Loader and Utilities Manual

(including ADSP-BFxxx and ADSP-21xxx)

Revision 1.3, May 2014

Part Number

82-100114-01

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

Copyright Information

©2014 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc. Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, SHARC, EngineerZone, VisualDSP++, CrossCore Embedded Studio, EZ-KIT Lite, and EZ-Board are registered trademarks of Analog Devices, Inc.

Blackfin+ is a trademark of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

Contents

Chapter 1: Preface 9
- Purpose of This Manual 9
- Intended Audience 9
- Manual Contents 9
- What's New in This Manual 10
- Technical Support 10
- Supported Processors 11
- Product Information 11
  - Analog Devices Web Site 11
  - EngineerZone 12
- Notation Conventions 12

Chapter 2: Introduction 15
- Definition of Terms 15
- Program Development Flow 18
  - Compiling and Assembling 19
  - Linking 19
  - Loading, Splitting, or Both 19
  - Non-Bootable Files Versus Boot-Loadable Files 20
- Boot Modes 22
  - No-Boot Mode 22
  - PROM Boot Mode 22
  - Host Boot Mode 22
- Boot Kernels 23
- Boot Streams 23
- Loader File Searches 24
- Loader File Extensions 24

Chapter 3: Loader/Splitter for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Blackfin Processors 27
- ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processor Booting 27
- ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processor Loader Guide 32
  - Loader Command Line for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processors 32
  - CCES Loader and Splitter Interface for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processors 42

Chapter 4: Loader/Splitter for ADSP-BF53x/BF561 Blackfin Processors 43
- ADSP-BF53x/BF561 Processor Booting 43
  - ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor Booting 44
  - ADSP-BF561 Processor Booting 56
  - ADSP-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management 64

ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support 67
ADSP-BF53x/BF561 Processor Loader Guide 71
Loader Command Line for ADSP-BF53x/BF561 Processors 72
CCES Loader and Splitter Interface for ADSP-BF53x/BF561 Processors 82

Chapter 5: Loader/Splitter for ADSP-BF60x Blackfin Processors 83
ADSP-BF60x Processor Booting 83
ADSP-BF60x Processor Boot Modes 84
ADSP-BF60x BCODE Field for Memory, RSI, and SPI Master Boot 85
Building a Dual-Core Application 86
CRC32 Protection 87
Block Sizes 88
ADSP-BF60x Processor Loader Guide 88
CCES Loader and Splitter Interface for ADSP-BF60x Processors 88
ROM Splitter Capabilities for ADSP-BF60x Processors 89
ADSP-BF60x Loader Collateral 90

Chapter 6: Loader/Splitter for ADSP-BF70x Blackfin Processors 91
ADSP-BF70x Processor Booting 91
ADSP-BF70x Processor Boot Modes 92
ADSP-BF70x BCODE Field for SPI Boot 92
Secure Boot and Encrypted Images 93
CRC32 Protection 94
Block Sizes 94
ADSP-BF70x Processor Loader Guide 95
CCES Loader and Splitter Interface for ADSP-BF70x Processors 95
ROM Splitter Capabilities for ADSP-BF70x Processors 96
ADSP-BF70x Loader Collateral 97

Chapter 7: Loader for ADSP-21160 SHARC Processors 99
ADSP-21160 Processor Booting 99
Power-Up Booting Process 100
Boot Mode Selection 101
ADSP-21160 Boot Modes 102
ADSP-21160 Boot Kernels 107
ADSP-21160 Interrupt Vector Table 111
ADSP-21160 Multi-Application (Multi-DXE) Management 111
Processor Loader Guide 112
Loader Command Line for Processors 112
CCES Loader Interface for Processors 116

Chapter 8: Loader for ADSP-21161 SHARC Processors 117
ADSP-21161 Processor Booting 117
Power-Up Booting Process 118
Boot Mode Selection 118
ADSP-21161 Processor Boot Modes 119
ADSP-21161 Processor Boot Kernels 127
ADSP-21161 Processor Interrupt Vector Table 130
ADSP-21161 Multi-Application (Multi-DXE) Management 130
ADSP-21161 Processor Loader Guide 131
Loader Command Line for Processors 132
CCES Loader Interface for Processors 136

Chapter 9: Loader for ADSP-2126x/2136x/2137x/214xx SHARC Processors 137
ADSP-2126x/2136x/2137x/214xx Processor Booting 137
Power-Up Booting Process 138
ADSP-2126x/2136x/2137x/214xx Processor Interrupt Vector Table 139
General Boot Definitions 139

Boot Mode Selection....139
Boot DMA Configuration Settings....140
ADSP-2126x/2136x/2137x/214xx Processor Boot Kernels....147
ADSP-2126x/2136x/2137x/214xx Processor Boot Streams....151
Multi-Application (Multi-DXE) Management....159
ADSP-2126x/2136x/2137x Processor Compression Support....161
ADSP-2126x/2136x/2137x/214xx Processor Loader Guide....165
Loader Command Line for ADSP-2126x/2136x/2137x/214xx Processors....165
CCES Loader Interface for ADSP-2126x/2136x/2137x/214xx Processors....171

Chapter 10: Splitter for SHARC Processors....173
Splitter Command Line....173
Splitter File Searches....175
Splitter Output File Extensions....175
Splitter Command-Line Switches....175

Chapter 11: File Formats....179
Source Files....179
C/C++ Source Files....179
Assembly Source Files....180
Assembly Initialization Data Files....180
Header Files....181
Linker Description Files....181
Linker Command-Line Files....181
Build Files....181
Assembler Object Files....182
Library Files....182
Linker Output Files....182
Memory Map Files....182
Bootable Loader Output Files....182
Non-Bootable Loader Output Files in Byte Format....188
Splitter Output Files....189
Debugger Files....191

Chapter 12: Utilities....193
hexutil - Hex-32 to S-Record File Converter....193
elf2dyn - ELF to Dynamically-Loadable Module Converter....194
Dynamically-Loadable Modules....194
Syntax....195
File Formats and -l Switch....196
Exported Symbols....197
Section Alignment....197
elf2elf - ELF to ELF File Converter....198
dyndump - Display the Contents of Dynamically-Loadable Modules....199
-f Family....200
Output....200
dynreloc - Relocate Dynamically-Loadable Modules....200
Explicit Mappings....201
Region Mappings....201
signtool - Sign and Encrypt Boot Streams for Secure Booting....202
Syntax....202
Output Formats....203
Key Generation for Signing....203
Key Generation for Encryption....204
Signing and Encrypting Boot Streams....204
Extracting Public Keys....204

1

Preface

Thank you for purchasing CrossCore® Embedded Studio (CCES), Analog Devices development software for Blackfin® and SHARC® processors.

Purpose of This Manual

The *Loader and Utilities Manual* contains information about the loader/splitter program for Analog Devices processors.

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. The manual assumes 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 hardware reference and programming reference manuals, that describe their target architecture.

Manual Contents

The manual contains:

- Chapter 1, Introduction, provides an overview of the loader utility (or loader) program as well as the process of loading and splitting, the final phase of the application development flow.
- Chapter 2, Loader/Splitter for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Blackfin Processors, explains how the loader/splitter utility is used to convert executable files into boot-loadable or non-bootable files for the ADSP-BF50x, ADSP-BF51x, ADSP-BF52x, ADSP-BF54x, and ADSP-BF59x Blackfin processors.

- Chapter 3, Loader/Splitter for ADSP-BF53x/BF561 Blackfin Processors, explains how the loader/splitter utility is used to convert executable files into boot-loadable or non-bootable files for the ADSP-BF53x and ADSP-BF561 Blackfin processors.
- Chapter 4, Loader/Splitter for ADSP-BF60x Blackfin Processors, explains how the loader/splitter utility (`elfloader.exe`) is used to convert executable files into boot-loadable or non-bootable files for the ADSP-BF60x Blackfin processors.
- Chapter 5, Loader for ADSP-21160 SHARC Processors, explains how the loader utility is used to convert executable files into boot-loadable files for the ADSP-21160 SHARC processors.
- Chapter 6, Loader for ADSP-21161 SHARC Processors, explains how the loader utility is used to convert executable files into boot-loadable files for the ADSP-21161 SHARC processors.
- Chapter 7, Loader for ADSP-2126x/2136x/2137x/214xx SHARC Processors, explains how the loader utility is used to convert executable files into boot-loadable files for the ADSP-2126x, ADSP- 2136x, ADSP-2137x, ADSP-2146x, ADSP-2147x, and ADSP-2148x SHARC processors.
- Chapter 8, Splitter for SHARC Processors, explains how the splitter utility is used to convert executable files into non-bootable files for the earlier SHARC processors.
- Appendix A, File Formats, describes source, build, and debugger file formats.
- Appendix B, Utilities, describes several utility programs included with CrossCore Embedded Studio, some of which run from a command line only.

What's New in This Manual

This is Revision 1.3 of the *Loader and Utilities Manual*, supporting CrossCore Embedded Studio (CCES) 1.1.0.

This revision includes support for new Blackfin processors and utility programs in the Utilities appendix.

For future revisions, this section will document loader and splitter functionality that is new to CCES, including support for new SHARC and/or Blackfin processors. In addition, modifications and corrections based on errata reports against the previous revisions of the manual will also be noted here.

Technical Support

You can reach Analog Devices processors and DSP technical support in the following ways:

- Post your questions in the processors and DSP support community at EngineerZone®:

  http://ez.analog.com/community/dsp
- Submit your questions to technical support directly at:

  http://www.analog.com/support
- E-mail your questions about processors, DSPs, and tools development software from **CrossCore Embedded Studio** or **VisualDSP++**®:

  Choose **Help > Email Support**. This creates an e-mail to *processor.tools.support@analog.com* and automatically attaches your **CrossCore Embedded Studio** or **VisualDSP++** version information and `license.dat` file.
- E-mail your questions about processors and processor applications to:

processor.tools.support@analog.com

processor.china@analog.com (Greater China support)

- Contact your Analog Devices sales office or authorized distributor. Locate one at:

  http://www.analog.com/adi-sales

- Send questions by mail to:

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

Supported Processors

The CrossCore Embedded Studio loader and utility programs support the following processor families from Analog Devices.

- Blackfin (ADSP-BFxxx)
- SHARC (ADSP-21xxx)

Refer to the CrossCore Embedded Studio online help for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices Web site and the CrossCore Embedded Studio online help.

Analog Devices Web Site

The Analog Devices Web site, *http://www.analog.com*, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

To access a complete technical library for each processor family, go to *http://www.analog.com/processors/technical_library*. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, *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 about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. *MyAnalog.com* provides access to books, application notes, data sheets, code examples, and more.

Visit *MyAnalog.com* to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

EngineerZone

EngineerZone is a technical support forum from Analog Devices. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit *http://ez.analog.com* to sign up.

Notation Conventions

Text conventions used in this manual are identified and described as follows. Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
File > Close	Titles in bold style indicate the location of an item within the CrossCore Embedded Studio IDE'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 ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
ⓘ Note:	**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.

Example	Description
Caution:	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.
Attention:	ATTENTION 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.

2

Introduction

The majority of this manual describes the loader utility (or loader) program as well as the process of loading and splitting, the final phase of the application development flow.

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

- Loader/Splitter for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Blackfin Processors
- Loader/Splitter for ADSP-BF53x/BF561 Blackfin Processors
- Loader/Splitter for ADSP-BF60x Blackfin Processors
- Loader for ADSP-21160 SHARC Processors
- Loader for ADSP-21161 SHARC Processors
- Loader for ADSP-2126x/2136x/2137x/214xx SHARC Processors
- Splitter for SHARC Processors
- File Formats
- Utilities

Definition of Terms

Loader and Loader Utility

The term **loader** refers to a **loader utility** that is part of CrossCore Embedded Studio. The loader utility post-processes one or multiple executable (`.dxe`) files, extracts segments that have been declared by the `TYPE(RAM)` command in a Linker Description File (`.ldf`), and generates a loader file (`.ldr`). Since the `.dxe` file meets the Executable and Linkable Format (ELF) standard, the loader utility is often called `elfloader` utility. See also *Loader Utility Operations*.

Splitter Utility

The **splitter utility** is part of CrossCore Embedded Studio. The splitter utility post-processes one or multiple executable (`.dxe`) files, extracts segments that have been declared by the `TYPE(ROM)` command in a Linker Description File (`.ldf`), and generates a file consisting of processor instructions (opcodes). If burned into an EPROM or flash memory device

connected to the target processor's system bus, the processor can directly fetch and execute these instructions. See also *Splitter Utility Operations*.

Splitter and loader jobs can be managed either by separate utility programs or by the same program (see *Non-Bootable Files Versus Boot-Loadable Files*). In the latter case, the generated output file can contain code instructions and boot streams.

Loader File

A **loader file** is generated by the loader utility. The file typically has the `.ldr` extension and is often called an LDR file. Loader files can meet one of multiple formats. Common formats are Intel hex-32, binary, or ASCII representation. Regardless of the format, the loader file describes a boot image, which is the binary version of the loader file. See also *Non-Bootable Files Versus Boot-Loadable Files*.

Loader Command Line

If invoked from a command-line prompt, the loader and splitter utilities accept numerous control switches to customize the loader file generation.

Loader Properties Page

The **loader properties page** is part of the **Tool Settings** dialog box in the IDE. The properties page is a graphical tool that assists in composing the loader utility's command line.

Boot Mode

Most processors support multiple boot modes. A **boot mode** is determined by special input pins that are interrogated when the processor awakes from either a reset or power-down state. See also *Boot Modes*.

Boot Kernel

A **boot kernel** is software that runs on the target processor. It reads data from the boot source and interprets the data as defined in the boot stream format. The boot kernel can reside in an on-chip boot ROM or off-chip ROM device. Often, the kernel has to be prebooted from the boot source before it can be executed. In this case, the loader utility puts a default kernel to the front of the boot image, or, allows the user to specify a customized kernel. See also *Boot Kernels*.

Boot ROM

A **boot ROM** is an on-chip read-only memory that holds the boot kernel and, in some cases, additional advanced booting routines.

Second-Stage Loader

A **second-stage loader** is a special boot kernel that extends the default booting mechanisms of the processor. It is typically booted by a first-stage kernel in a standard boot mode configuration. Afterward, it executes and boots in the final applications. See also *Boot Kernels*.

Boot Source

A **boot source** refers to the interface through which the boot data is loaded as well as to the storage location of a boot image, such as a memory or host device.

