Set (= 1)	Active low SPI clock
CPHASE	Set (= 1)	Toggle SPICLK at the beginning of the first bit

The SPI DMA channel is used when downloading the boot kernel information to the processor. At reset, the DMA parameter registers are initialized to the values listed in Table 6-6.

Table 6-6. Parameter Register Settings for SPI Slave Boot

Parameter Register	Initialization Value	Comment
`SPICTL`	`0x0000 4D22`	
`SPIDMAC`	`0x0000 0007`	Enabled, RX, initialized on completion
`IISPI`	`0x0008 0000`	Start of Block 0 normal word memory
`IMSPI`	`0x0000 0001`	32-bit data transfers
`CSPI`	`0x0000 0180`	

SPI Master Boot Mode

In SPI master boot mode, the ADSP-2126x/2136x initiates the booting operation by:

1. Activating the `SPICLK` signal and asserting the `FLG0` signal to the active low state.
2. Writing the read command `0x03` and address `0x00` to the slave device.

SPI master boot mode is used when the processor is booting from an SPI compatible serial PROM, serial flash, or slave host processor. The specifics of booting from these devices are discussed individually:

- "Booting From an SPI Flash" on page 6-14
- "Booting From an SPI PROM (16-Bit Address)" on page 6-16
- "Booting From an SPI Host Processor" on page 6-16

On reset, the interface starts up in SPI master mode performing a three hundred eighty-four 32-bit word DMA transfer.

SPI master booting uses the default bit settings shown in Table 6-7.

Table 6-7. SPI Master Boot Mode Bit Settings

Bit	Setting	Comment
SPIEN	Set (= 1)	SPI Enabled
MS	Set (= 1)	Master device
MSBF	Cleared (= 0)	LSB first
WL	10	32-bit SPI Receive Shift register word length
DMISO	Cleared (= 0)	MISO enabled
SENDZ	Set (= 1)	Send zeros
SPIRCV	Set (= 1)	Receive DMA enabled
CLKPL	Set (= 1)	Active low SPI clock
CPHASE	Set (= 1)	Toggle SPICLK at the beginning of the first bit

The SPI DMA channel is used when downloading the boot kernel information to the processor. At reset, the DMA parameter registers are initialized to the values listed in Table 6-8.

Table 6-8. Parameter Registers Settings for SPI Master Boot

Parameter Register	Initialization Value	Comment
SPICTL	0x0000 5D06	
SPIBAUD	0x0064	CCLK/400 =500 KHz@ 200 MHz
SPIFLG	0xfe01	FLG0 used as slave-select
SPIDMAC	0x0000 0007	Enable receive interrupt on completion
IISPI	0x0008 0000	Start of block 0 normal word memory
IMSPI	0x0000 0001	32-bit data transfers
CSPI	0x0000 0180	0x100 instructions = 0x180 32-bit words

From the perspective of the processor, there is no difference between booting from the three types of SPI slave devices. Since SPI is a full-duplex protocol, the processor is receiving the same amount of bits that it sends as a read command. The read command comprises a full 32-bit word (which is what the processor is initialized to send) comprised of a 24-bit address with an 8-bit opcode. The 32-bit word that is received while this read command is transmitted is thrown away in hardware and can never be recovered by the user. Consequently, special measures must be taken to guarantee that the boot stream is identical in all three cases.

The processor boots in Least Significant Bit First (LSB) format, while most serial memory devices operate in Most Significant Bit First (MSB) format. Therefore, it is necessary to program the device in a fashion that is compatible with the required LSBF format. See "Bit Reverse Option for SPI Port Boot Modes" on page 6-13 for details.

Also, because the processor always transmits 32 bits before it begins reading boot data from the slave device, the loader must insert extra data into the byte stream (in the loader file) if using memory devices that do not use the LSB format. The loader includes an option for creating a boot stream compatible with both endian formats, and devices requiring 16-bit and 24-bit addresses, as well as those requiring no read command at all. See "Initial Word Option for SPI Master Boot Modes" on page 6-14 for details.

Figure 6-1 shows the initial 32-bit word sent out from the processor. As shown in the figure, the processor initiates the SPI master boot process by writing an 8-bit opcode (LSB first) to the slave device to specify a read operation. This read opcode is fixed to `0xC0` (`0x03` in MSB first format). Following that, a 24-bit address (all zeros) is always driven by the processor. On the following `SPICLK` cycle (cycle 32), the processor expects the first bit of the first word of the boot stream. This transfer continues until the boot kernel has finished loading the user program into the processor.

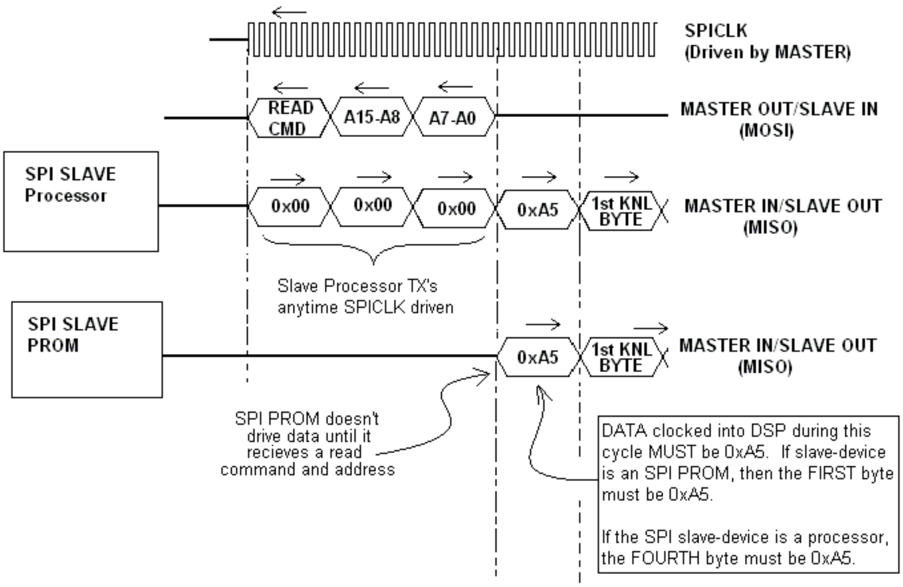


Figure 6-1. SPI Master Mode Booting Using Various Serial Devices

Bit Reverse Option for SPI Port Boot Modes

SPI PROM. For the SPI PROM boot type, the entirety of the SPI master `.LDR` file needs the option of bit-reversing when loading to SPI PROMs. This is because the default setting for the `SPICTL` register (see Table 6-8 on page 6-11) sets the bit order to be LSB first. SPI EPROMs are usually MSB first, so the `.LDR` file must be sent in bit-reversed order.

SPI Master and SPI Slave. When loading to other slave devices, the SPI master and SPI slave boot types do not need bit reversing necessarily. For SPI slave and SPI master boots to non-PROM devices, the same default exists (bit reversed); however, the host (master or slave) can simply be configured to transmit LSB first.

Initial Word Option for SPI Master Boot Modes

Before final formatting (binary, include, etc.) the loader must prepends the word `0xA5` to the beginning of the byte stream. During SPI master booting, the SPI port discards the first byte read from the SPI.