Boot Image

A **boot image** that can be seen as the binary version of a loader file. Usually, it has to be stored into a physical memory that is accessible by either the target processor or its host device.

Often it is burned into an EPROM or downloaded into a flash memory device using the Programmer plug-in.

The boot image is organized in a special manner required by the boot kernel. This format is called a boot stream. A boot image can contain one or multiple boot streams. Sometimes the boot kernel itself is part of the boot image.

Boot Stream

A **boot stream** is basically a list of boot blocks. It is the data structure that is processed and interpreted by the boot kernel. The loader utility generates loader files that contain one or multiple boot streams. A boot stream often represents one application. However, a linked list of multiple application-level boot streams is referred to as a boot stream.

Boot Host

A **boot host** is a processor or programmable logic that feeds the device configured in a slave boot mode with a boot image or a boot stream.

Boot Block

Multiple **boot blocks** form a boot stream. These blocks consist of boot data that is preceded by a block header. The header instructs the boot kernel how to interpret the payload data. In some cases, the header may contain special instructions only. In such blocks, there is likely no payload data present.

Boot Code

Boot code refers to all boot-relevant ROM code. Boot code typically consists of the preboot routine and the boot kernel.

Boot Strapping

If the boot process consists of multiple steps, such as preloading the boot kernel or managing second-stage loaders, this is called **boot strapping**.

Initialization Code

Initialization code or *initcode* is part of a boot stream for Blackfin processors and is a special boot block. While normally all boot blocks of an application are booted in first and control is passed to the application afterward, the initialization code executes at boot time. It is common that an initialization code is booted and executed before any other boot block. This initialization code can customize the target system for optimized boot processing.

Global Header

Some boot kernels expect a boot stream to be headed by a special information tag. The tag is referred to as a **global header**.

Callback Routine

Some processors can optionally call a user-defined routine after a boot block has been loaded and processed. This is referred to as a **callback routine**. It provides hooks to implement checksum and decompression strategies.

Slave Boot

The term **slave boot** spans all boot modes where the target processor functions as a slave. This is typically the case when a host device loads data into the target processor's memories. The target processor can wait passively in idle mode or support the host-controlled data transfers actively. Note that the term host boot usually refers only to boot modes that are based on so-called host port interfaces.

Master Boot

The term **master boot** spans all boot modes where the target processor functions as master. This is typically the case when the target processor reads the boot data from parallel or serial memories.

Boot Manager

A **boot manager** is firmware that decides which application is to be booted. An application is usually represented as a project in the IDE and stored in a `.dxe` file. The boot manger itself can be managed within an application `.dxe` file, or have its own separate `.dxe` file. Often, the boot manager is executed by initialization code.

In slave boot scenarios, boot management is up to the host device and does not require special tools support.

Multi-dxe Boot

A loader file can contain data of multiple application (`.dxe`) files if the loader utility was invoked by specifying multiple `.dxe` files. Either a boot manager decides which application is to be booted exclusively or, alternatively, one application can terminate and initiate the next application to be booted. In some cases, a single application can also consist of multiple `.dxe` files.

Next .dxe File Pointer

If a loader file contains multiple applications, some boot stream formats enable them to be organized as a linked list. The next `.dxe` pointer (NDP) is simply a pointer to a location where the next application's boot stream resides.

Preboot Routine

A preboot routine is present in the boot ROM of parts that feature OTP memory on a processor. Preboot reads OTP memory and customizes several MMR registers based on factory and user instructions, as programmed to OTP memory. A preboot routine executes prior to the boot kernel.

Program Development Flow

The **Program Development Flow** figure is a simplified view of the application development flow.

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.

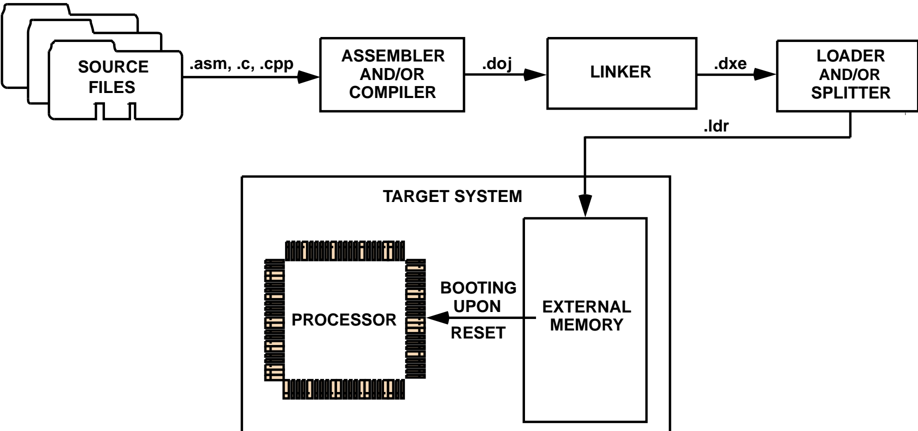


Figure 1. Program Development Flow

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. The assembler and compiler are documented in the *Assembler and Preprocessor Manual* and *C/C++ Compiler Manual*, which are part of the online help.

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 ELF standard, 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 debugging targets, such as the simulator or emulator. Refer to the *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 **boot process, introduction to booting**. Therefore, the linker output must be transformed to a format readable by the processor. This process is handled by the loader and/or splitter utility. The loader/splitter utility uses the debugged and tested executable files as well as shared memory and overlay files as inputs to yield a processor-loadable file.

CrossCore Embedded Studio includes these loader and splitter utilities:

- `elfloader.exe` (loader utility) for Blackfin and SHARC processors. The loader utility for Blackfin processors also acts as a ROM splitter when evoked with the corresponding switches.
- `elfspl21k.exe` (ROM splitter utility) for earlier SHARC processors. Starting with the ADSP-214xx processors, splitter functionality is available through `elfloader.exe`.

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's PROM space on an EZ-KIT Lite®/EZ-Board® board via the Device Programmer plug-in. Refer to the online help for information on the Device Programmer.
- Use the IDE to simulate booting in a simulator session. 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 for 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 into an external memory device (burned into EPROM) within your target system. The loader utility 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 the File Formats appendix.)

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 utility to produce a non-bootable memory image file.

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

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

  In the IDE, with the addition of the `-readall` switch, the loader utility for the ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Blackfin processors can call the splitter program automatically. For more information, see `-readall #`.
- For earlier SHARC processors, splitter operations are handled by the splitter program, `elfspl21k.exe`. Starting with the ADSP-214xx processors, splitter functionality is available through `elfloader.exe`.

Loader Utility Operations

Common tasks performed by the loader utility can include:

- Processing loader properties or command-line switches.

- Formatting the output `.ldr` file according to user specifications. Supported formats are binary, ASCII, Intel hex-32, and more. Valid file formats are described in the File Formats appendix.
- 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.

Using CCES Loader Interface

Run the loader utility from the CrossCore Embedded Studio command line (`elfloader`) or within the IDE. To use the loader utility for a project, the project's output (artifact) type must be a loader file (`.ldr`). The IDE invokes the `elfloader.exe` utility to build the output loader file.

To run the loader utility within the IDE and/or modify the loader settings, use the loader pages. The pages (also called *properties pages*) show the default loader properties for the project's target processor. The loader properties control how the loader utility processes executable files into boot-loadable files, letting you select and modify kernels, boot modes, and output file formats. Settings on the loader properties pages correspond to switches typed on the `elfloader` command line.

See the CCES online help for more information about the loader interface.

Splitter Utility Operations

Splitter utility operations depend on the processor family, splitter properties, and command-line switches, which control which utility is invoked, and how it processes executable files into non-bootable files:

- For Blackfin processors, the loader utility includes the ROM splitter capabilities invoked through the CCES IDE or command line. The IDE settings correspond to switches typed on the `elfloader` command line. Refer to the CCES online help for more information.
- For SHARC processors earlier than ADSP-214xx, the splitter functionality is available in CCES via the command-line (`elfspl21k.exe`). Refer to the Splitter for SHARC Processors chapter for more information.
- For SHARC ADSP-214xx processors, the loader utility includes section splitting capabilities via the `-splitter` switch.). Refer to the Splitter for SHARC Processors chapter for more information.

Using CCES Splitter Interface

For Blackfin and SHARC processors, use the splitter capabilities of the loader from the CrossCore Embedded Studio command line (`elfloader`) or within the IDE. To use the splitter capabilities for a project, the project's output (artifact) type must be a loader file (`.ldr`). The IDE invokes the `elfloader.exe` utility to build the output loader file.

For Blackfin processors, use the CCES splitter page. The page (also called *properties page*) show the default splitter properties for the project's target processor. The properties control how the loader utility processes executable files into non-bootable files, letting you select and modify address masks, data packing options, and output file formats.

For the ADSP-214xx SHARC processors, use the CCES **Additional Options** properties page of the loader and specify the `-splitter`*`section-name`* switch.

Settings on the properties pages correspond to switches typed on the `elfloader` command line. See the CCES online help for more information about the loader/splitter interface.

Boot Modes

Once an executable file is fully debugged, the loader utility is ready to convert the executable file into a processor-loadable (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 utility 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 a boot kernel (described in *Boot Kernels*) parses the boot stream and initializes memories accordingly. The boot kernel runs on the target processor. Depending on the architecture, the boot 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 signaled 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 boot 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 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-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 do not require to load a boot kernel-a kernel is already present in the on-chip boot ROM. It allows the entire application program's body to be booted into the internal and external memories of the processor. The boot ROM has the capability to parse address and count information for each bootable block.

Boot Streams

The loader utility's output (`.ldr` file) is essentially the same executable code as in the input `.dxe` file; the loader utility simply repackages the executable as shown in the **.dxe Files Versus .ldr Files** figure.

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 and destination in the processor's memory. Depending on the processor family, there can be additional information in the tag. Common block types are "zero" (memory is filled with `0`s); nonzero (code or data); and final (code or data). Depending on the processor family, there can be other block types.

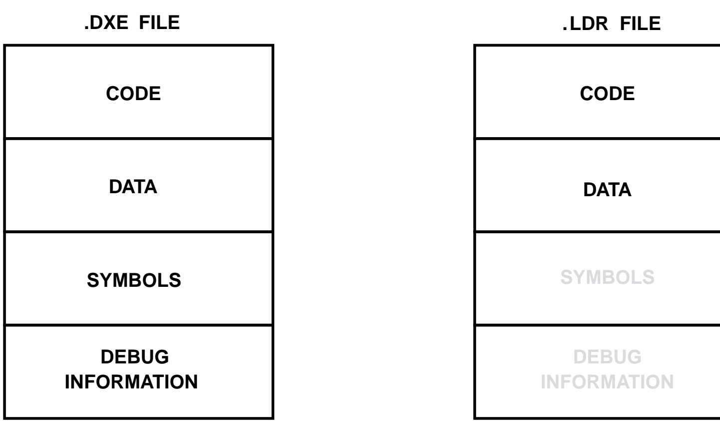


Figure 2. .dxe Files Versus .ldr Files

Loader File Searches

File searches are important in the loader utility operations. The loader utility 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 utility searches only in that location for the file.
- Default directory-If path information is not included in the file name, the loader utility searches for the file in the current working directory.
- Overlay and shared memory files-The loader utility 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 utility 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.

Loader File Extensions

Some loader switches take a file name as an optional parameter. The **File Extensions** table lists the expected file types, names, and extensions.

Table 1. 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 utility expects the overlay input files with the .ovl file extension, shared memory input files with the .sm extension, or both but does not expect those files to appear on a command line or properties pages. The loader utility expects to find these files in the directory of the associated .dxe files, in the current working directory, or in the directory specified for the .ldf file.

3

Loader/Splitter for ADSP-BF50x/BF51x/BF52x/BF54x/ BF59x Blackfi Processors

This chapter explains how the loader/splitter utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable or non-bootable files for the ADSP-BF50x, ADSP-BF51x, ADSP-BF52x, ADSP-BF54x, and ADSP-BF59x Blackfin processors.

Refer to the Introduction chapter for the loader utility overview. Loader operations specific to the ADSP-BF50x/ BF51x/BF52x/BF54x and ADSP-BF59x Blackfin processors are detailed in the following sections.

- *ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processor Booting*

  Provides general information on various boot modes, including information on second-stage kernels.
- *ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processor Loader Guide*

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

ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processor Booting

Refer to the processor's data sheet and hardware reference manual for detailed information on system configuration, peripherals, registers, and operating modes.

- Blackfin processor data sheets can be found at:

  http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/data-sheets/resources/index.html.
- Blackfin processor manuals can be found at:

  http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/manuals/resources/index.html or downloaded into the CCES IDE via **Help > Install New Software**.

The following table lists the part numbers that currently comprise the ADSP-BF50x/BF51x/BF52x/BF54x/BF59x families of Blackfin processors. Future releases of CrossCore Embedded Studio may support additional processors.

Table 2. ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Part Numbers

Processor Family	Part Numbers
ADSP-BF504	ADSP-BF504, ADSP-BF504F, ADSP-BF506
ADSP-BF518	ADSP-BF512, ADSP-BF514, ADSP-BF516, ADSP-BF518
ADSP-BF526	ADSP-BF522, ADSP-BF524, ADSP-BF526
ADSP-BF527	ADSP-BF523, ADSP-BF525, ADSP-BF527
ADSP-BF548	ADSP-BF542, ADSP-BF544, ADSP-BF547, ADSP-BF548, ADSP-BF549
ADSP-BF548M	ADSP-BF542M, ADSP-BF544M, ADSP-BF547M, ADSP-BF548M, ADSP-BF549M
ADSP-BF592	ADSP-BF592-A

Upon reset, an ADSP-BF50x/BF51x/BF52x/BF54x/BF59x processor starts fetching and executing instructions from the on-chip boot ROM at address `0xEF00 0000`. The boot ROM is an on-chip read-only memory that holds a boot kernel program to load data from an external memory or host device. The boot ROM details can be found in the corresponding hardware reference manual.

There are other boot modes available, including idle (no-boot) mode. The processor transitions into the boot mode sequence configured by the `BMODE` pins; see the **ADSP-BF50x Boot Modes**, **ADSP-BF51x Boot Modes**, **ADSP-BF52x/BF54x**, and**ADSP-BF59x Boot Modes** tables. The `BMODE` pins are dedicated mode-control pins; that is, no other functions are performed by the pins. The pins can be read through bits in the system configuration register (`SYSCR`).

Table 3. ADSP-BF50x Boot Modes

Boot Source	BMODE[2:0]	Start Address
Idle (no-boot)	`000`	N/A
Stacked parallel flash memory in async mode	`001`[1]	`0x2000 0000`
Stacked parallel flash memory in sync burst mode	`010`	`0x2000 0000`
SPI0 master from SPI memory	`011`	`0x0000 0000`

[1] ADSP-BF504 processors do not support BMODE 001 or 010 because they have no internal flash.

Boot Source	BMODE[2:0]	Start Address
SPI0 slave from host device	100	N/A
16-bit PPI host	101	N/A
Reserved	110	N/A
UART0 slave from UART host	111	N/A

Table 4. ADSP-BF51x Boot Modes

Boot Source	BMODE[2:0]	Start Address
Idle (no-boot)	000	N/A
8- or 16-bit external flash memory (default mode)	001	0x2000 0000
Internal SPI memory	010	0x2030 0000
External SPI memory (EEPROM or flash)	011	0x0000 0000
SPI0 host device	100	N/A
One-time programmable (OTP) memory	101	N/A
SDRAM memory	110	N/A
UART0 host	111	N/A

Table 5. ADSP-BF52x/BF54x Boot Modes

Boot Source	BMODE[3:0]	Start Address
Idle (no-boot)	0000	N/A
8- or 16-bit external flash memory (default mode)	0001	0x2000 0000

Boot Source	BMODE[3:0]	Start Address
16-bit asynchronous FIFO	0010	0x2030 0000
8-, 16-, 24-, or 32-bit addressable SPI memory	0011	0x0000 0000
External SPI host device	0100	N/A
Serial TWI memory	0101	0x0000 0000
TWI host	0110	N/A
UART0 host on ADSP-BF52x processors; UART1 host on ADSP-BF54x processors	0111	N/A
UART1 host on the ADSP-BF52x processors; reserved on ADSP-BF54x processors	1000	N/A
Reserved	1001	N/A
SDRAM/DDR	1010	0x0000 0010
OTP memory	1011	default page 0x40
8- or 16-bit NAND flash memory	1100, 1101	0x0000 0000
16-bit host DMA	1110	N/A
8-bit host DMA	1111	N/A

Table 6. ADSP-BF59x Boot Modes

Boot Source	BMODE[2:0]	Start Address
Idle (no-boot)	000	N/A
Reserved	001	N/A
External serial SPI memory using SPI1	010	N/A
SPI host device using SPI1	011	N/A

Boot Source	BMODE[2:0]	Start Address
External serial SPI memory using SPI0	`100`	N/A
PPI host	`101`	N/A
UART host	`110`	N/A
Internal L1 ROM	`111`	`0x2000 0000`

In general, there are two categories of boot modes: master and slave. In master boot modes, the processor actively *loads data* from parallel or serial memory devices. In slave boot modes, the processor *receives data* from parallel or serial memory devices.

The Blackfin loader utility generates `.ldr` files that meet the requirements of the target boot mode; for example:

- **HOSTDP** (`-b HOSTDP`)

  When building for the HOSTDP boot, the loader utility aligns blocks with payload to the appropriate FIFO depth for the target processor. Note that HOSTDP differs from other boot modes in the default setting for the *`-NoFillBlock`* switch. The HOSTDP boot mode directs the loader not to produce fill (zero) blocks by default.

  To enable fill blocks for HOSTDP builds in the CCES IDE:

  1. Open the **Properties** dialog box for the project.
  2. Choose **C/C++ Build > Settings**. The **Tool Settings** page appears.
  3. Click **Additional Options** under **CrossCore Blackfin Loader**. The loader **Additional Options** properties page appears.
  4. Click **Add** (+). The **Enter Value** dialog box appears.
  5. In **Additional Options**, type in `-FillBlock`.
  6. Click **OK** to close the dialog box.
  7. Click **Apply**.

- **NAND** (`-b NAND`)

  When building for NAND boot, the loader utility appends 256 bytes to the boot NAND loader stream, a requirement for the boot kernel for the prefetch mechanism. While fetching one 256 byte block of data, it prefetches the next 256 byte block of data. The padding ensures that the final block of the loader stream is programmed, and the error correction parity data is written.

- **OTP** (`-b OTP`)

  When building for OTP boot, no width selection is used. OTP is always a 32-bit internal transfer. Use Intel hex-32 format for the OTP boot mode and provide the offset to the start address for the OTP page. The OTP flash programmer requires the offset to the start address for the OTP page when Intel hex loader format is selected. To specify the start address in the CCES IDE:

  1. Open the **Properties** dialog box for the project.
  2. Choose **C/C++ Build > Settings**. The **Tool Settings** page appears.

3. Click **General** under **CrossCore Blackfin Loader**. The loader **General** properties page appears.
4. In **Boot format (-f)**, ensure **Intel hex** is selected.
5. Disable **Use default start kernel**. The **Start address (-p)** is enabled.
6. In **Start address (-p)**, enter the page number multiplied by 16. For example, if you are building for OTP boot and writing to page `0x40L`, specify start address `0x400`.
7. Click **Apply**.

   On the loader command-line, the above example corresponds to:

   ```
   -b otp -f hex -p 0x400
   ```

Refer to the CCES online help for information about the loader properties pages.

ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processor Loader Guide

Loader utility operations depend on the loader properties, which control how the utility processes executable files. You select features, such as boot modes, boot kernels, and output file formats via the properties. The properties are specified on the loader utility's command line or the **Tool Settings** dialog box in the IDE (**CrossCore Blackfin Loader** pages). The default loader settings for a selected processor are preset in the IDE.

Note:

The IDE's **Tool Settings** correspond to switches displayed on the command line.

These sections describe how to produce a bootable (single and multiple) or non-bootable loader file:

- *Loader Command Line for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processors*
- *CCES Loader and Splitter Interface for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processors*

Loader Command Line for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processors

The loader utility uses the following command-line syntax for the ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Blackfin processors.

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 (`.dxe`) file 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 utility to process the files. Enclose long file names within straight quotes, `"long file name"`.
- `-proc processor` - Part number of the processor (for example, `-proc ADSP-BF542`) for which the loadable file is built. Provide a processor part number for every input `.dxe` if designing multiprocessor systems; see the part numbers in *ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processor Booting*.

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

Note:

Command-line switches can 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 utility processes the input files in the order presented on the command line.

Loader Command-Line Switches for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x

A summary of the loader command-line switches for the ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Blackfin processors appears in the following table. For a quick on-line help on the switches available for a specific processor: for an ADSP-BF548 processor, use the following command line.

```
elfloader -proc ADSP-BF548 -help
```

Table 7. ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Loader Command-Line Switches

Switch	Description
`-b {flash|prom|spimaster|spislave|twimaster|twislave|uart|fifo|otp|nand|ppi|hostdp}`	The `-b` switch directs the loader utility to prepare a boot-loadable file for the specified boot mode. The default boot mode for all processors described in this chapter is PROM/FLASH. Other valid boot modes include: • SPI master (`-b spimaster`) for the ADSP-BF50x, BF51x/52x/54x/54xM, and ADSP-BF59x processors. • SPI slave (`-b spislave`) for the ADSP-BF50x, BF51x/52x/54x/54xM, and ADSP-BF59x processors. • UART (`-b uart`) for the ADSP-BF50x, BF51x/52x/54x/54xM, and ADSP-BF59x processors. • TWI master (`-b twimaster`) for the ADSP-BF52x/54x/54xM processors. • TWI slave (`-b twislave`) for the ADSP-BF52x/54x/54xM processors. • FIFO (`-b fifo`) for the ADSP-BF52x/54x/54xM processors. • OTP (`-b otp`) for the ADSP-BF51x/52x/54x/54xM processors. • NAND (`-b nand`) for the ADSP-BF52x/54x/54xM processors. • PPI (`-b ppi`) - for the ADSP-BF50x and BF59x processors. • HOSTDP (`-b hostdp`) for the ADSP-BF52x, BF544/7/8/9, and BF544M/547M/548M/549M processors. See additional information in this chapter on the HOSTDP, NAND, and OTP boot modes.

Switch	Description
`-CRC32 [polynomial]`	The `-CRC32` (polynomial coefficient) switch directs the loader utility to generate CRC32 checksum. Use a polynomial coefficient if specified; otherwise, use default `0xD8018001`. This switch inserts an initcode boot block that calls an initialization routine residing in the on-chip boot ROM. The argument field of the boot block provides the used polynomial. The loader utility calculates the CRC checksum for all subsequent data blocks and stores the result in the block header's argument field.
`-callback sym=`*`symbol`*`[arg= const32 ]`	The `-callback` switch takes a `sym=`*`symbol`* (no spaces) assignment. The switch directs the loader utility to isolate the named subroutine into a separate block, set the block header's `BFLAG_CALLBACK` flag, and fill in the block header's argument field with the specified constant 32-bit values. The switch is used for boot-time callbacks. The callback is guaranteed to be made prior to the target address of `sym=`*`symbol`*. **Note:** The `-callback` cannot be used with `-CRC32`.
`-dmawidth {8|16|32}`	The `-dmawidth {8|16|32}` switch specifies a DMA width (in bits) for memory boot modes. It controls the `DMACODE` bit field issued to the boot block headers by the `-width` switch. For FIFO boot mode, 16 is the only DMA width. SPI, TWI, and UART modes use 8-bit DMA.
`-f {hex|ascii|binary|include}`	The `-f {hex|ascii|binary|include}` switch specifies the format of a boot-loadable file: Intel hex-32, ASCII, binary, or include. If the `-f` switch does not appear on the command line, the default file format is hex for flash/PROM boot modes; and ASCII for other boot modes.
`-FillBlock`	FILL blocks are enabled by default for all boot modes, except `-b hostdp`.

Switch	Description
`-h` or `-help`	The `-help` switch 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-BF542 -h` to obtain help for the ADSP-BF542 processor.
`-init` *`filename.dxe`*	The `-init` *`filename.dxe`* switch directs the loader utility to include the initialization code from the named executable file. The loader utility places the code and data from the initialization sections at the top of the boot stream. The boot kernel loads the code and then calls it. It is the code's responsibility to save/restore state/registers and then perform an RTS back to the kernel. Initcodes can be written in C language and are compliant to C calling conventions. The `-init` *`filename.dxe`* switch can be used multiple times to specify the same file or different files a number of times. The loader utility will place the code from the initialization files in the order the files appear on the command line. *ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Multi-DXE Loader Files*
`-initcall sym=`*`sym_symbol`* `at=`*`at_symbol`* `[stride=`*`DstAddrGap`* `count=`*`times`*`]`	While the `-init` *`filename.dxe`* switch integrates initialization codes managed by a separate application program, the `-initcall` switch controls calls to initialization subroutines that are part of the same application. The `-initcall` switch directs the loader utility to dispatch a boot-time initialization call to the `sym` subroutine when the `at` symbol is encountered and loaded. The `stride` and `count` parameters are optional: • If an optional `stride=` constant 32-bit value is specified, the loader utility insets the target program call every `stride=` target address locations. • If an optional `count=` constant 32-bit value is specified, the loader utility insets the target program call `count=` times, every `stride=` target address locations apart. A `count` value without a `stride` value is an error. For example, the following command line:

Switch	Description
	`-initcall sym=_initcode at=_othersymbol stride=0x100 count=5` results in function `_initcode` being called five times the first time, just prior to data in `_othersymbol` being booted. Thereafter, every 256 destination load addresses `_initcode` is called again until a total of five calls have been made. `-initcall` restrictions: • `-initcall` target (*`sym_symbol`*) must be a routine entry point, end with an RTS. It can be written in C language and can rely on the presence of a stack. However, the routine must not call any libraries, not rely on compiler run-time environment (such as heaps) - must be self-contained • `-initcall` subroutine must be previously loaded and still in memory • `-initcall` subroutine cannot contain any forward references to code not yet loaded • *`sym_symbol`* address must be less than *`at_symbol`* address For more information, see *ADSP-BF50x/BF51x/BF52x/BF54x/ BF59x Multi-DXE Loader Files*.
`-kb {flash|prom|spimaster| spislave|uart|twimaster|twislave| fifo|nand|ppi}`	The `-kb` switch specifies the boot mode for the initialization code and/or boot kernel output file if two output loader files are selected. The `-kb` switch must be used in conjunction with the `-o2` switch. If the `-kb` switch is absent from the command line, the loader utility generates the file for the init and/or boot kernel code in the same boot mode as used to output the user application program. Other valid boot modes include: • PROM/FLASH (`-kb prom` or `-kb flash`) - the default boot mode for all processors described in this chapter. • SPI master (`-kb spimaster`) for the ADSP-BF50x, BF51x/52x/54x/54xM, and ADSP-BF59x processors. • SPI slave (`-kb spislave`) for the ADSP-BF50x, BF51x/52x/54x/ 54xM, and ADSP-BF59x processors. • UART (`-kb uart`) for the ADSP-BF50x, BF51x/52x/54x/54xM, and ADSP-BF59x processors.

Switch	Description
	• TWI master (`-kb twimaster`) for the ADSP-BF52x/54x/54xM processors. • TWI slave (`-kb twislave`) for the ADSP-BF52x/54x/54xM processors. • FIFO (`-kb fifo`) for the ADSP-BF52x/54x/54xM processors. • NAND (`-kb nand`) - for the ADSP-BF52x/54x/54xM processors. • PPI (`-kb ppi`) for the ADSP-BF50x and BF59x processors.
`-kf {hex|ascii|binary|include}`	The `-kf {hex|ascii|binary|include}` switch specifies the output file format (hex, ASCII, binary, include) for the initialization and/or boot kernel code if two output files from the loader utility are selected: one file for the init code and/or boot kernel and one file for user application code. The `-kf` switch must be used in conjunction with the`-o2` switch. If `-kf` is absent from the command line, the loader utility generates the file for the initialization and /or boot kernel code in the same format as for the user application code.
`-kp #`	The `-kp #` switch specifies a hex flash/PROM start address for the initialization and/or boot kernel code. A valid value is between `0x0` and `0xFFFFFFFF`. The specified value is ignored when neither kernel nor initialization code is included in the loader file.
`-kwidth {8|16|32}`	The `-kwidth {8|16|32}` switch specifies an external memory device width (in bits) for the initialization code and/or the boot kernel if two output files from the loader utility are selected. If `-kwidth` is absent from the command line, the loader utility generates the boot kernel file in the same width as the user application program. The *`-kwidth #`* switch must be used in conjunction with the *`-o2`* switch.
`-l` *`userkernel.dxe`*	The `-l` *`userkernel.dxe`* switch specifies the user boot kernel file. There is no default kernel for the ADSP-BF50x/BF51x/BF52x/BF54x/BF59x processors.