SPI PROM. For the SPI PROM boot type, the word `0xA5` prepended to the stream is one byte in length. SPI PROMs receives a 24-bit read command before any data is sent to the processor, the processor then discards the first byte it receives after this 24-bit opcode is sent (totaling one 32-bit word).

SPI Master. For the SPI master boot type, the word `0xA5000000` prepended to the stream is 32 bits in length. An SPI host configured as a slave begins sending data to the processor while the processor is sending the 24-bit PROM read opcode. These 24-bits must be zero-filled because the processor discards the first 32-bit word that it receives from the slave.

The `0xA5` byte is *only* required for SPI master boot mode.

Figure 6-2 and Table 6-9 illustrates the first 32-bit word for both the SPI PROM and SPI master cases.

With bit reversing for SPI master boot mode, the 32-bit word is handled according to the host width. With bit reversing for SPI PROM boot, the 8-bit word is reversed as a byte and prepended (see Table 6-10).

Booting From an SPI Flash

For SPI flash devices, the format of the boot stream is identical to that used in SPI slave mode, with the first byte of the boot stream being the first byte of the kernel. This is because SPI flash devices do not drive out data until they receive an 8-bit command and a 24-bit address.

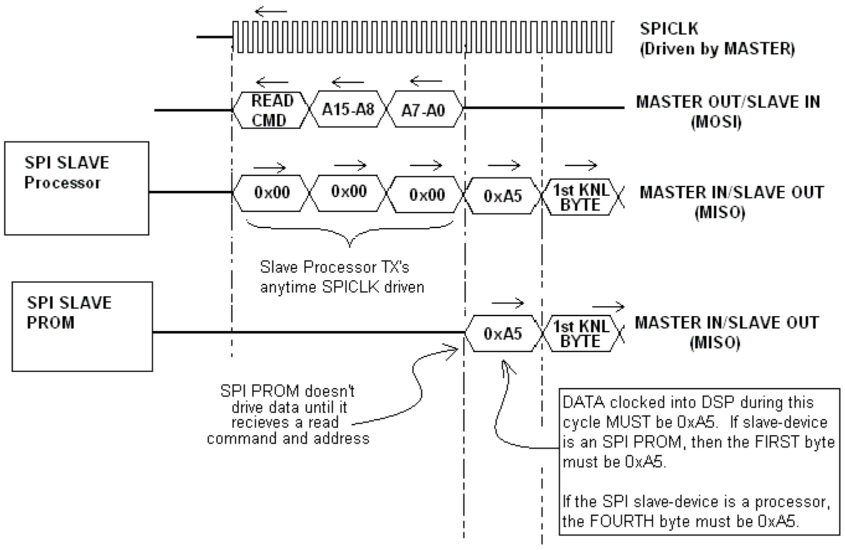


Figure 6-2. SPI Master Boot From a Slave Processor Versus a Slave PROM

Table 6-9. Initial Word for SPI Master and SPI PROM in .LDR File

Boot Type	**Additional Word**	**-hostwidth**		
		32	**16**	**8**
SPI Master[1]	`0xA5000000`	`A5000000`	`0000`	`00`
			`A500`	`00`
				`00`
				`A5`
SPI PROM[2]	`0xA5`	`A5`	`A5`	`A5`

1 Initial word for SPI master boot type is always 32 bits. See Figure 6-1 on page 6-13 for explanation.

2 Initial word for SPI PROM boot type is always 8 bits. See Figure 6-1 on page 6-13 for explanation

Table 6-10. Default Settings for PROM and SPI Boot Modes

Boot Type Selection	Host Width	Output Format	Bit Reverse	Initial Word
`-bPROM`	8	Intel Hex	No	-
`-bSPIslave`	32	ASCII	No	-
`-bSPIflash`	32	ASCII	No	-
`-bspimaster`	32	ASCII	No	`0x000000a5`
`-bspiprom`	8	Intel Hex	Yes	`0xa5`

Booting From an SPI PROM (16-Bit Address)

Figure 6-2 shows the initial 32-bit word sent out from the processor from the perspective of the serial PROM device.

As shown in Figure 6-2, SPI EEPROMs only require an 8-bit opcode and a 16-bit address. These devices begin transmitting on clock cycle 24. However, because the processor is not expecting data until clock cycle 32, it is necessary for the loader to pad an extra byte to the beginning of the boot stream when programming the PROM. In other words, the first byte of the boot kernel is the second byte of the boot stream. The VisualDSP++ tools automatically handles this in the loader file generation process for SPI PROM devices.

Booting From an SPI Host Processor

Typically, host processors in SPI slave mode transmit data on every `SPICLK` cycle. This means that the first four bytes that are sent by the host processor are part of the first 32-bit word that is thrown away by the processor (see Figure 6-1). Therefore, it is necessary for the loader to pad an extra four bytes to the beginning of the boot stream when programming the host; for example, the first byte of the kernel is the fifth byte of the boot stream. VisualDSP++ automatically handles this in the loader file generation process.

Internal Boot Mode

In internal boot mode, upon reset, the processor starts executing the application stored in the internal ROM.

To facilitate internal booting, the `-nokernel` command-line switch commands the loader:

- To omit a boot kernel.
  The `-nokernel` switch denotes that a running on the processor (already booted) subroutine imports the `.LDR` file. The loader does not insert a boot kernel into the `.LDR` file—a similar subroutine is present already on the processor. Instead, the loader file begins with the first header of the first block of the boot stream.

- To omit any interrupt vector table handling.
  In internal boot mode, the boot stream is not imported by a boot kernel executing from within the IVT; no self-modifying `FINAL_INIT` code (which overwrites itself with the IVT) is needed. Thus, the loader does not give any special handling to the 256 instructions located in the IVT (`0x80000-0x800FF` for ADSP-2126x processors and `0x90000-0x900FF` for ADSP-2136x processors). Instead, the IVT code or data are handled like any other range of memory.

- To omit an initial word of `0xa5`.
  When `-nokernel` is selected, the loader does not place an initial word (`A5`) in the boot stream as required for SPI master booting.

- To replace the `FINAL_INIT` block with a `USER_MESG` header.
  The `FINAL_INIT` block (which typically contains the IVT code) should not be included in the `.LDR` file because the contents of the IVT (if any) is incorporated in the boot-stream. Instead, the loader appends one final bock header to terminate the loader file.

The final block header has a block tag of `0x0` (`USER_MESG`). The header indicates to a subroutine processing the boot-stream that this is the end of the stream. The header contains two 32-bit data words, instead of count and address information (unlike the other headers). The words can be used to provide version number, error checking, additional commands, return addresses, or a number of other messages to the importing subroutine on the processor.

The two 32-bit values can be set on the command line as arguments to the "-nokernel[message1, message2]" switch. The first optional argument is *`msg_word1`*, and the second optional argument is *`msg_word2`*, where the values are interpreted as 32-bit unsigned numbers. If only one argument is issued, that argument is *`msg_word1`*. It is not possible to specify *`msg_word2`* without specifying *`msg_word`*1.) If one or no arguments are issued at the command line, the default values for the arguments are `0x00000000`.

Listing 6-1 shows a sample format for the `USER_MESG` header.