Switch	Description
`-M`	The `-M` switch generates make dependencies only, no output file is generated.
`-maskaddr #`	The `-maskaddr #` switch masks all EPROM address bits above or equal to #. For example, `-maskaddr 29` (default) masks all the bits above and including `A29` (`AND`ed 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]. The `-maskaddr#` switch requires `-romsplitter` and affects the ROM section address only.
`-MaxBlockSize #`	The `-MaxBlockSize #` switch specifies the maximum block size up to `0x7FFFFFF0`. The value must be a multiple of 4. The default maximum block size is `0xFFF0` or the value specified by the `-MaxBlockSize` switch.
`-MaxFillBlockSize #`	The `-MaxFillBlockSize #` switch specifies the maximum fill block size up to `0xFFFFFFF0`. The value must be a multiple of two. The default fill block size is `0xFFF0`.
`-MM`	The `-MM` switch generates make dependencies while producing the output files.
`-Mo` *`filename`*	The `-Mo` *`filename`* switch writes make dependencies to the named file. Use the `-Mo` switch with either `-M` or `-MM`. If `-Mo` is absent, the default is a `<stdout>` display.
`-Mt` *`target`*	The `-Mt` *`target`* switch specifies the make dependencies target output file. Use the `-Mt` switch with either `-M` or `-MM`. If `-Mt` is not present, the default is the name of the input file with an `.ldr` extension.
`-NoFillBlock`	The `-NoFillBlock` switch directs the loader utility not to produce FILL blocks, zero, or repeated blocks. The `-NoFillBlock` switch is set automatically in the HOSTDP (`-b HOSTDP`) boot mode.

Switch	Description
`-NoInitCode`	The `-NoInitCode` switch directs the loader utility not to expect an init code file. The loader utility may expect an init code file, specified through the `-init` *`filename.dxe`* switch if the application has an external memory section. The init code file should contain the code to initialize registers for external memory initialization.
`-o` *`filename`*	The `-o` *`filename`* switch directs the loader utility to use the specified file as the name of the loader utility's output file. If the *`filename`* is absent, the default name is the root name of the input file with an `.ldr` extension.
`-o2`	The `-o2` switch directs the loader utility to produce two output files: one file for code from the initialization block and/or boot kernel and one file for user application code. To have a different format, boot mode, or output width for 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, respectively. Combine `-o2` with `-l` *`filename`* and/or `-init` *`filename.dxe`*.
`-p #`	The `-p #` switch specifies a hex flash/PROM output start address for the application code. A valid value is between `0x0` and `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). For boot mode `-b OTP` and `-f hex` format, use `-p` to supply the offset to the start address for the OTP page (page # multiplied by 16).
`-proc` *`processor`*	The `-proc` *`processor`* switch specifies the target processor. The *`processor`* can be one of the processors listed in the **ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Part Numbers** table in *ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processor Booting*.
`-quickboot sec=`*`section`*	The `-quickboot` switch takes a `sec=`*`section`* (no spaces) assignment.

Switch	Description
	The switch directs the loader utility to mark blocks within the LDF-defined output section name with the`BFLAG_QUICKBOOT` flag. The switch is used to mark blocks to skip on warm-boot cycles.
`-readall #`	The `-readall #` switch directs the loader utility to integrate fixed-position ROM sections within the loader boot stream. The switch calls the splitter utility as a transparent sub-process to the loader utility. Memory segments declared with the `TYPE(ROM)` command in the LDF file are processed by the splitter. Segments with the `TYPE(RAM)` command emit to the boot stream. The valid switch argument is an integer between `0` and `32`, where `29` is the default. In the resulting loader (`.ldr`) file in Intel hex-32 format, the ROM-based splitter data is merged with the RAM-based loader stream. The `#` argument is similar to the `-maskaddr #` switch, which designates the upper PROM address bit position for extended address mapping. The splitter utility is required to provide the `-maskaddr #` parameter to the loader utility to generate a ROM-based splitter stream, but the required splitter parameter is not available on the loader command line. The loader utility solves this requirement by supporting the `-readall#` switch.
`-romsplitter`	The `-romsplitter` switch creates a non-bootable image only. This switch overwrites the `-b` switch and any other switch bounded by the boot mode. 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 Intel hex and ASCII formats.
`-save [sec=`*`section`*`]`	The `-save` switch takes a *`sec=section`* (no spaces) assignment. The switch directs the loader utility to mark blocks within the LDF-defined section name with the `BFLAG_SAVE` flag. The switch is used to mark blocks to archive for low-power or power-fail cycles.

Switch	Description
`-si-revision [none|any|x.x]`	Sets revision for the build, with *x.x* being the revision number for the processor hardware. If `-si-revision` is not used, the target is a default revision from the supported revisions.
`-v`	The `-v` switch directs the loader utility to output verbose loader messages and status information as the loader processes files.
`-width {8|16|32}`	The `-width {8|16|32}` switch specifies an external memory device width (in bits) to the loader utility in flash/PROM boot mode (default is 8). For FIFO boot mode, the only valid width is `16`. For SPI, TWI, and UART boot modes, the only valid width is 8.

ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Multi-DXE Loader Files

An ADSP-BF50x/BF51x/BF52x/BF54x/BF59x loader (`.ldr`) file can contain data of multiple application (`.dxe`) files. At boot time, the boot kernel boots one application file exclusively, or one application file initiates the boot of the next application file. In some cases, a single application can consist of multiple `.dxe` files.

Initialization code is a subroutine called at boot time. Unlike the ADSP-BF53x/BF56x processors, the ADSP-BF50x/BF51x/BF52x/BF54x/BF59x processors support initcode written in both assembly and C.

CrossCore Embedded Studio supports two methods of integrating multiple initcode subroutines:

- The `-init filename.dxe`command-line switch expects a `.dxe` file. The initcode is managed by a separate project. If the initcode is written in C language, ensure that the `.dxe` file does not include the CRT code because the boot kernel expects a subroutine.

  The `-init filename.dxe` switch can be used multiple times to specify the same file or different files a number of times. The loader utility places the code from the initialization files in the order the files appear on the command line. All initcodes are inserted after the first regular `.dxe` file.

  The loader utility equips every initcode with a dedicated first boot block, which has the `BFLAG_FIRST` flag set. Initcodes, however, do not feature a final block; they are terminated by a boot block, tagged by the `BFLAG_INIT` flag. Therefore, in absence of the `BFLAG_FINAL` flag, the boot kernel continues processing of the subsequent `.dxe` data after finishing execution of the initcode.
- The `-initcall sym=sym_symbol`command-line switch relies on initcode subroutines that are part of the same project. Initcode subroutines invoked by the `-initcall` switch are not accompanied by any first boot blocks with the `BFLAG_FIRST` flag set. In the loader file, the initcode subroutines translate to boot blocks tagged by the `BFLAG_INIT` flag.

When writing an initcode subroutine in C, ensure that the code does not rely on libraries or heap support, which may not be available in memory by the time the initcode executes. An initcode routine is expected to return

properly to the boot kernel by an RTS instruction and to meet C-language calling conventions (see the *C/C++ Compiler and Library Manual for Blackfin Processors*).

Refer to the initcode examples provided with the installation in `<install_path>/Blackfin/ldr/init_code`.

CCES Loader and Splitter Interface for ADSP-BF50x/BF51x/BF52x/BF54x/BF59x Processors

Once a project is created in the CrossCore Embedded Studio IDE, you can change the project's output (artifact) type.

The IDE invokes the `elfloader.exe` utility to build the output loader file. To modify the default loader properties, use the project's **Tool Settings** dialog box. The controls on the pages correspond to the loader command-line switches and parameters (see *Loader Command-Line Switches for ADSP- BF50x/BF51x/BF52x/BF54x/BF59x*). The loader utility for Blackfin processors also acts as a ROM splitter when evoked with the corresponding switches.

The loader pages (also called *loader properties pages*) show the default loader settings for the project's target processor. Refer to the CCES online help for information about the loader/splitter interface.

4

Loader/Splitter for ADSP-BF53x/BF561 Blackfi Processors

This chapter explains how the loader/splitter utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable or non-bootable files for the ADSP-BF53x and ADSP-BF561 Blackfin processors.

Refer to the Introduction chapter for the loader utility overview. Loader operations specific to the ADSP-BF53x and ADSP-BF561 Blackfin processors are detailed in the following sections.

- *ADSP-BF53x/BF561 Processor Booting*

  Provides general information on various boot modes.
- *ADSP-BF53x/BF561 Processor Loader Guide*

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

ADSP-BF53x/BF561 Processor Booting

At power-up, after a reset, the processor transitions into a boot mode sequence configured by the `BMODE` pins. The `BMODE` pins are dedicated mode-control pins; that is, no other functions are performed by these pins. The pins can be read through bits in the system reset configuration register `SYSCR`.

An ADSP-BF53x or an ADSP-BF561 Blackfin processor can be booted from an 8- or 16-bit flash/PROM memory or from an 8-,16-, or 24-bit addressable SPI memory. The ADSP-BF561 processors does not support 24-bit addressable SPI memory boot. There is also a no-boot option (bypass mode) in which execution occurs from a 16-bit external memory. For more information, refer to:

- *ADSP-BF531/BF532/BF533/BF534/BF536/BF537/ BF538/BF539 Processor Booting*
- *ADSP-BF561 Processor Booting*

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

- *ADSP-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management*
- *ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support*

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 or jumps to 16-bit external memory for execution (if `BMODE = 0`) located at `0x2000 0000`. The ROM description can be found in *ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor On-Chip Boot ROM*.

The **Boot Mode Selections for ADSP-BF531/BF532/BF533/BF538/BF539 Processors** table summarizes boot modes and execution start addresses for the named processors.

Table 8. Boot Mode Selections for ADSP-BF531/BF532/BF533/BF538/BF539 Processors

Boot Source	BMODE[1:0]	Execution Start Address	
		ADSP-BF531 ADSP-BF532	ADSP-BF533 ADSP-BF538 ADSP-BF539
Executes from a 16-bit external ASYNC bank 0 memory (no-boot mode); see *ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor No-Boot Mode*	00	0x2000 0000	0x2000 0000
8- or 16-bit flash/PROM	01	0xFFA0 8000	0xFFA0 0000
SPI host in SPI slave mode	10	0xFFA0 8000	0xFFA0 0000
8-, 16-, or 24-bit addressable SPI memory in SPI master boot mode with support for Atmel AT45DB041B, AT45DB081B, and AT45DB161B DataFlash devices	11	0xFFA0 8000	0xFFA0 0000

The **ADSP-BF534/BF536/BF537 Processor Boot Modes** table summarizes boot modes for the ADSP-BF534/BF536/BF537 processors, which in addition to all of theADSP-BF531/BF532/BF533 processor boot modes, also boot from a TWI serial device, a TWI host, and a UART host.

Table 9. ADSP-BF534/BF536/BF537 Processor Boot Modes

Boot Source	BMODE[2:0]
Executes from an external 16-bit memory connected to ASYNC bank 0; (no-boot mode or bypass on-chip boot ROM); see *ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539 Processor No-Boot Mode*	`000`
8- or 16-bit flash/PROM	`001`
Reserved	`010`
8-, 16-, or 24-bit addressable SPI memory in SPI master mode with support for Atmel AT45DB041B, AT45DB081B, and AT45DB161B DataFlash devices	`011`
SPI host in SPI slave mode	`100`
TWI serial device	`101`
TWI host	`110`
UART host	`111`

- Execute from 16-bit external memory - execution starts from address `0x2000 0000` with 16-bit packing. The boot ROM is bypassed in this mode. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).
- Boot from 8-bit or 16-bit external flash memory - the 8-bit or 16-bit flash boot routine located in boot ROM memory space is set up using asynchronous memory bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup). The boot ROM evaluates the first byte of the boot stream at address `0x2000 0000`. If it is `0x40`, 8-bit boot is performed. A `0x60` byte assumes a 16-bit memory device and performs 8-bit DMA. A `0x20` byte also assumes 16-bit memory but performs 16-bit DMA.
- Boot from serial SPI memory (EEPROM or flash) - 8-, 16-, or 24-bit addressable devices are supported as well as AT45DB041, AT45DB081, AT45DB161, AT45DB321, AT45DB642, and AT45DB1282 DataFlash devices from Atmel. The SPI uses the `PF10`/SPI `SSEL1` output pin to select a single SPI EEPROM/flash device, submits a read command and successive address bytes (`0x00`) until a valid 8-, 16-, or 24-bit, or Atmel addressable device is detected, and begins clocking data into the processor.
- Boot from SPI host device - the Blackfin processor operates in SPI slave mode and is configured to receive the bytes of the `.ldr` file from an SPI host (master) agent. To hold off the host device from transmitting while the boot ROM is busy, the Blackfin processor asserts a `GPIO` pin, called host wait (`HWAIT`), to signal the host device not to send any more bytes until the flag is deasserted. The flag is chosen by the user and this information is transferred to the Blackfin processor via bits `10:5` of the `FLAG` header.

- Boot from UART - using an autobaud handshake sequence, a boot-stream-formatted program is downloaded by the host. The host agent selects a baud rate within the UART's clocking capabilities. When performing the autobaud, the UART expects an "@" (boot stream) character (8 bits data, 1 start bit, 1 stop bit, no parity bit) on the `RXD` pin to determine the bit rate. It then replies with an acknowledgment that is composed of 4 bytes: `0xBF`, the value of `UART_DLL`, the value of `UART_DLH`, and `0x00`. The host can then download the boot stream. When the processor needs to hold off the host, it deasserts `CTS`. Therefore, the host must monitor this signal.
- Boot from serial TWI memory (EEPROM/flash) - the Blackfin processor operates in master mode and selects the TWI slave with the unique ID `0xA0`. It submits successive read commands to the memory device starting at two byte internal address `0x0000` and begins clocking data into the processor. The TWI memory device should comply with Philips I2C Bus Specification version 2.1 and have the capability to auto-increment its internal address counter such that the contents of the memory device can be read sequentially.
- Boot from TWI host - the TWI host agent selects the slave with the unique ID `0x5F`. The processor replies with an acknowledgment, and the host can then download the boot stream. The TWI host agent should comply with Philips I2C Bus Specification version 2.1. An I2C multiplexer can be used to select one processor at a time when booting multiple processors from a single TWI.

To augment the boot modes, a secondary software loader can be added to provide additional booting mechanisms. The secondary loader could provide the capability to boot from flash, variable baud rate, and other sources.

The following loader topics also are discussed in this chapter.

- *ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processor Boot Streams*
- *ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processor Memory Ranges*

Note:

Refer to the processor's data sheet and hardware reference manual for more information on system configuration, peripherals, registers, and operating modes:

- Blackfin processor data sheets can be found at

  http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/data-sheets/resources/index.html.
- Blackfin processor manuals can be found at

  http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/manuals/resources/index.html or downloaded into the CCES IDE via **Help > Install New Software**.

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

The on-chip boot ROM for the 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`).

Note that the on-chip boot ROM of the ADSP-BF534/BF536 and ADSP-BF537 processors executes at the Reset priority level, does not degrade to the lowest priority interrupt.

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 the ADSP-BF533/BF534/BF536/BF537/BF538 and ADSP-BF539 processors; `0xFFA0 8000` for the ADSP-BF531/BF532 processors) for execution. The on-chip boot ROM does this by checking the `NOBOOT` bit (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 the start of L1 memory.
3. The `NOBOOT` bit, if bit 4 of the `SYSCR` register is not set, performs the full boot sequence; see the **ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processors: Booting Sequence** figure).

The boot ROM has the capability to parse address and count information for each bootable block.

The loader utility 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 the booting sequence figure and detailed in the following section. The headers, in turn, are read and parsed by the on-chip boot ROM during booting.

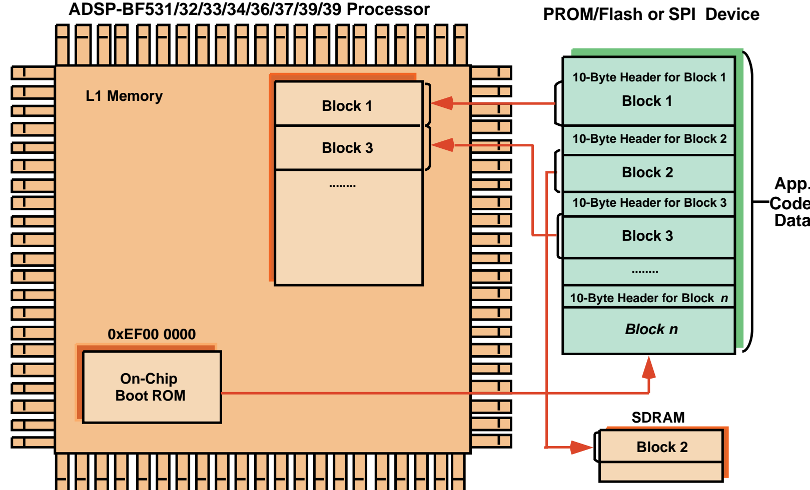


Figure 3. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processors: Booting Sequence

The 10-byte header provides all 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 Block Headers and Flags*
- *ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Initialization Blocks*

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

As the loader utility converts the code from an input `.dxe` file into blocks comprising the output loader file, each block receives a 10-byte header (see the **ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Boot Stream Structure** figure), followed by a block body (if a non-zero block) or no-block body (if a zero block). A description of the header structure can be found in the **ADSP-BF531/BF532/BF533 Block Header Structure** table.

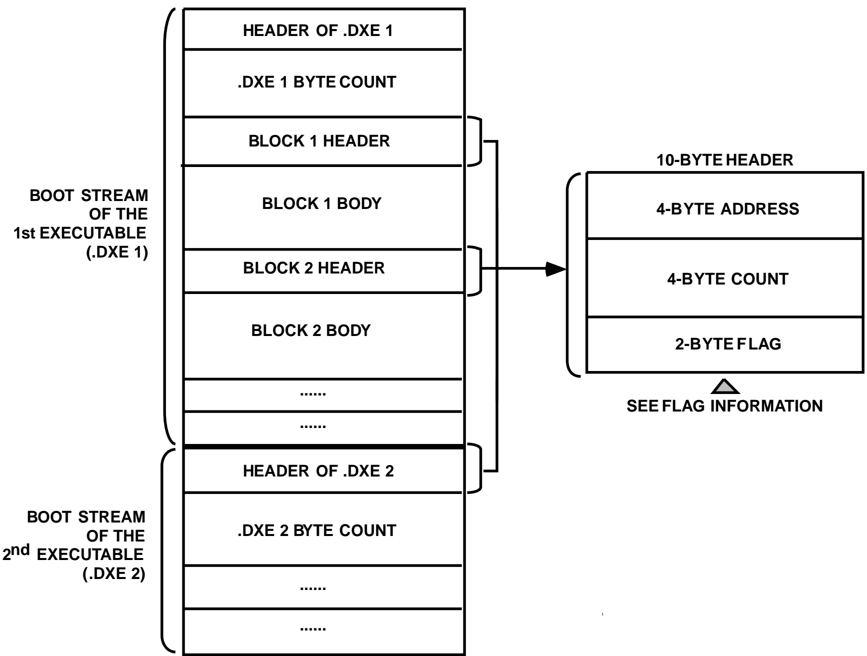


Figure 4. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Boot Stream Structure

Table 10. 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

Refer to the **Flag Bit Assignments for 2-Byte Block Flag Word** figure and **Flag Structure** table for the flag's bit descriptions.

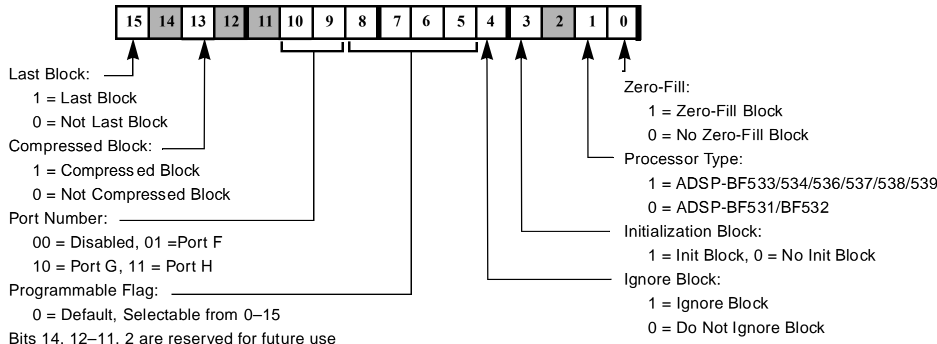


Figure 5. Flag Bit Assignments for 2-Byte Block Flag Word

Table 11. Flag Structure

Bit Field	Description
Zero-fill block	Indicates that the block is for a buffer filled with zeros. The body of a zero block is not included within the loader file. When the loader utility parses through the `.dxe` file and encounters a large buffer with zeros, it creates a zero-fill block to reduce the `.ldr` file size and boot time. If this bit is set, there is no block body in the block.
Processor type	Indicates the processor, either the ADSP-BF531/BF532/BF538 or the ADSP-BF533/BF534/BF536/BF537/BF539. Once booting is complete, the on-chip boot ROM jumps to `0xFFA0 0000` on the ADSP-BF533/BF536/BF537/BF538/BF539 processor and to `0xFFA0 8000` on the ADSP-BF531/BF532/ processors.
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. The figures in *ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Initialization Blocks* illustrate the process. Initialization code can be included within the `.ldr` file by using the `-init` switch (see `-init filename.dxe`).
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.

Bit Field	Description
	Note: This flag is equivalent to the `FIRST` flag in boot streams on the ADSP-BF51x/BF52x/BF54x processors. Because the `IGNORE` flag is used for other purposes on the ADSP-BF51x/BF52x/BF54x processors, the `FIRST` flag is invented to indicate the first header.
Compressed block	Indicates that the block contains compressed data. The compressed block can include a number of blocks compressed together to form a single compressed block.
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-BF534/BF536/BF537 processor can have a special last block if the boot mode is two-wire interface (TWI). The loader utility saves all the data from `0xFF90 3F00` to `0xFF90 3FFF` and makes the last block with the data. The loader utility, however, creates a regular last block if no data is in that memory range. The space of `0xFF90 3F00` to `0xFF90 3FFF` is saved for the boot ROM to use as a data buffer during a boot process.

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Initialization Blocks

The `-init` *`filename`* option directs the loader utility to produce the initialization blocks from the initialization section's code in 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 booted into the memory (see the **ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Initialization Block Execution** figure).

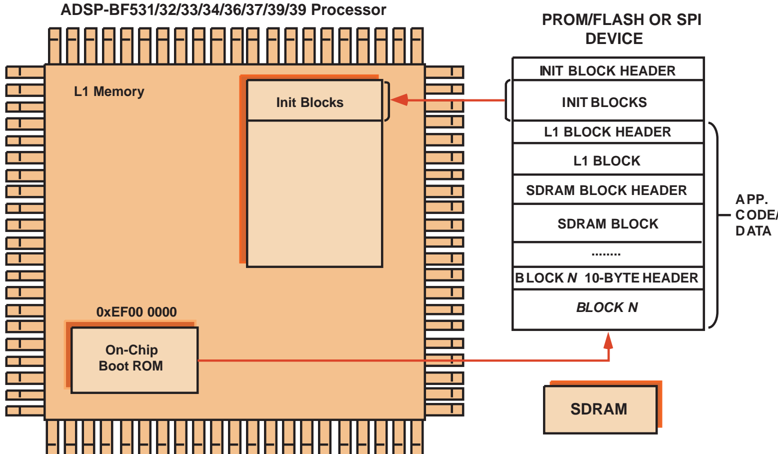
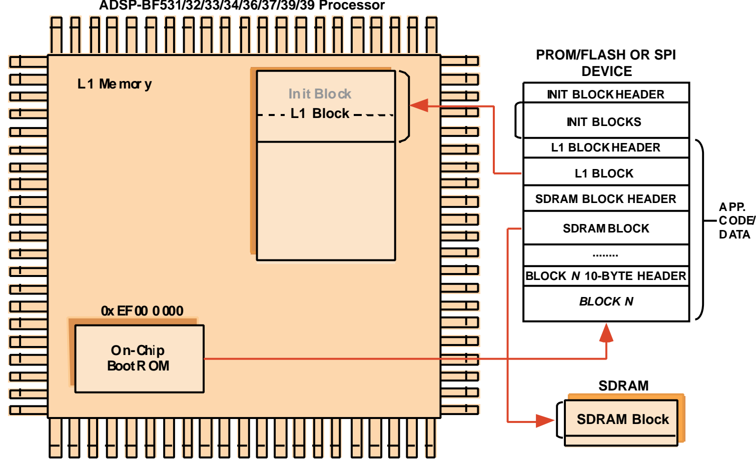


Figure 6. ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Initialization Block Execution

Following execution of the initialization blocks, the boot process continues with the rest of data blocks until it encounters a final block (see the **ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/ BF539 Processors: Booting Application Code** figure). The initialization code example follows in **Initialization Block Code Example**.

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



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.
    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 = LO(EBIU_SDRRC);
     /* SDRAM Refresh Rate Control Register */
     P0.H = HI(EBIU_SDRRC);
     R0 = 0x074A(Z);
     W[P0] = R0;
     SSYNC;

```

```
    P0.L = LO(EBIU_SDBCTL);
    /* SDRAM Memory Bank Control Register */
    P0.H = HI(EBIU_SDBCTL);
    R0 = 0x0001(Z);
    W[P0] = R0;
    SSYNC;

    P0.L = LO(EBIU_SDGCTL);
    /* SDRAM Memory Global Control Register */
    P0.H = HI(EBIU_SDGCTL);
    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 No-Boot Mode

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

To create a proper `.ldr` file that can be burned into either a parallel flash or EPROM device, you must modify the standard LDF file in order for the reset vector to be located accordingly. The **Section Assignment (LDF File)**

Example and **ROM Segment Definitions (LDF File) Example** code fragments illustrate the required modifications in case of an ADSP-BF533 processor.

Section Assignment (LDF File) Example

```
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) }
```

ROM Segment Definitions (LDF File) Example

```
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 **Section Handling (Source File) Example**.

Section Handling (Source File) Example