Listing 6-1. Internal Booting: FINAL_INIT Block Header Format

```
0x00000000    /* USER_MESG tag */
0x00000000    /* msg_word1 (1st cmd-line parameter) */
0x00000000    /* msg_word2 (2nd cmd-line parameter) */
```

Boot Kernels

The boot-loading process starts with a transfer of the boot kernel program into the processor memory. The boot kernel sets up the processor and loads boot data. After the boot kernel finishes initializing the rest of the system, the boot kernel loads boot data over itself with a final DMA transfer.

Table 6-11 lists ADSP-2126x/2136x boot kernels shipped with VisualDSP++.

Table 6-11. ADSP-2126x/2136x Default Boot Kernel Files

PROM	SPI Slave, SPI Flash, SPI Master, SPI PROM
26x_prom.dxe	26x_spi.dxe
36x_prom.dxe	36x_spi.dxe

At processor reset, a boot kernel is loaded into the seg_ldr memory segment as defined in the Linker Description File for the default loader kernel that corresponds to the target processor, for example, 2126x_ldr.ldf, which is stored in the tools installation directory (...\2126x\ldr).

Boot Kernel Modification and Loader Issues

Boot kernel customization is required for some systems. In addition, the operation of other tools (such as the C/C++ compiler) is influenced by whether the loader is used.

If you do not specify a boot kernel file via the **Load** page of the **Project Options** dialog box in VisualDSP++ (or via the -l command-line switch), the loader places a default boot kernel (see Table 6-11) in the loader output file based on the specified boot type.

If you do not want to use any boot kernel file, check the **No kernel** box (or specify the -nokernel command-line switch). The loader places no boot kernel in the loader output file.

Rebuilding a Boot Kernel File

If you modify the boot kernel source (.ASM) file by inserting correct values for your system, you must rebuild the boot kernel (.DXE) before generating the boot-loadable (.LDR) file. The boot kernel source file contains default values for the SYSCON register. The WAIT, SDCTL, and SDRDIV initialization code are in boot kernel file comments.

To modify a boot kernel source file:

1. Copy the applicable boot kernel source file (`26x_prom.asm`, `26x_spi.asm`, `36x_prom.asm`, or `36x_spi.asm`).
2. Apply the appropriate changes.

After modifying the boot kernel source file, rebuild the boot kernel (`.DXE`) file. Do this from within the VisualDSP++ IDDE (refer to VisualDSP++ online Help for details) or rebuild a boot kernel file from the command line.

Rebuilding a Boot Kernel Using Command Lines

Rebuild a boot kernel using command lines as follows.

PROM Booting. The default boot kernel source file for PROM booting is `26x_prom.asm` for ADSP-2126x processors. After copying the default file to `my_prom.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel.

```
easm21k -proc ADSP-21262 my_prom.asm
linker -T 2162x_ldr.ldf my_prom.doj
```

SPI Booting. The default boot kernel source file for link booting is `2126x_SPI.asm` for ADSP-2126x processors. After copying the default file to `my_SPI.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel:

```
easm21k -proc ADSP-21262 my_SPI.asm
linker -T 2126x_ldr.ldf my_SPI.doj
```

Loader File Issues

If you modify the boot kernel for the PROM or SPI booting modes, ensure that the `seg_ldr` memory segment is defined in the `.LDF` file. Refer to the source of this memory segment in the `.LDF` file located in the ...`\ldr\` installation directory of the target processor.

Because the loader uses this address for the first location of the reset vector during the boot-load process, avoid placing code at the address. When using any of the processor's power-up booting modes, ensure that the address does not contain a critical instruction, because the address is not executed during the booting sequence. Place a `NOP` or `IDLE` in this location. The loader generates a warning if the vector address `0x80004` for ADSP-2126x processors (`0x90004` for ADSP-2136x processors) does not contain `NOP` or `IDLE`.

When using VisualDSP++ to create the loader file, specify the name of the customized boot kernel executable in the **Kernel file** box on the **Load** page of the **Project Options** dialog box.

Interrupt Vector Table

If the ADSP-2126x or ADSP-2136x processor is booted from an external source (PROM or SPI boot modes), the interrupt vector table is located in internal memory (`0x80000-0x800FF` for ADSP-2126x processors, `0x90000-0x900FF` for ADSP-2136x processors). If the processor is not booted and executes from external memory (no-boot mode), the vector table must be located in external memory.

The `IIVT` bit in the `SYSCTL` control register can be used to override the booting mode when determining the location of the interrupt vector table. If the processor is not booted (no-boot mode), setting `IIVT` to 1 selects an internal vector table, and setting `IIVT=0` selects an external vector table. If the processor is booted from an external source (any boot mode other than no-boot), `IIVT` has no effect. The default initialization value of `IIVT` is zero.

Loader File Section Header

The loader generates and inserts a header at the beginning of a block of contiguous data and instructions in the loader file. The kernel uses headers to properly place blocks into processor memory. The architecture of the header follows the convention used by other SHARC processors.

For all of the ADSP-2126x/36x processor boot types, the structures of block header are the same. The header consists of three 32-bit words: the block tag, word count, and destination address. The order of these words is as follows.

`0x000000TT`	First word. Tag of the data block (`T`).
`0x0000CCCC`	Second word. Data word length or data word count (`C`) of the data block.
`0xAAAAAAAA`	Third word. Start address (`A`) of the data block.

ADSP-2126x/2136x Data Tags

Table 6-12 details the ADSP-2126x/2136x processor data tags.

Table 6-12. ADSP-2126x/2136x Data Tag Descriptions

Tag	Count[1]	Address	Padding
`0x0` `FINAL_INIT`			None
`0x1` `ZERO_LDATA`	Number of 16-, 32-, or 64-bit words	Logical short, normal, or long word address	None
`0x2` `ZERO_L48`[2]	Number of 48-bit words	Logical normal word address	None
`0x3` `INIT_L16`	Number of 16-bit words	Logical short word address	If count is odd, pad with 16-bit zero word (See "INIT_L16 Blocks" on page 6-26 for details.)

Table 6-12. ADSP-2126x/2136x Data Tag Descriptions (Cont'd)

Tag	Count[1]	Address	Padding
0x4 INIT_L32	Number of 32-bit words	Logical normal word address	None
0x5 INIT_L48	Number of 48-bit words	Logical normal word address	If count is odd, pad with 48-bit zero word. See "INIT_L48 Blocks" on page 6-24 for details.
0x6 INIT_L64	Number of 64-bit words	Logical long word address	None. See "INIT_L64 Blocks" on page 6-26 for details.
0x7 ZERO_EXT8	Number of 32-bit words	Physical external address	None
0x8 ZERO_EXT16	Number of 32-bit words	Physical external address	None
0x9 INIT_EXT8	Number of 32-bit words	Physical external address	None
0xA INIT_EXT16	Number of 32-bit words	Physical external address	None
0x0 USR_MESG	msg_word1	msg_word2	None. See "Internal Boot Mode" on page 6-17 for more info on *msgword*.

1 The count is the actual number of words and does NOT include padded words added by the loader.

2 40-bit data and 48-bit words are treated identically.

The ADSP-2126x/2136x processor uses eleven block tags, a lesser number of tags compared to other SHARC predecessors. There is only one initialization tag per width because there is no need to draw distinction between pm and dm sections during initialization. The same tag is used for 16-bit (short word), 32-bit (normal word), and 64-bit (long word) blocks that contain only zeros. The 0x1 tag is used for ZERO_INIT blocks of 16-bit, 32-bit, and 64-bit words. The 0x2 tag is used for ZERO_INIT blocks of 40-bit data and 48-bit instructions.