```
.SECTION rom_code;
_reset_vector: l0 = 0;
               l = 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 */
```

ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor Memory Ranges

The on-chip boot ROM on the 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-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`)

Note:

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

Note:

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

ADSP-BF561 Processor Booting

The booting sequence for the ADSP-BF561 dual-core processors is similar to the ADSP-BF531/BF532/BF533 processor boot sequence described in *ADSP-BF531/BF532/BF533/BF534/BF536/BF537/BF538/BF539 Processor On-Chip Boot ROM*. 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`.

The ADSP-BF561 ROM details can be found in *ADSP-BF561 Processor On-Chip Boot ROM*.

The **ADSP-BF561 Processor Boot Mode Selections** table summarizes the boot modes and execution start addresses for the ADSP-BF561 processors.

Table 12. ADSP-BF561 Processor Boot Mode Selections

Boot Source	BMODE[1:0]
16-bit external memory (bypass boot ROM)	00
8- or 16-bit flash	01
SPI host	10
SPI serial EEPROM (16-bit address range)	11

- Execute from 16-bit external memory - execution starts from address `0x2000 0000` with 16-bit packing. The boot ROM is bypassed in this mode. All configuration settings are set for the slowest device possible (3-cycle hold time, 15-cycle R/W access times, 4-cycle setup).
- Boot from 8-bit/16-bit external flash memory - the 8-bit/16-bit flash boot routine located in boot ROM memory space is set up using asynchronous memory bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).
- Boot from SPI host - the ADSP-BF561 processor is configured as an SPI slave device and a host is used to boot the processor. The host drives the SPI clock and is therefore responsible for the timing. The baud rate should be equal to or less than one fourth of the ADSP-BF561 system clock (`SCLK`).
- Boot from SPI serial EEPROM (16-bit addressable) - the SPI uses the `PF2` output pin to select a single SPI EPROM device, submits a read command at address `0x0000`, and begins clocking data into the beginning of L1 instruction memory. A 16-bit/24-bit addressable SPI-compatible EPROM must be used.

The following loader topics also are discussed in this chapter.

- *ADSP-BF561 Processor Boot Streams*
- *ADSP-BF561 Processor Initialization Blocks*
- *ADSP-BF561 Dual-Core Application Management*
- *ADSP-BF561 Processor Memory Ranges*

Note:

Refer to the *ADSP-BF561 Embedded Symmetric Multiprocessor* data sheet and the *ADSP-BF561 Blackfin Processor Hardware Reference* manual for information about the processor's operating modes and states, including background information on system reset and booting.

ADSP-BF561 Processor On-Chip Boot ROM

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`.

Similar to 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 whether 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 (`.dxe`) file. The debugger loads the `.dxe` file 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 utility converts the `.dxe` file into a boot stream (`.ldr`) file 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 memory, 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.

Note:

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 (see *ADSP-BF561 Dual-Core Application Management* and *ADSP-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management*).

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 the **ADSP-BF561 Global Header Structure** table. The global header also contains a signature in the upper 4 bits that prevents the boot ROM from reading in a boot stream from a blank device.

Table 13. 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`

Bit Field	Description
`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` file 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 instructs 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 Block Headers and Flags).*). 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 utility 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 Block Headers and Flags*.

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 the **ADSP-BF561 Processor Boot Stream Structure** table. When learning about the ADSP-BF561 boot stream structure, keep in mind that the count byte for each `.dxe` is, itself, a block encapsulated by a block header.

Table 14. ADSP-BF561 Processor Boot Stream Structure

Bit Field		Description	
	`0-7`	LSB of the global header	32-Bit Global Header
	`8-15`	8-15 of the global header	
	`16-23`	16-23 of the global header	

Bit Field		Description	
	24-31	MSB of the global header	
	32-39	LSB of the address field of 1st `.dxe` count block (no care)	10-Byte .dxe1 Header
	40-47	8-15 of the address field of 1st `.dxe` count block (no care)	
	48-55	16-23 of the address field of 1st `.dxe` count block (no care)	
	56-63	MSB of the address field of 1st `.dxe` count block (no care)	
	64-71	LSB (4) of the byte count field of 1st `.dxe` count block	
	72-79	8-15 (0) of the byte count field of 1st `.dxe` count block	
	80-87	16-23 (0) of the byte count field of 1st `.dxe` count block	
	88-95	MSB (0) of the byte count field of 1st `.dxe` count block	
	96-103	LSB of the flag word of 1st `.dxe` count block - ignore bit set	
	104-111	MSB of the flag word of 1st `.dxe` count block	
	112-119	LSB of the first 1st `.dxe` byte count	32-Bit Block 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	

Bit Field		Description	
1-0-Byte Block Header	144-151	LSB of the address field of the 1st data block in 1st .dxe	.dxe1 Block Data
	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	
Block Data	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	

Bit Field		Description	
	...	And so on ...	
10-Byte Block Header	...	LSB of the address field of the nth data block in 1st `.dxe`	
	...	8-15 of the address field of the nth data block in 1st `.dxe`	
	...	16-23 of the address field of the nth data block in 1st `.dxe`	
	...	MSB of the address field of the nth data block in 1st `.dxe`	
	...	LSB of the byte count of the nth data block in 1st `.dxe`	
	...	8-15 of the byte count of the nth data block in 1st `.dxe`	
	...	16-23 of the byte count of the nth data block in 1st `.dxe`	
	...	MSB of the byte count of the nth data block in 1st `.dxe`	
	...	LSB of the flag word of the nth block in 1st `.dxe`	
	...	MSB of the flag word of the nth block in 1st `.dxe`	
Block data	...	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)	10-Byte .dxe2 Header

Bit Field	Description
...	8-15 of the address field of 2nd .dxe count block (no care)
...	And so on ...

ADSP-BF561 Processor Initialization Blocks

The initialization block or a second-stage loader utility must be used to initialize the SDRAM memory of the ADSP-BF561 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 Block Headers and Flags*.

Both the initialization block and second-stage loader utility can be used to force the boot ROM to load a specific .dxe file 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 the 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 file 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 Multi-Application (Multi-DXE) Management*.

ADSP-BF561 Dual-Core Application Management

A typical ADSP-BF561 dual-core application is separated into two executable files: one executable file for each core. The default linker description (.ldf) file for the ADSP-BF561 processor creates two separate executable files (p0.dxe and p1.dxe) and some shared memory files (sml2.sm and sml3.sm). By modifying the LDF, it is possible to create a dual-core application that combines both cores into a single .dxe file. This is not recommended unless the application is a simple assembly language program which does not link any C 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 utility combines the contents of the shared memory files (sml2.sm, sml3.sm) only into the boot stream generated from the .dxe file for core A (p0.dxe).

By default, the boot ROM loads only 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. (Or, when the -noSecondStageKernel switch is called, the loader utility combines the two .dxe files into one.) If the he second-stage boot loader is used, it 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 if a loader file includes the codes and data from a number of `.dxe` files.

Attention:

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-BF561 Processor Memory Ranges

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

- Core A
  - L1 instruction SRAM (`0xFFA0 0000` - `0xFFA0 3FFF`)
  - L1 instruction cache/SRAM (`0xFFA1 0000` - `0xFFA1 3FFF`)
  - L1 data bank A SRAM (`0xFF80 0000` - `0xFF80 3FFF`)
  - L1 data bank A cache/SRAM (`0xFF80 4000` - `0xFF80 7FFF`)
  - L1 data bank B SRAM (`0xFF90 0000` - `0xFF90 3FFF`)
  - L1 data bank B cache/SRAM (`0xFF90 4000` - `0xFF90 7FFF`)
- Core B
  - L1 instruction SRAM (`0xFF60 0000` - `0xFF6 03FFF`)
  - L1 instruction cache/SRAM (`0xFF61 0000` - `0xFF61 3FFF`)
  - L1 data bank A SRAM (`0xFF40 0000` - `0xFF40 3FFF`)
  - L1 data bank A cache/SRAM (`0xFF40 4000` - `0xFF40 7FFF`)
  - L1 data bank B SRAM (`0xFF50 0000` - `0xFF50 3FFF`)
  - L1 data bank B cache/SRAM (`0xFF50 4000` - `0xFF50 7FFF`)
- 128K of shared L2 memory (`FEB0 0000` - `FEB1 FFFF`)
- Four banks of configurable synchronous DRAM (`0x0000 0000` - (up to) `0x1FFF FFFF`)

Attention:

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-BF53x and ADSP-BF561 Multi-Application (Multi-DXE) Management

This section describes how to generate and boot more than one `.dxe` file for the ADSP-BF531/BF532/BF533/BF534/BF536/BF537/ BF538/BF539 and ADSP-BF561 processors. For further information about the ADSP-BF561 processors, refer to *ADSP-BF561 Dual-Core Application Management*.

The ADSP-BF531/BF532/BF533/BF534/ BF536/BF537/BF538/BF539 and ADSP-BF561 loader file structure and the silicon revision of 0.1 and higher allow generation and booting of multiple `.dxe` files into a single processor from external memory. As illustrated in the **ADSP-BF531/BF32/BF33/BF534/ BF536/BF537/BF538/ BF539/ BF561 Processors: Multi-Application Booting Streams** figure, 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` file 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 Block Headers and Flags*.

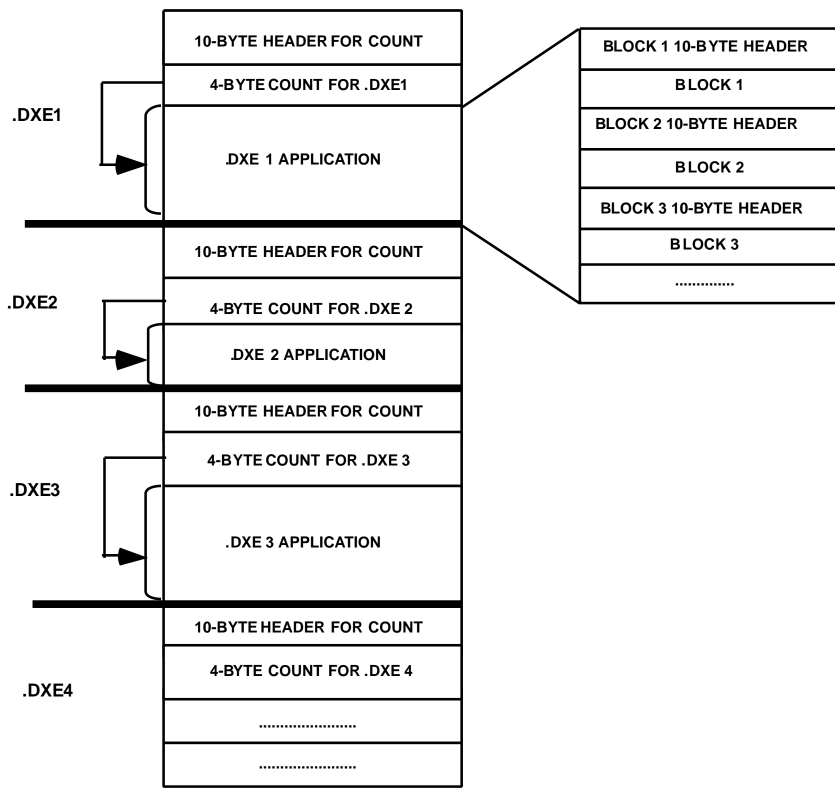


Figure 7. ADSP-BF531/BF32/BF33/BF534/ BF536/BF537/BF538/ BF539/BF561 Processors: Multi-Application Booting Streams

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

- Use the second-stage loader switch, `-l` *`userkernel.dxe`*. The option allows you to use your own second-stage loader.

  After the second-stage loader is booted into internal memory via the on-chip boot ROM, the loader 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.
- Use the initialization block switch, `-init` *`filename.dxe`*, where *`filename.dxe`* 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. On the ADSP-BF531 and ADSP-BF561 processors, the initialization code is an assembly written subroutine.

A sample initialization code is included in Listing 3-5. 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.

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;
```

ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support

The loader utility for the ADSP-BF531/BF532/BF533/BF534/BF536/BF537 processors offers a loader file (boot stream) compression mechanism known as zLib. The zLib compression is supported by a third party dynamic link library, `zLib1.dll`. Additional information about the library can be obtained from the *http://www.zlib.net* Web site.

The `zLib1.dll` dynamic link library is included with CrossCore Embedded Studio. The library functions perform the boot stream compression and decompression procedures when the appropriate options are selected for the loader utility. The initialization executable files with built-in decompression mechanism must perform the decompression on a compressed boot stream in a boot process. The default initialization executable files with decompression functions are included in CrossCore Embedded Studio.

The loader `-compression` switch directs the loader utility to perform the boot stream compression from the command line. The IDE also includes a dedicated loader properties page to manage the compression. Refer to the online help for details.

The loader utility takes two steps to compress a boot stream. First, the utility generates the boot stream in the conventional way (builds data blocks), then applies the compression to the boot stream. The decompression initialization is the reversed process: the loader utility decompresses the compressed stream first, then loads code and data into memory segments in the conventional way.

The loader utility compresses the boot stream on the `.dxe`-by-`.dxe` basis. For each input `.dxe` file, the utility compresses the code and data together, including all code and data from any associated overlay (`.ovl`) and shared memory (`.sm`) files.

Compressed Streams

The **Loader File with Compressed Streams** figure illustrates the basic structure of a loader file with compressed streams.

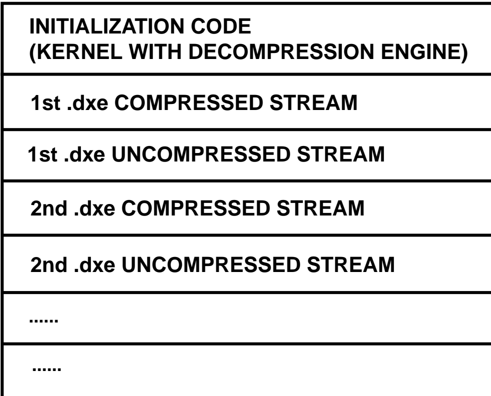


Figure 8. Loader File with Compressed Streams

The initialization code is on the top of the loader file. The initialization code is loaded into the processor first and is executed first when a boot process starts. Once the initialization code is executed, the rest of the stream is brought into the processor. The initialization code calls the decompression routine to perform the decompression operation on the stream, and then loads the decompressed stream into the processor's memory in the same manner a conventional boot kernel does when it encounters a compressed stream. Finally, the loader utility loads the uncompressed boot stream in the conventional way.

The **Compressed Block** figure illustrates the structure of a compressed block.

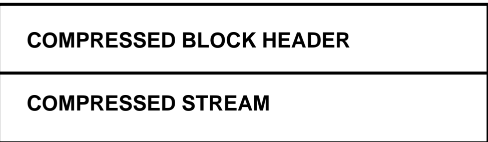


Figure 9. Compressed Block

Compressed Block Headers

A compressed stream always has a header, followed by the payload compressed stream. The **Compressed Block Header** figure shows the structure of a compressed block header.

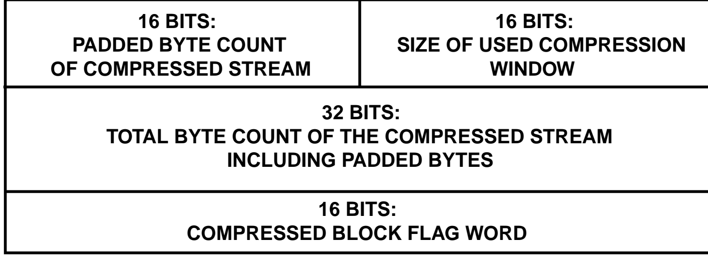


Figure 10. Compressed Block Header

The first 16 bits of the compressed block header hold the padded byte count of the compressed stream. The loader utility always pads the byte count if the resulting compressed stream from the loader compression engine is an odd number. The loader utility rounds up the byte count of the compressed stream to be a next higher even number. This 16-bit value is either `0x0000` or `0x0001`.

The second 16 bits of the compressed block header hold the size of the compression window, used by the loader compression engine. The value range is 8-15 bits, with the default value of 9 bits. The compression window size specifies to the compression engine a number of bytes taken from the window during the compression. The window size is the 2's exponential value.

As mentioned before, the compression/decompression mechanism for Blackfin processors utilizes the open-source lossless data-compression library zLib1. The zLib1 deflate algorithm, in turn, is a combination of a variation of Huffman coding and LZ77 compression algorithms.

LZ77 compression works by finding sequences of data that are repeated within a sliding window. As expected, with a larger sliding window, the compression algorithm is able to find more repeating sequences of data, resulting in higher compression ratios. However, technical limitations of the zLib1 decompression algorithm dictate that the window size of the decompressor must be the same as the window size of the compressor. For a more detailed technical explanation of the compression/decompression implementation on a Blackfin processor, refer to the `readme.txt` file in the `<install_path>/Blackfin/ldr/zlib/src` directory.