For clarity, the letter L has been added to the names of the internal block tags. L indicates that the associated section header uses the *logical* word count and *logical* address. Previous SHARC boot kernels do not use logical values. For example, the count for a 16-bit block may be the number of 32-bit words rather than the actual number of 16-bit words.

Only four tags are required to handle an external memory, two for each packing mode (see "Packing Options for External Memory" on page 6-6) because Parallel Port DMA is the only way to access the external memory. The external memory can be accessed only via the *physical* address of the memory. This means that each 32-bit word corresponds to either four (for 8-bit) or two (for 16-bit) external addresses. The EXT appended to the name of the block tag indicates that the address is a physical external address.

Two data tags, USER_MESG and FINAL_INIT, differ from the standard format for other SHARC data tags. The USER_MESG header is described on page 6-17 and the FINAL_INIT header on page 6-27.

INIT_L48 Blocks

The INIT_L48 block has one packing and one padding requirements. First, there must be an even number of 48-bit words in the block. If there is an odd number of instructions, then the loader must append one additional 48-bit instruction that is all zeros. In all cases, the count placed into the header is the original logical number of words. That is, the count does not include the padded word. Once the number of words in the block is even, the data in this block is packed according to Table 6-13. The table also shows the case where one zero-word must be added.

Table 6-13. INIT_L48 Block Packing and Zero-Padding (ASCII Format)

Original Data	Packed Into an Even Number of 32-Bit Words	-hostwidth		
		32	16	8
111122223333	22223333	22223333	3333	33
444455556666	66661111	55551111	2222	33
AAAABBBBCCCC	44445555	44445555	1111	22
	BBBBCCCC	BBBBCCCC	6666	22
	0000AAAA	0000AAAA	5555	11
	00000000	00000000	4444	11
			CCCC	66
			BBBB	66
			AAAA	55
			0000	55
			0000	44
			0000	44
				CC
				CC
				BB
				BB
				AA
				AA
				00
				00
				00
				00

Table 6-13. INIT_L48 Block Packing and Zero-Padding (ASCII Format)

Original Data	Packed Into an Even Number of 32-Bit Words	-hostwidth		
		32	16	8
				00
				00

INIT_L16 Blocks

For 16-bit initialization blocks, the number of 16-bit words in the block must be even. If an odd number of 16-bit words is in the block, then the loader adds one additional word (all zeros) to the end of the block, as shown in Table 6-14. The count stored in the header is the actual number of 16-bit words. (The count does not include the padded word.)

Table 6-14. INIT_L16 Block Packing and Zero-Padding (ASCII Format)

Original Data	Packed into an Even Number of 32-bit Words	-hostwidth		
		32	16	8
1122	33441122	33441122	1122	22
3344	00005566	00005566	3344	11
5566			5566	44
			0000	33
				66
				55
				00
				00

INIT_L64 Blocks

For 64-bit initialization blocks, the data is packed as shown in Table 6-15.

Table 6-15. INIT_L64 Block Packing (ASCII Format)

Original Data	Packed into an Even Number of 32-bit Words	-hostwidth		
		32	16	8
1111222233334444	33334444	33334444	4444	44
	11112222	11112222	3333	44
			2222	33
			1111	33
				22
				22
				11
				11

FINAL_INIT Blocks

The final 256-instructions of the `.LDR` file contain the instructions for the interrupt vector table (IVT). The instructions are initialized by a special self-modifying subroutine in the boot kernel (see Listing 6-3). To support the self-modifying code, the loader modifies the `FINAL_INIT` block as follows.

1. Places a multi-function instruction at the fifth instruction of the block.
   The loader places the instruction `R0=R0-R0, DM(I4,M5)=R9, PM(I12,M13)=R11;` at `0x80004` for ADSP-2126x processors or `0x90004` for ADSP-2136x processors. The instruction overwrites whatever instruction is at that address. The opcode for this instruction is `0x39732D802000`.

2. Places an RTI instruction in the IVT.
   The loader places an RTI instruction (opcode `0x0B3E00000000`) at the first address in the IVT entry associated with the boot-source, either PROM or SPI. Unlike the multifunction instruction placed

at 0x80004 (for ADSP-2126x processors) or 0x90004 (for ADSP-2136x processors), which overwrites the data, the loader preserves the user-specified instruction which the RTI replaces. This instruction is stored in the header for FINAL_INIT as shown in Listing 6-2.

- For PROM boot mode, the RTI is placed at address 0x80050 for ADSP-2126x and at 0x90050 for ADSP-2136x processors.
- For all SPI boot modes, the RTI is placed at address 0x80030 for ADSP-2126x and at 0x90030 for ADSP-2136x processors (high priority SPI interrupt).

3. Saves an IVT instruction in the FINAL_INIT block header.
   The count and address of a FINAL_INIT block are constant; to avoid any redundancy, the count and address are not placed into the block header. Instead, the 32-bit count and address words are used to hold the instruction that overwrites the RTI inserted into the IVT. Listing 6-1 illustrates the block header for FINAL_INIT if, for example, the opcode 0xAABBCCDDEEFF is assumed to be the user-intended instruction for the IVT.

Listing 6-2. FINAL_INIT Block Header Format

```
0x00000000          /* FINAL_INIT tag = 0x0   */
0xEEFF0000          /* LSBs of instructions   */
0xAABBCCDD          /* 4 MSBs of instructions */
```