Note:

It is not recommended to use memory ranges used by the zlib kernel. The memory ranges used by the kernel, such as heap and static data, are defined in the LDF file, for example in `<install_- path>/Blackfin/ldr/zlib/src/blkfin_zlib_init.ldf`.

In the Blackfin implementation, the decompressor is part of the decompression initialization files (see *Decompression Initialization Files*). These files are built with a default decompressor window size of 9 bits (512 bytes). Thus, if you choose a non-default window size for the compressor from the **Compression window size (-compressWS)** drop-down list on the loader's **Compression** properties page, then the decompressor must be re-built with the new window size. Refer to the CCES online help for information about the **Compression** properties page. For details on re-building of the decompressor init project, refer to the `readme.txt` file located in the `<install_path>/Blackfin/ldr/zlib/src` directory.

While it is true that a larger compression window size results in better compression ratios, note that there are counter factors that decrease the overall effective compression ratios with increasing window sizes for Blackfin's implementation of zlib. This is because of the limited memory resources on an embedded target, such as a Blackfin processor. For more information, refer to the `readme.txt` file in the `<install_path>/Blackfin/ldr/zlib/src` directory.

The last 16 bits of the compressed header is the flag word. The valid compression flag assignments are shown in the **Flag Word of Compressed Block Header** figure.

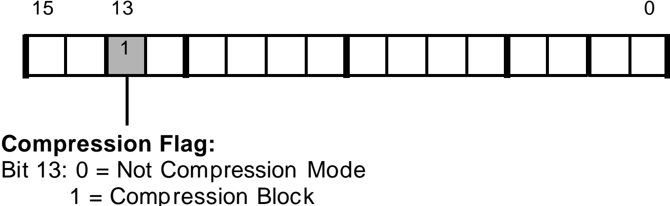


Figure 11. Flag Word of Compressed Block Header

Uncompressed Streams

Following the compressed streams (illustrated in the **Loader File with Compressed Streams** figure in *Compressed Streams*), the loader file includes the uncompressed streams. The uncompressed streams include application codes, conflicted with the code in the initialization blocks in the processor's memory spaces, and a final block. The uncompressed stream includes only a final block if there is no conflicted code. The final block can have a zero byte count. The final block indicates the end of the application to the initialization code.

Booting Compressed Streams

The **ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Compressed Stream: Booting Sequence** figure shows the booting sequence of a loader file with compressed streams. The loader file is prestored in the flash memory.

1. The boot ROM is pointing to the start of the flash memory. The boot ROM reads the initialization code header and boots the initialization code.
2. The boot ROM jumps to and starts executing the initialization code.
3. (A) The initialization code scans the header for any compressed streams (see the compression flag structure in the **Flag Word of Compressed Block Header** figure in *Compressed Block Headers*). The code decompresses the streams to the decompression window (in parts) and runs the initialization kernel on the decompressed data.

   (B) The initialization kernel boots the data into various memories just as the boot ROM kernel does.
4. The initialization code sets the boot ROM to boot the uncompressed blocks and the final block (`FINAL` flag is set in the block header's flag word). The boot ROM boots the final payload, overwriting any areas used by the initialization code. Because the final flag is set in the header, the boot ROM jumps to `EVT1` (`0xFFA0 0000` for the ADSP-BF533/BF534/BF536/BF537/BF538 and ADSP-BF539 processors; `0xFFA0 8000` for the ADSP-BF531/BF532 processors) to start application code execution.

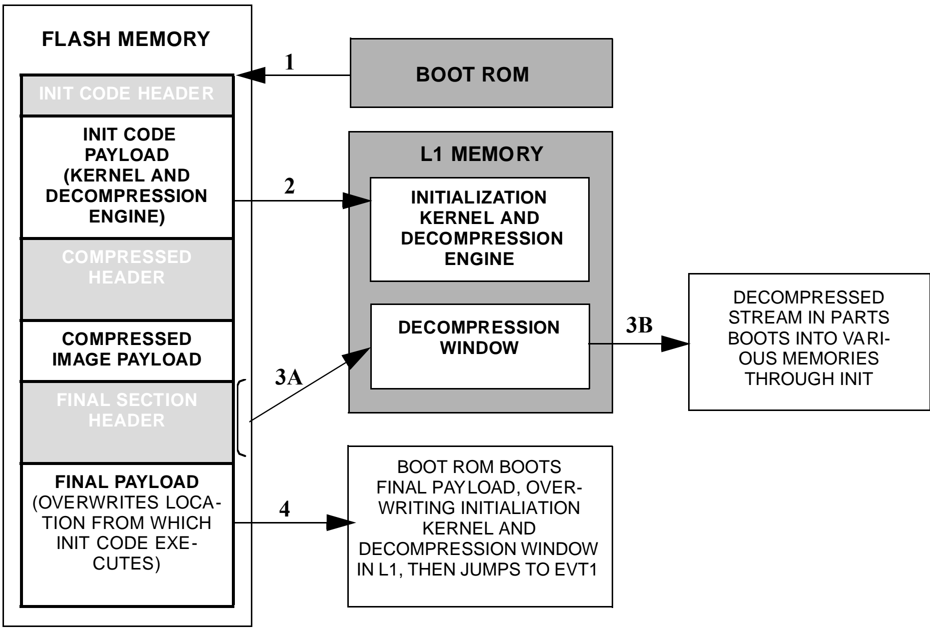


Figure 12. ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Compressed Stream: Booting Sequence

Decompression Initialization Files

As stated before, a decompression initialization `.dxe` file must be used when building a loader file with compressed streams. The file has a built-in decompression engine to decompress the compressed streams from the loader file.

The decompression initialization file can be specified from the loader properties page or from the loader command line via the `-init` *`filename.dxe`* switch. CrossCore Embedded Studio includes the default decompression initialization files, which the loader utility uses if no other initialization file is specified. The default decompression initialization file is stored in the `<install_path>/Blackfin/ldr/zlib` directory. The default file is built for the compression window size of 9 bits.

To use a different compression window size, build your own decompression initialization file. For details, refer to the `readme.txt` file located in the `<install_path>/Blackfin/ldr/zlib/src` directory. The size can be changed through the loader properties page or the `-compressWS #` command-line switch. The valid range for the window size is [8-15] bits.

ADSP-BF53x/BF561 Processor Loader Guide

Loader utility operations depend on the loader properties, which control how the utility processes executable files. You select features, such as boot modes, boot kernels, and output file formats via the properties. The properties are specified on the loader utility's command line or the **Tool Settings** dialog box in the IDE (**CrossCore Blackfin Loader** pages). The default loader settings for a selected processor are preset in the IDE.

Note:

The IDE's **Tool Settings** correspond to switches displayed on the command line.

These sections describe how to produce a bootable or non-bootable loader file:

- *Loader Command Line for ADSP-BF53x/BF561 Processors*
- *CCES Loader and Splitter Interface for ADSP-BF53x/BF561 Processors*

Loader Command Line for ADSP-BF53x/BF561 Processors

The loader utility uses the following command-line syntax for the ADSP-BF53x/BF561 Blackfin processors.

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 (`.dxe`) file 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 utility to process the files. Enclose long file names within straight quotes, `"long file name"`.
- `-proc processor` - Part number of the processor (for example, `-proc 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 utility.

Note:

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 utility processes the input files in the order presented on the command line.

Loader Command-Line Switches for ADSP-BF533/BF561 Processors

A summary of the loader command-line switches for the ADSP-BF53x/BF561 Blackfin processors appears in the following table.

Table 15. ADSP-BF53x/BF561 Loader Command-Line Switches

Switch	Description
`-b {prom|flash|spi|spislave|uart|twi|fifo}`	The `-b` switch specifies the boot mode and directs the loader utility to prepare a boot-loadable file for the specified boot mode.

Switch	Description
	If -b does not appear on the command line, the default is -b flash. Other valid boot modes include: • SPI master (-b spi) for all processors described in this chapter. • SPI slave (-b spislave) for the ADSP-BF531/2/3/4/6/7/9 and ADSP-BF561 processors. • UART (-b uart) for the ADSP-BF534/6/7 processors. • TWI (-b twi) for the ADSP-BF534/6/7 processors. • FIFO (-b fifo) for the ADSP-534/6/7 processors.
-compression	The -compression switch directs the loader utility to compress the boot stream; see *ADSP-BF531/BF532/BF533/BF534/BF536/BF537 Processor Compression Support*. Either a default or user initialization .dxe file with decompression code must be provided for -compression. This switch is for flash/PROM boot modes only and does not apply to the ADSP-BF538, ADSP-BF539, or ADSP-BF561 processors.
-compressWS #	The -compressWS # switch specifies a compression window size in bytes. The number is a 2's exponential value to be used by the compression engine. The valid values are [8,15] bits, with the default of 9 bits. This switch is for flash/PROM boot modes only and does not apply to the ADSP-BF538, ADSP-BF539, or ADSP-BF561 processors.
-dmawidth {8|16}	The -dmawidth {8|16} switch specifies a DMA width (in bits) to the loader utility. For FIFO boot mode, 16 is the only DMA width. For other boot modes, all DMA widths are valid with the default of 8. The switch does not apply to the ADSP-BF561 processors.
-enc *dll_filename*	The -enc *dll_filename* switch encrypts the data stream from the application input .dxe files with the encryption algorithms in the dynamic library file *dll_filename*. The *dll_filename* is required. Two functions with the following APIs are required in the encryption DLL:

Switch	Description

	For setting the encryption initial value:
	`int EncryptInit(unsigned int FixedData);`
	For getting encrypted data:
	`int EncryptBlock(unsigned int * buffer, unsigned int BlkSize, char * message);`
	The loader calls the encryption routines as it is creating the `ldr` output file. The loader sets reserved bit 2 in the block header to indicate the payload is encrypted.

`-f {hex|ascii|binary|include}`	The `-f {hex|ASCII|binary|include}` switch 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 mode format is hex for flash/PROM and ASCII for SPI, SPI slave, UART, and TWI.
`-ghc #`	The `-ghc #` switch specifies a 4-bit value (global header cookie) for bits 31-28 of the global header (see the **ADSP-BF561 Global Header Structure** figure in *ADSP-BF561 Processor Boot Streams*). **Note:** The switch applies to the ADSP-BF561 processors only.
`-h` or `-help`	The `-h[elp]` switch 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-BF533 -h` to obtain help for the ADSP-BF533 processor.
`-init` *`filename.dxe`*	The `-init` *`filename.dxe`* switch directs the loader utility to include the initialization code from the named file. The loader utility places the code from the initialization sections 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 an RTS back to the kernel.

Switch	Description
`-kb {prom|flash|spi|spislave|uart|twi|fifo}`	The `-kb` switch specifies the boot mode for the boot kernel output file if you generate two output files from the loader utility: one for the boot kernel and another for user application code. The `-kb` switch must be used in conjunction with the `-o2` switch. If the `-kb` switch is absent from the command line, the loader utility generates the file for the boot kernel in the same boot mode as used to output the user application program. Valid boot modes include: • PROM/FLASH (`-kb prom` or `-kb flash`) - the default boot mode for all processors described in this chapter. • SPI master (`-kb spi`) for all processors described in this chapter. • SPI slave (`-kb spislave`) for the ADSP-BF531/2/3/4/6/7/9 and ADSP-BF561 processors. • UART (`-kb uart`) for the ADSP-BF534/6/7 processors. • TWI (`-kb twi`) for the ADSP-BF534/6/7 processors. • FIFO (`-kb fifo`) for the ADSP-534/6/7 processors.
`-kf {hex|ascii|binary|include}`	The `-kf {hex|asci|binary|include}` switch specifies the output file format (hex, ASCII, binary, or include) for the boot kernel if you output two files from the loader utility: one for the boot kernel and one for user application code. The `-kf` switch must be used in conjunction with the`-o2` switch. If the `-kf` switch is absent from the command line, the loader utility generates the file for the boot kernel in the same format as for the user application program.
`-kenc` *`dll_filename`*	The `-kenc` *`dll_filename`* switch specifies the user encryption dynamic library file for the encryption of the data stream from the kernel file. The *`dll_filename`* is required. Two functions with the following APIs are required in the encryption DLL: For setting the encryption initial value: `int EncryptInit (unsigned int FixedData);` For getting encrypted data: `int EncryptBlock (unsigned int * buffer, unsigned int BlkSize, char * message);`

Switch	Description
	The loader calls the encryption routines as it is creating the `.knl` output file. The loader sets reserved bit 2 in the block header to indicate the payload is encrypted.
`-kp #`	The `-kp #` switch specifies a hex flash/PROM output start address for the kernel code. A valid value is between `0x0` and `0xFFFFFFFF`. The specified value is ignored when no kernel or/and initialization code is included in the loader file.
`-kwidth {8|16|32}`	The `-kwidth {8|16|32}` switch specifies the width of the boot kernel output file when there are two output files: one for the boot kernel and one for user application code. Valid values are: • `8` or `16` for PROM or flash boot kernel • `16` for FIFO boot kernel • `8` for SPI and other boot kernels If this switch is absent from the command line, the default file width is: • the `-width` parameter for flash/PROM boot mode • `16` for FIFO boot mode • `8` when booting from SPI and other boot modes The `-kwidth` switch must be used in conjunction with the `-o2` switch.
`-M`	The `-M` switch generates make dependencies only, no output file is generated.
`-maskaddr #`	The `-maskaddr #` switch masks all EPROM address bits above or equal to #. For example, `-maskaddr 29` (default) masks all the bits above and including `A29` (`AND`ed 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]. The `-maskaddr #` switch requires `-romsplitter` and affects the ROM section address only.
`-MaxBlockSize #`	The `-MaxBlockSize #` switch specifies the maximum block byte count, which must be a multiple of 16.