Listing 6-3. FINAL_INIT Section

```
/* ====================== FINAL_INIT ======================== */
/* The FINAL_INIT subroutine in the boot kernel program sets up a
DMA to overwrite itself. The code is the very last piece that
runs in the kernel; it is self-modifying code, It uses a DMA
```

```
to overwrite itself, initializing the 256 instructions that
reside in the Interrupt Vector Table.                       */
/* ------------------------------------------------------------ */

final_init:

  /* ---------- Setup for IVT instruction patch ------------ */
  I8=0x80030;      /* Point to SPI vector to patch from PX    */
  R9=0xb16b0000;   /* Load opcode for “PM(0,I8)=PX” into R9  */
  PX=pm(0x80002);  /* User instruction destined for 0x80030
                      is passed in the section-header for
                      FINAL_INIT. That instr. is initialized upon
                      completion of this DMA (see comments below)
                      using the PX register. */
  R11=BSET R11 BY 9;  /* Set IMDW to 1 for inst. write    */
  DM(SYSCTL)=R11;     /* Set IMDW to 1 for inst. write    */

  /* ------ Setup loop for self-modifying instruction ------- */
  I4=0x80004;          /* Point to 0x080004 for self-modifying
                          code inserted by the loader at 0x80004
                          in bootstream                      */
  R9=pass R9, R11=R12; /* Clear AZ, copy power-on value
                          of SYSCTL to R11                   */
  DO 0x80004 UNTIL EQ; /* Set bottom-of-loop address (loopstack)
                          to 0x80004 and top-of-loop (PC Stack)
                          to the address of the next
                          instruction.                       */
  PCSTK=0x80004;       /* Change top-of-loop value from the
                          address of this instruction to
                          0x80004.                           */

  /* ------------ Setup final DMA parameters -------------- */
  R1=0x80000;DM(IISX)=R1; /* Setup DMA to load over ldr      */
  R2=0x180; DM(CSX)=R2;   /* Load internal count             */
  DM(IMSX)=M6;             /* Set to increment internal ptr    */

  /*----------------- Enable SPI interrupt -------------------*/
  bit clr IRPTL SPIHI; /* Clear any pending SPI interr. latch  */
  bit set IMASK SPIHI;   /* Enable SPI receive interrupt       */
  bit set MODE1 IRPTEN;  /* Enable global interrupts           */

  FLUSH CACHE;         /* Remove any kernel instr's from cache */

  /*---------- Begin final DMA to overwrite this code -------- */
  ustat1=dm(SPIDMAC);
  bit set ustat1 SPIDEN;
  dm(SPIDMAC)=ustat1;    /* Begin final DMA transfer          */

  /*------------ Initiate self-modifying sequence ----------- */
  JUMP 0x80004 (DB);    /* Causes 0x80004 to be the return
                           address when this DMA completes and
                           the RTI at 0x80030 is executed.   */
    IDLE;               /* After IDLE, patch then start       */
    IMASK=0;            /* Clear IMASK on way to 0x80004     */

/* ========================================================== */
/* When this final DMA completes, the high-priority SPI interrupt
is latched, which triggers the following chain of events:

1) The IDLE in the delayed branch to completes
2) IMASK is cleared
3) The PC (now 0x80004 due to the "JUMP RESET (db)") is pushed
   on the PC stack and the processor vectors to 0x80030 to
   service the interrupt.
   Meanwhile, the loader (anticipating this sequence) has auto-
   matically inserted an "RTI" instruction at 0x80030. The user
   instruction intended for that address is instead placed
   in the FINAL_INIT section-header and has loaded into PX before
   the DMA was initiated.)

4) The processor executes the RTI at 0x80030 and vectors to the
   address stored on the PC stack (0x80004).
   Again, the loader has inserted an instruction into the boot
   stream and has placed it at 0x40005 (opcode x39732D802000):
     R0=R0-R0,DM(I4,M5)=R9,PM(I12,M13)=R11;

   This instruction does the following.
   A) Restores the power-up value of SYSCTL (held in R11).
   B) Overwrites itself with the instruction "PM(0,I8)=PX;"
      The first instruction of FINAL_INIT places the opcode for
      this new instruction, 0xB16B00000000, into R9.
   C) R0=R0-R0 causes the AZ flag to be set.

   This satisfies the termination-condition of the loop set up
   in FINAL_INIT ("DO RESET UNTIL EQ;"). When a loop condition
   is achieved within the last three instructions of a loop,
   the processor branches to the top-of-loop address (PCSTK)
   one final time.

5) We manually changed this top-of-loop address 0x80004, and so
   to conclude the kernel, the processor executes the instruction
   at 0x80004 *again*.

6) There’s a new instruction at 0x80004: "PM(0,I8)=PX;". This
   initializes the user-intended instruction at 0x80030 (the vec-
   tor for the High-Priority-SPI interrupt).

At this point, the kernel is finished, and execution continues
at 0x80005, with the only trace as if nothing happened!      */
/* ========================================================== */
```

ADSP-2126x/2136x Processor Loader Guide

Loader operations depend on the loader options, which control how the loader processes executable files. You select features such as boot mode, boot kernel, and output file format via the loader options. 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. For information specific to the ADSP-2126x/2136x processor, refer to the VisualDSP++ online help for that processor. When you open the **Load** page, the default loader settings for the selected processor are already set. Use the **Additional Options** box to enter options that have no dialog box equivalent.

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

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

- "Using the ADSP-2126x/2136x Loader Command Line" on page 6-32
- "Using the VisualDSP++ Interface (Load Page)" on page 6-34

Using the ADSP-2126x/2136x Loader Command Line

Use the following syntax for the ADSP-2126x/2136x SHARC loader command line.

```
elfloader inputfile -proc processor -switch [-switch …]
```

where:

- `inputfile`—Name of the executable file (`.DXE`) to be processed into a single boot-loadable file. An input file name can include the drive and directory. Enclose long file names within straight quotes, “`long file name`”.
- `-proc processor`—Part number of the processor (for example, `-proc ADSP-21262`) for which the loadable file is built. The `-proc` switch is mandatory.
- `-switch …`—One or more optional switches to process. Switches select operations and boot modes for the loader. A list of all switches and their descriptions appear in Table 6-17 on page 6-35.

Command-line switches are not case-sensitive and may be placed on the command line in any order.

The following command line,

```
elfloader Input.dxe -bSPIflash -proc ADSP-21262
```

runs the loader with:

- `Input.dxe`—Identifies the executable file to process into a boot-loadable file. Note that the absence of the `-o` switch causes the output file name to default to `Input.ldr`.
- `-bspiflash`—Specifies SPI flash port booting as the boot type for the boot-loadable file.
- `-proc ADSP-21262` —Specifies ADSP-21262 as the target processor.

File Searches

File searches are important in loader processing. The loader supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-11.

File Extensions

Some loader switches take a file name as an optional parameter. Table 6-16 lists the expected file types, names, and extensions.

Table 6-16. File Extensions

Extension	File Description
`.DXE`	Executable files and boot kernel files. The loader recognizes overlay memory files (`.OVL`) and shared memory files (`.SM`), but does not expect these files on the command line. Place `.OVL` and `.SM` files in the same directory as the `.DXE` file that refers to them. The loader finds the files when processing the `.DXE` file. The `.OVL` and `.SM` files may also be placed in the `.OVL` and `.SM` file output directory specified in the `.LDF` file or the current working directory.
`.LDR`	Loader output file.

Loader Command-Line Switches

Table 6-17 is a summary of the ADSP-2126x and ADSP-2136x loader switches.

Using the VisualDSP++ Interface (Load Page)

When developing a **DSP loader file** project in VisualDSP++, modify the default option settings on the **Load** page of the **Projects Options** dialog box. For information specific to the ADSP-2126x/2136x processor, refer to the VisualDSP++ online help for that processor.

Table 6-17. ADSP-2126x/2136x Loader Command-Line Switches

Switch	Description

`-bprom`
`-bspislave|-bspi`
`-bspimaster`
`-bspiprom`
`-bspiflash`
	Specifies the boot mode. The `-b` switch directs the loader to prepare a boot-loadable file for the specified boot mode.
	The valid modes (boot types) are PROM, SPI slave, SPI master, SPI PROM, and SPI flash.
	If `-b` does not appear on the command line, the default is `-bprom`.
	To use a custom boot kernel, the boot type selected with the `-b` switch must correspond with the boot kernel selected with the `-l` switch. Otherwise, the loader automatically selects a default boot kernel based on the selected boot type (see "Boot Kernels" on page 6-18). Do not use the `-e` switch with the `-nokernel` switch.

`-fhex`
`-fASCII`
`-fbinary`
`-finclude`
`-fs1`
`-fs2`
`-fs3`
	Specifies the format of a boot-loadable file (Intel hex-32, ASCII, binary, include). If the `-f` switch does not appear on the command line, the default boot file format is
	Intel hex-32 for PROM and SPI PROM, ASCII for SPI slave, SPI flash, and SPI master.
	Available formats depend on the boot type selection (`-b` switch):
	• For PROM and SPI PROM boot types, select a hex, ASCII, `s1`, `s2`, `s3`, or include format.
	• For other SPI boot types, select an ASCII or binary format.

`-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 target processor, add the `-proc` switch to the command line. For example: type `elfloader -proc ADSP-21262 -h` to obtain help for ADSP-2126x/2136x processors.

`-hostwidth #`	Sets up the word width for the `.LDR` file. By default, the word width for PROM and SPI PROM boot modes is 8; for SPI slave, SPI flash, and SPI master boot modes is 32. The valid word widths are: • 8 for Intel hex-32 and Motorolla S-records formats; • 8, 16, or 32 for ASCII, binary, and include formats

Table 6-17. ADSP-2126x/2136x Loader Command-Line Switches

Switch	Description
`-l` *`userkernel`*	Directs the loader to use the specified *`userkernel`* and to ignore the default boot kernel for the boot-loading routine in the output boot-loadable file. Note: The boot kernel file selected with this switch must correspond to the boot type selected with the `-b` switch). If the `-l` switch does not appear on the command line, the loader searches for a default boot kernel file in the installation directory (see "Boot Kernels" on page 6-18). Note that the loader does not search for any kernel file if `-nokernel` is selected.
`-nokernel[`*`message1, message2`*`]`	Supports internal boot mode. The `-nokernel` switch directs the loader: • Not to include the boot kernel code into the loader (`.LDR`) file. • Not to perform any special handling for the 256 instructions located in the IVT. • To put two 32-bit hex messages in the final block header (optional). • Not to include the initial word in the loader file. For more information, see "Internal Boot Mode" on page 6-17).
`-o` *`filename`*	Directs the loader to use the specified *`filename`* as the name for the loader's output file. If the `-o` *`filename`* is absent, the default name is the root name of the input file with an `.LDR` extension.
`-p`*`address`*	Specifies the PROM start address. This EPROM address corresponds to 0x80000 (ADSP-2126x processors) or to 0x90000 (ADSP-2136x processors). The -p switch starts the boot-loadable file at the specified address in the EPROM. If the -p switch does not appear on the command line, the loader starts the EPROM file at address 0x0.

`-proc` *`processor`*	Specifies the processor. This is a mandatory switch. The *`processor`* argument is one of the following:
	`ADSP-21261` `ADSP-21363` `ADSP-21367`
	`ADSP-21262` `ADSP-21364` `ADSP-21368`
	`ADSP-21266` `ADSP-21365` `ADSP-21369`
	`ADSP-21267` `ADSP-21366`

Table 6-17. ADSP-2126x/2136x Loader Command-Line Switches

Switch	Description
`-si-revision #|none`	Provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` switch. The parameter 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`. Revision `0.1` is distinct from and "lower" than revision `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` 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 switch parameter (a valid revision 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.
`-version`	Directs the loader to show its version information. Type `elfloader -version` to display the version of the loader drive. Add the `-proc` switch, for example, `elfloader -proc ADSP-21262 -version` to display version information of both loader drive and SHARC loader.

VisualDSP++ invokes the `elfloader` utility to build the output file. Dialog box buttons and fields correspond to command-line switches and parameters (see Table 6-17 on page 6-35). Use the **Additional Options** box to enter options that have no dialog box equivalent.

7 SPLITTER FOR SHARC AND TIGERSHARC PROCESSORS

This chapter explains how the splitter program (`elfspl21k.exe`) is used to convert executable (`.DXE`) files into non-bootable files for ADSP-21xxx SHARC and the ADSP-TSxxx TigerSHARC processors. Non-bootable PROM image files execute from external memory of a processor. For TigerSHARC processors, the splitter creates a 32-bit image file while for SHARC processor, the splitter creates a 64-/48-/40-/32-/16-bit image file.

In most instances, developers working with SHARC and TigerSHARC processor use the loader instead of the splitter. One of the exceptions is a SHARC system that must execute instructions from external memory. The non-bootable PROM image files are often used with ADSP-21065L processor systems, which have limited internal memory.

Refer to "Introduction" on page 1-1 for the splitter overview; the introductory material applies to both processor families.

Splitter operations are detailed in the following sections.

- "SHARC and TigerSHARC Splitter Command Line" on page 7-2
  Provides reference information about the splitter command-line syntax and switches.

- "VisualDSP++ Interface (Split Page)" on page 7-8
  Provides reference information about the splitter graphical user interface.

SHARC and TigerSHARC Splitter Command Line

Use the following syntax for the SHARC and TigerSHARC splitter command line.

```
elfspl21k [-switch…] -pm &|-dm &|-64 &| -proc name inputfile
```

or

```
elfspl21k [-switch…] -s segment inputfile
```

where:

- *`inputfile`*—Specifies the name of the executable file (`.DXE`) to be processed into a non-bootable file for a single-processor system.

  The name of the *`inputfile`*`.DXE` file must appear at the end of the command. The name can include the drive, directory, file name, and file extension. Enclose long file names within straight quotes; for example, "`long file name`".

- *`-switch…`*—One or more optional switches to process. Switches select operations and boot modes for the splitter. Switches may be used in any order. A list of the splitter switches and their descriptions appear in Table 7-2 on page 7-5.

- `-pm &| -dm &| -64`—The `&|` symbol between these switches indicates AND/OR. The splitter command line must include one or more of the `-pm`, `-dm`, and `-64` switches (or the `-s` switch).

- `-s` *`segment`*—The `-s` switch can be used without the `-pm`, `-dm`, or `-64` switch. The splitter command line must include one or more of the `-pm`, `-dm`, and, `-64` switches or the `-s` switch.

Most items in the splitter command line are not case sensitive; for example, `-pm` and `-PM` are interchangeable. However, the names of memory segment names must be identical, including case, to the names used in the executable.

TigerSHARC processors do not have `-pm`, `-dm`, or `-64` switches.

Each of the following command lines,

```
elfspl21k -pm -o pm_stuff my_proj.dxe -proc ADSP-21161
elfspl21k -dm -o dm_stuff my_proj.dxe -proc ADSP-21161
elfspl21k -64 -o 64_stuff my_proj.dxe -proc ADSP-21161
elfspl21k -s seg-code -o seg-code my_proj.dxe -proc ADSP-21161
```

runs the splitter for the ADSP-21161 processor. The first command produces a PROM file for program memory. The second command produces a PROM file for data memory. The third command produces a PROM file for DATA64 memory. The fourth command produces a PROM file for section `seg-code`.

The switches on these command lines are as follows.

`-pm`
`-dm`
`-64`
	Selects program memory (`-pm`), data memory (`-dm`), or DATA64 memory (`-64`) as sources in the executable for extraction and placement into the image. DATA64 memory does not apply to ADSP-2106x processors. The `-pm`, `-dm`, or `-64` switches do not apply to ADSP-TSxxx processors.
	Because these are the only switches used to identify the memory source, the specified sources are PM, DM, or DATA64 memory segments. Because no other content switches appear on these command lines, the output file format defaults to a Motorola 32-bit format (s3), and the PROM word width of the output defaults to 8 bits for all PROMs.

`-o pm_stuff`
`-o dm_stuff`
`-o 64=stuff`
`-o seg-code`
	Specify names for the output files. Use different names so the output of a run does not overwrite the output of a previous run. The output names are `pm_stuff.s_#` and `dm_stuff.s_#`.

`my_proj.dxe`	Specifies the name of the input (`.DXE`) file to be processed into non-bootable PROM image files.

File Searches

File searches are important in the splitter process. The splitter supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described on page 1-11.

Output File Extensions

The splitter follows the conventions shown in Table 7-1 for output file extensions.

Table 7-1. Output File Extensions

Extension	File Description
`.S_#`	Motorola S-record format file. The # indicates the position (`0` = least significant, `1` = next-to-least significant, and so on). For info about Motorola S-record file format, refer to "Splitter Output Files in Motorola S-Record Format" on page A-11.
`.H_#`	Intel hex-32 format file. The # indicates the position (`0` = least significant, `1` = next-to-least significant, and so on). For information about Intel hex-32 file for mat, refer to "Splitter Output Files in Intel Hex-32 Format" on page A-13.
`.STK`	Byte stacked format file. These files are intended for host transfer of data, not for PROMs. For more information about byte stacked file format, format files, refer to "Splitter Output Files in Byte Stacked Format" on page A-13.

Command-Line Switches

A list of the splitter command-line switches and their descriptions appears in Table 7-2.

Table 7-2. Splitter Command-Line Switches

Item	Description
`-64`	The `-64` (include DATA64 memory) switch directs the splitter to extract all sections declared as 64-bit memory segments from the input `.DXE` file. This switch influences the operation of the `-ram` and `-norom` switches, adding 64-bit data memory as their target.
`-dm`	The `-dm` (include data memory) switch directs the splitter to extract memory sections declared as data memory ROM from the input `.DXE` file. The `-dm` switch influences the operation of the `-ram` and `-norom` switches, adding data memory as their target.
`-o` *`imagefile`*	The `-o` (output file) switch directs the splitter to use *`imagefile`* as the name of the splitter output file(s). If not specified, the default name for the splitter output file is `inputfile.ext`, where `ext` depends on the output format.
`-norom`	The `-norom` (no ROM in PROM) switch directs the splitter to ignore ROM memory sections in the *`inputfile`* when extracting information for the output image. The `-dm` and `-pm` switches select data memory or program memory. The operation of the `-s` switch is not influenced by the `-norom` switch.
`-pm`	The `-pm` (include program memory) switch directs the splitter to extract memory sections declared program memory ROM from the input`.DXE` file. The `-pm` switch influences the operation of the `-ram` and `-norom` switches, adding program memory as the target.

Table 7-2. Splitter Command-Line Switches (Cont'd)

Item	Description
`-r # [# …]`	The `-r` (PROM widths) switch specifies the number of PROM files and their width in bits. The splitter can create PROM files for 8-, 16-, and 32-bit wide PROMs. The default width is 8 bits. Each # parameter specifies the width of one PROM file. Place # parameters in order from most significant to least significant. The sum of the # parameters must equal the bit width of the destination memory (40 bits for DM, 48 bits for PM, or 64 bits for 64-bit memory). **Example:** `elfspl21k -dm -r 16 16 8 myfile.dxe -proc ADSP-21062` This command extracts data memory ROM from `myfile.dxe` and creates the following output PROM files. • `myfile.s_0`—8 bits wide, contains bits 7–0 • `myfile.s_1`—16 bits wide, contains bits 23–8 • `myfile.s_2`—16 bits wide, contains bits 39–24 The width of the three output files is 40 bits.
`-ram`	The `-ram` (include RAM in PROM) switch directs the splitter to extract RAM sections from the *`inputfile`*. The `-dm`, `-pm`, and `-64` switches select the memory. The `-s` switch is not influenced by the `-ram` switch.

`-f h`
`-f s1`
`-f s2`
`-f s3`
`-f b`
	The `-f` (PROM file format) switch directs the splitter to generate a non-bootable PROM image file in the specified format.
	Available selection include:
	• `h`—Intel hex-32 format
	• `s1`—Motorola EXORciser format
	• `s2`—Motorola EXORMAX format
	• `s3`—Motorola 32-bit format
	• `b`—byte stacked format
	If the `-f` switch does not appear on the command line, the default format for the PROM file is Motorola 32-bit (`s3`).
	For information on file formats, see "Build Files" on page A-5.

`-s` *`sectionname`*	The `-s` (include memory section) switch directs the splitter to extract the contents of the specified memory section (*`sectionname`*). Use the `-s` *`sectionname`* switch as many times as needed. Each instance of the `-s` switch can specify only one *`sectionname`*. Do not use `-s` with (`-pm`, `-dm`, or `-64`).

Table 7-2. Splitter Command-Line Switches (Cont'd)

Item	Description
`-proc processor`	Specifies the processor type to the splitter. This is a mandatory switch. Valid processors are: • `ADSP-21060`, `ADSP-21061`, `ADSP-21062`, `ADSP-21065L` • `ADSP-21160`, `ADSP-21161` • `ADSP-21261`, `ADSP-21262`, `ADSP-21266`, `ADSP-21267`, • `ADSP-21363`, `ADSP-21364`, `ADSP-21365`, `ADSP-21366`, `ADSP-21367`, `ADSP-21368`, `ADSP-21369` • `ADSP-TS101`, `ADSP-TS201`, `ADSP-TS202`, and `ADSP-TS203`
`-u #`	(Byte stacked format files only) The `-u` (user flags) switch, which may be used only in combination with the `-f b` switch, directs the splitter to use the number # in the user-flags field of a byte stacked format file. If the `-u` switch is not used, the default value for the number is `0`. By default, # is decimal. If # is prefixed with `0x`, the splitter interprets the number as hexadecimal. For more information, see "Splitter Output Files in Byte Stacked Format" on page A-13.
`-si-revision #|none`	Provides a silicon revision of the specified processor. The switch parameter represents a silicon revision of the processor specified by the `-proc` switch. The parameter 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`. Revision `0.1` is distinct from and "lower" than revision `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` 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 switch parameter (a valid revision value)—`-si-revision` alone or with an invalid value—the loader generates an error.
`-version`	Directs the splitter to show its version information.

VisualDSP++ Interface (Split Page)

VisualDSP++ invokes the splitter to build non-bootable PROM image files when you select **Splitter file** as the project output.

Splitter operation relies on splitter options, which control the processing of the executable files into output files. Modify the default splitter options from the **Split** page (also called splitter property page) of the **Project Options** dialog box.

The page buttons and fields correspond to splitter command-line switches and parameters (see Table 7-2 on page 7-5). Use the **Additional Options** box to enter options that do not have dialog box equivalents. Refer to VisualDSP++ online Help for details.

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-15

Most of the development tools use industry-standard file formats. These formats are described in "Format References" on page A-17.

Source Files

This section describes the following source (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 on) 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 the VisualDSP++ 4.0 C/C++ Compiler and Library Manual for the target processor.

[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 the Programming Reference manual for your processor.

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++ 4.0 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 <`*`filename`*`>` 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 based 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 there is some 8-bit data in a 16-bit wide buffer, the assembler loads the equivalent 8-bit value into the most significant eight bits 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++ 4.0 Assembler and Preprocessor Manual.*

Linker Description Files

Linker Description Files (`.LDF`) are ASCII text files that contain commands for the linker in the linker scripting language. For information on the scripting language, see the *VisualDSP++ 4.0 Linker and Utilities Manual*.

Linker Command-Line Files

Linker command-line files (`.TXT`) are ASCII text files that contain command-line inputs for the linker. For more information on the linker command line, see the *VisualDSP++ 4.0 Linker and Utilities Manual*.

Build Files

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

- "Assembler Object Files" on page A-6
- "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
- "Loader Output Files in Include Format" on page A-10
- "Loader Output Files in Binary Format" on page A-11
- "Splitter Output Files in Motorola S-Record Format" on page A-11
- "Splitter Output Files in Intel Hex-32 Format" on page A-13
- "Splitter Output Files in Byte Stacked Format" on page A-13
- "Splitter Output Files in ASCII Format" on page A-15

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 "Format References" on page A-17.

Library Files

Library files (`.DLB`), the output of the archiver, 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-17.

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-17.

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 processor external memory.

Memory Map Files

The linker can output memory map files (`.XML`), which are ASCII text files that contain memory and symbol information for the executable files. The `.XML` file contains a summary of memory defined with the `MEMORY{}` command 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-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 shows the organization of an example data record.

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 shows an end-of-file record.

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
`01`	Record type
`FF`	Checksum

HEXUTIL Utility

The `hexutil` utility program converts Intel hex-32 to Motorola S format or produces an unformatted data file. This example shows how to include this file in a program:

```
%hexutil input_file <switches>
```

where `<switches>` are:

```
-s1|s2|s3|StripHex
```

to specify the output format (`s3` is the default, and `StripHex` generates unformatted data), and

```
-o
```

to specify the output file name in the form `<input_root>.s`.

Loader Output Files in Include Format

The loader can output include format files (`.LDR`). These files permit the inclusion of the loader file in a C program.

The word width (8-, 16-, or 32-bit) of the loader file depends on the specified boot type. Specify the width of the loader output with "-hostwidth #".

Similar to Intel hex-32 output, loader output files include format file have three basic parts in the following order.

1. Boot kernel
2. User application code
3. Saved user code in conflict with the kernel code

The kernel code is the first part. User application code is followed by the saved user code.

Files in `include` format are ASCII text files that consist of 48-bit instructions, one per line. Each instruction is presented as three 16-bit hexadecimal numbers. For each 48-bit instruction, the data order is lower, middle, and then upper 16 bits. Example lines from an `include` format file are:

```
0x005c, 0x0620, 0x0620,
0x0045, 0x1103, 0x1103,
0x00c2, 0x06be, 0x06be
```

This example shows how to include this file in a C program:

```
const unsigned loader_file[] =
   {
   #include "foo.ldr"
   };
```

```
const unsigned loader_file_count = sizeof loader_file
                                   / sizeof loader_file;
```

`loader_file_count` reflects the actual number of elements in the array and cannot be used to process the data.

Loader Output Files in Binary Format

The loader can output binary format files (`.LDR`) to support a variety of PROM and microcontroller storage applications.

Binary format files use less space than the other loader file formats. Binary files have the same contents as the corresponding ASCII file, but in binary format.

Splitter Output Files in Motorola S-Record Format

The splitter can output Motorola S-record format files (.S_#), which conform to the Intel standard. The three file formats supported by the PROM splitter differ only in the width of the address field: `S1` (16 bits), `S2` (24 bits), or `S3` (32 bits).

An S-record file begins with a header record and ends with a termination record. Between these two records are data records, one per line.

S00600004844521B	Header record
S10D00043C4034343426142226084C	Data record (S1)
S903000DEF	Termination record (S1)

Table A-5 shows the organization of an example header record.

Table A-6 shows the organization of an S1 data record.

Table A-5. Example – Header Record

Field	Purpose
`S00600004844521B`	Example record
`S0`	Start character
`06`	Byte count of this record
`0000`	Address of first data byte
`484452`	Identifies records that follow
`1B`	Checksum

Table A-6. Example – S1 Data Record

Field	Purpose
`S10D00043C4034343426142226084C`	Example record
`S1`	Record type
`0D`	Byte count of this record
`0004`	Address of the first data byte
`3C`	First data byte
`08`	Last data byte
`4C`	Checksum

The S2 data record has the same format, except that the start character is S2 and the address field is six characters wide. The S3 data record is the same as the S1 data record except that the start character is S3 and the address field is eight characters wide.

Termination records have an address field that is 16-, 24-, or 32 bits wide, whichever matches the format of the preceding records. Table A-7 shows the organization of an S1 termination record.

The S2 termination record has the same format, except that the start character is S8 and the address field is six characters wide.

Table A-7. Example – S1 Termination Record

Field	Purpose
`S903000DEF`	Example record
`S9`	Start character
`03`	Byte count of this record
`000D`	Address
`EF`	Checksum

The S3 termination record is the same as the S1 format, except the start character is S7 and the address field is eight characters wide.

For more information, see "HEXUTIL Utility" on page A-9.

Splitter Output Files in Intel Hex-32 Format

The splitter can output Intel hex-32 format (`.H_#`) files. These ASCII files support a variety of PROM devices. For an example of how the Intel hex-32 format appears for an 8-bit wide PROM, see "Loader Output Files in Intel Hex-32 Format" on page A-7.

The splitter prepares a set of PROM files. Each PROM holds a portion of each instruction or data. This configuration differs from the loader output.

Splitter Output Files in Byte Stacked Format

The splitter can output files in byte stacked (`.STK`) format. These files are not intended for PROMs, but are ideal for microcontroller data transfers.

A file in byte stacked format comprises a series of one line headers, each followed by a block (one or more lines) of data. The last line in the file is a header that signals the end of the file.

Lines consist of ASCII text that represents hexadecimal digits. Two characters represent one byte. For example, `F3` represents a byte whose decimal value is 243.

Table A-8 shows an example of a header record in byte stacked format.

Table A-8. Example – Header Record in Byte Stacked Format

Field	Purpose
`2000800000000080000001E`	Example record
`20`	Width of address and length fields (in bits)
`00`	Reserved
`80`	PROM splitter flags (80 = PM, 00 = DM)
`00`	User defined flags (loaded with `-u` switch)
`00000008`	Start address of data block
`0000001E`	Number of bytes that follow

In the above example, the start address and block length fields are 32 (0x20) bits wide. The file contains program memory data (the MSB is the only flag currently used in the PROM splitter flags field). No user flags are set. The address of the first location in the block is `0x08`. The block contains 30 (1E) bytes (5 program memory code words). The number of bytes that follow (until next header record or termination record) must be nonzero.

A block of data records follows its header record, five bytes per line for data memory, and six byte per line for program memory. For example:

Program Memory Section (Code or Data)

```
3C4034343426
142226083C15
```

Data Memory Section

```
3C40343434
2614222608
```

The bytes are ordered left to right, most significant to least.

The termination record has the same format as the header record, except for the rightmost field (number of records), which is all zeros.

Splitter Output Files in ASCII Format

When the Blackfin 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 consumes all the executable file types produced by the linker (`.DXE`, `.SM`, `.OVL`). To simulate IO, the debugger also consumes the assembler data file format (`.DAT`) and the 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 and starts reading from the LSB. The `SPORT` register then zero-fills bits shorter than the word size or conversely truncates bits beyond the word size on the MSB end.

In the following example (Table A-9), 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-9. 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