Switch	Description
-MaxZeroFillBlockSize #	The -MaxZeroFillBlockSize # switch specifies the maximum block byte count for zero-filled blocks. The valid values are from 0x0 to 0xFFFFFFF0, and the default value matches -MaxBlockSize #.
-MM	The -MM switch generates make dependencies while producing the output files.
-Mo *filename*	The -Mo *filename* switch writes make dependencies to the named file. Use the -Mo switch with either -M or -MM. If -Mo is not present, the default is a <stdout> display.
-Mt *filename*	The -Mt *filename* switch specifies the make dependencies target output file. Use the -Mt switch with either -M or -MM. If -Mt is not present, the default is the name of the input file with an .ldr extension.
-noFinalBlock	The -noFinalBlock switch directs the loader utility not to make a special final block for TWI boot. The switch applies to the ADSP-BF537 processors only.
-noFinalTag	The -noFinalTag switch directs the loader utility not to set the final block tag for the first .dxe file. As a result, the boot process continues with code from the second .dxe file, following the first file. The switch applies to the ADSP-BF56x processors only.
-noInitCode	The -noInitCode switch directs the loader utility not to expect an initialization input file even though an external memory section is present in the input .dxe file. The switch applies to the ADSP-BF531/BF532/BF533, ADSP-BF534/BF536/BF537/BF538/BF539 processors only.
-noSecondStageKernel	The -noSecondStageKernel switch directs the loader utility not to include a default second-stage kernel into the loader stream. The switch applies to the ADSP-BF56x processors only.
-o *filename*	The -o *filename* switch directs the loader utility to use the specified file as the name of the loader utility's output file. If the

Switch	Description
	`filename` is absent, the default name is the root name of the input file with an `.ldr` extension.
`-o2`	The `-o2` switch 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, respectively. Combine `-o2` with `-l` *`filename`* and/or `-init` *`filename`* on the ADSP-BF531/BF532/BF533, ADSP-BF534/BF536/BF537/BF538/BF539, ADSP-BF561 processors.
`-p #`	The `-p #` switch specifies a hex flash/PROM output start address for the application code. A valid value is between `0x0` and `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 {#|PF#|PG#|PH# }`	The `-pflag { #|PF#|PG#|PH# }` switch specifies a 4-bit hex value for a strobe (programmable flag) or for one of the ports: `F`, `G`, or `H`. There is no default value. The value is dynamic and varies with processor, silicon revision, boot mode, and width. The loader generates warnings for illegal combinations. The **-pFlag Values for ADSP-BF531/BF532/BF533 Processors**, **-pFlag Values for ADSP-BF534/BF536/BF537**, and **-pFlag Values for ADSP-BF538/BF539 Processors** tables show the valid values for the switch. The switch applies to the ADSP-BF531x and ADSP-BF561 processors only.
`-proc` *`processor`*	The `-proc` *`processor`* switch specifies the target processor. The *`processor`* can be one of the following: `ADSP-BF531`, `ADSP-BF532`, `ADSP-BF533`, `ADSP-BF534`, `ADSP-BF536`, `ADSP-BF537`, `ADSP-BF538`, `ADSP-BF539`, `ADSP-BF561`.
`-romsplitter`	The `-romsplitter` switch creates a non-bootable image only. This switch overwrites the `-b` switch and any other switch bounded by the boot mode.

Switch	Description
	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`	The `-ShowEncryptionMessage` switch displays a message returned from the encryption function.
`-si-revision [none|any|`*x.x*`]`	Sets revision for the build, with *x.x* being the revision number for the processor hardware. If `-si-revision` is not used, the target is a default revision from the supported revisions.
`-v`	The `-v` switch directs the loader utility to output verbose loader messages and status information as the loader processes files.
`-width {8|16|32}`	The `-width {8|16|32}` switch specifies the loader output file's width in bits. Valid values are `8` and `16`, depending on the boot mode. The default value is `16` for FIFO boot mode and `8` for all other boot modes. • For flash/PROM booting, the size of the output file depends on the `-width` switch. • For FIFO booting, the only available width is `16`. • 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.
`-ZeroPadForced`	The `-ZeroPadForced` switch forces the loader utility to pad each data byte with a zero byte for 16-bit output. Use this switch only if your system requires zero padding in a loader file. Use this switch with caution: arbitrating pad data with zeros can cause the loader file to fail. The loader utility performs default zero padding automatically in general. The switch applies to the ADSP-BF531/BF532/BF533/BF534, ADSP-BF536/BF537/BF538/BF539 processors only.

Table 16. -pFlag Values for ADSP-BF531/BF532/BF533 Processors. (The ADSP-BF531/BF532/BF533 processors always have the RESVECT bit (bit 2 in the block header fla word) cleared.)

Silicon Revision	0.6	
Width	8	16
Flash boot mode	NONE	NONE
SPI boot mode	NONE	
SPI slave boot mode	1-15 PF1-15	

Table 17. -pFlag Values for ADSP-BF534/BF536/BF537. (The ADSP-BF534/BF536/BF537 processors always have the RESVECT bit (bit 2 in the block header fla word) set.)

Silicon Revision	0.3	
Width	8	16
Flash boot mode	NONE PF0-15 PG0-15 PH0-15	NONE PF0-15 PG0-15 PH0-15
SPI boot mode	NONE PF0-9 PF15 PG0-15 PH0-15	
SPI slave boot mode	NONE PF0-10 PF15 PG0-15	

Silicon Revision	0.3	
	PH0-15	
TWI boot mode	NONE PF0-15 PG0-15 PH0-15	
TWI slave boot mode	NONE PF0-15 PG0-15 PH0-15	
UART boot mode	NONE PF2-15 PG0-15 PH0-15	
FIFO boot mode		NONE PF0 PF2-15 PG0-15 PH0-15

Table 18. -pFlag Values for ADSP-BF538/BF539 Processors

Silicon Revision	All	
Width	8	16
Flash boot mode	NONE	NONE
SPI boot mode	NONE	

Silicon Revision	All	
SPI slave boot mode	`1-15` `PF1-15`	

Note:

The ADSP-BF538/BF539 processors always have the `RESVECT` bit (bit 2 in the block header flag word) set.

CCES Loader and Splitter Interface for ADSP-BF53x/BF561 Processors

Once a project is created in the CrossCore Embedded Studio IDE, you can change the project's output (artifact) type.

The IDE invokes the `elfloader.exe` utility to build the output loader file. To modify the default loader properties, use the project's **Tool Settings** dialog box. The controls on the pages correspond to the loader command-line switches and parameters (see *Loader Command-Line Switches for ADSP-BF533/BF561 Processors*). The loader utility for Blackfin processors also acts as a ROM splitter when evoked with the corresponding switches.

The loader pages (also called *loader properties pages*) show the default loader settings for the project's target processor. Refer to the CCES online help for information about the loader/splitter interface.

5

Loader/Splitter for ADSP-BF60x Blackfi Processors

This chapter explains how the loader/splitter utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable or non-bootable files for the ADSP-BF60x Blackfin processors.

Refer to the Introduction chapter for the loader utility overview. Loader operations specific to the ADSP-BF60x Blackfin processors are detailed in the following sections.

- *ADSP-BF60x Processor Booting*

  Provides general information on various boot modes.
- *ADSP-BF60x Processor Loader Guide*

  Provides information on how to build loader files.

ADSP-BF60x Processor Booting

For detailed information on the boot loader stream and modes for the ADSP-BF60x processors, refer to the booting chapter of the *ADSP-BF60x Blackfin Processor Hardware Reference*.

Refer to the processor's data sheet and hardware reference manual for detailed information on system configuration, peripherals, registers, and operating modes.

- Blackfin processor data sheets can be found at:

  http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/data-sheets/resources/index.html.
- Blackfin processor manuals can be found at:

  http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/manuals/resources/index.html or downloaded into the CCES IDE via **Help > Install New Software**.

The **ADSP-BF60x Part Numbers** table lists the part numbers that currently comprise the ADSP-BF60x family of Blackfin processors. Future releases of CrossCore Embedded Studio may support additional processors.

Table 19. ADSP-BF60x Part Numbers

ADSP-BF606	ADSP-BF608
ADSP-BF607	ADSP-BF609

This section covers the following topics:

- *ADSP-BF60x Processor Boot Modes*
- *ADSP-BF60x BCODE Field for Memory, RSI, and SPI Master Boot*
- *Building a Dual-Core Application*
- *CRC32 Protection*
- *Block Sizes*

ADSP-BF60x Processor Boot Modes

Table 20. ADSP-BF60x Processor Boot Modes

ADSP-BF60x Boot Mode	Boot (-b)	Boot Code (-bcode)	Notes
MEMORY	`-b MEMORY`	`-bcode #`[2]	Generic memory boot mode. Replaces `-b FLASH`. The argument for the -bcode switch is the `MDMACODE`, one of the supported numeric values specific to memory boot.
RSI0 master	`-b RSI`	`-bcode #`	The argument for the `-bcode` switch is the `RSICODE`, one of the supported numeric values specific to RSI master boot.
SPI0 master	`-b SPI`	`-bcode #`	The argument for the `-bcode` switch is the `SPIMCODE`, one of the supported numeric values specific to SPI master boot.
SPI0 slave	`-b SPISLAVE`		Boot code field in headers is not used for slave boot modes.

[2] Legal values for the `-bcode #` switch can be found in the booting chapter of the *ADSP-BF60x Blackfin Processor Hardware Reference*.

ADSP-BF60x Boot Mode	Boot (-b)	Boot Code (-bcode)	Notes
LP0 slave	`-b LPSLAVE`		Boot code field in headers is not used for slave boot modes.
UART0 slave	`-b UARTSLAVE`		Boot code field in headers is not used for slave boot modes.

ADSP-BF60x BCODE Field for Memory, RSI, and SPI Master Boot

A bootable loader stream is a series of boot blocks, each block beginning with a block header. Bits `0:3` of the ADSP-BF60x block header is a boot mode specific code field known as `BCODE`. The `-bcode #` switch controls what value is written to the `BCODE` field in the block headers in a bootable loader stream.

For detailed information on `.ldr` block headers, see the booting chapter of the *ADSP-BF60x Blackfin Processor Hardware Reference*.

The loader requires an explicit `BCODE` value when creating loader streams for master boot modes. For the ADSP-BF60x processors, this includes memory, RSI, and SPI master boot modes. When used in the context of a specific boot mode, `BCODE` is referred to by its boot-specific name: `MDMACODE` for memory boot, `RSICODE` for RSI boot, and `SPIMCODE` for SPI master boot.

Note:

The `-bcode` switch is not used for slave boot modes. The `BCODE` field is zero for slave boot modes.

When building loader streams, explicitly specify the `BCODE` field for the `.ldr` block headers:

1. Open the **Properties** dialog box for the project.
2. Choose **C/C++ Build** > **Settings**. The **Tool Settings** page appears.
3. Click **General** under **CrossCore Blackfin Loader**. The loader **General** properties page appears.
4. In **Boot code (-bcode)**, enter the number.

   The `BCODE` is specific for that particular boot mode. For `-bcode #` valid values, see the booting chapter of the *ADSP-BF60x Blackfin Processor Hardware Reference*.
5. Click **OK** to close the dialog box.
6. Click **Apply**.

If you do not specify `-bcode #`, the loader reports `Error ld0260`. For example, if `-bcode #` is not present when building a loader stream for RSI boot:

```
[Error ld0260]: Missing boot code header value for target ADSP-BF609 block headers.
   The -bcode # switch is required for specifying the boot code value
   for boot modes MEMORY, RSI, and SPI.
   For MEMORY boot, consult the MDMACODE table.
   For RSI boot, consult the RSICODE table.
```

```
   For SPI boot, consult the SPIMCODE table.
```

Building a Dual-Core Application

When building a dual-core application, use the `-NoFinalTag` switch to append the core 1 processing to core 0. The loader processes the input executables (`.dxe`) in order. If building at the command-line, place `DualCoreApp_Core1.dxe` after `DualCoreApp.dxe`:

```
elfloader -proc ADSP-BF609 -b SPI -bcode 0x1 DualCoreApp.dxe -NoFinalTag="DualCoreApp.dxe"
```

```
DualCoreApp_Core1.dxe -o DualCoreApp.ldr -f HEX -Width 8
```

Note:

Since the default startup code does not include functionality to allow core 0 to enable core 1, a convenient way to enable core 1 is to use the `adi_core_1_enable` function in the main program of `DualCoreApp`.

-NoFinalTag

Use the `-NoFinalTag` switch for multi-core booting.

Syntax

```
  -NoFinalTag[="DxeFile"]
```

Note:

Note there is no white space around = when specifying the executable name.

The `-NoFinalTag` switch directs the loader utility not to set the `FINAL` flag on the last code block from the `.dxe` file. When building an `.ldr` file with multiple `.dxe` files, this prevents the boot from halting after the first `.dxe` completes.

The loader processes the input `.dxe` files in the order they appear on the `elfloader` command-line. If the `-NoFinalTag` doesn't include the DxeFile file argument, it defaults to the first application `.dxe` file. Thus, the following command applies `-NoFinalTag` to `Core0.dxe`:

```
elfloader -proc ADSP-BF609 -b SPI -bcode 0x1 -init BF609_init_v00.dxe Core0.dxe
-NoFinalTag Core1.dxe -o DualCoreApp.ldr -f HEX -Width 8
```

You can explicitly specify the `.dxe` file to which `-NoFinalTag` applies. The following example is equivalent to the previous command:

```
elfloader -proc ADSP-BF609 -b SPI -bcode 0x1 -init BF609_init_v00.dxe Core0.dxe
-NoFinalTag="Core0.dxe" Core1.dxe -o DualCoreApp.ldr -f HEX -Width 8
```

The `-NoFinalTag` with a `.dxe` argument can appear multiple times. In the following example, there are four application `.dxe` files, and the `-NoFinalTag` switch appears twice, scoped to the appropriate `.dxe`:

```
elfloader -proc ADSP-BF609 -b spi -bcode 0x1 -f hex -width 8 -o multicore.ldr
Rel/App0_Core0.dxe Rel/App0_Core1.dxe Rel/App1_Core0.dxe Rel/App1_Core1.dxe
-NoFinalTag="App0_Core0.dxe" -NoFinalTag="App1_Core0.dxe"
```

As long as the .dxe files on the command line are uniquely named, you only need to specify the .dxe file name, without a full path. If the .dxe files are not uniquely named, the -NoFinalTag .dxe files must be matched by paths. When specifying pathnames, use forward slash (/).

```
elfloader -proc ADSP-BF609 -b spi -bcode 0x1 -f hex -width 8 -o multicore.ldr
AppA/App_Core0.dxe AppA/App_Core1.dxe AppB/App_Core0.dxe AppB/App_Core1.dxe
-NoFinalTag="AppA/App_Core0.dxe" -NoFinalTag="AppB/App_Core0.dxe" -verbose
```

Use -verbose to get a "per dxe file" build summary from elfloader:

```
*** Summary of -NoFinalTag usage
```

```
    NoFinalTag AppA/App_Core0.dxe was successful
```

```
    -NoFinalTag AppB/App_Core0.dxe was successful
```

Any -NoFinalTag .dxe file that did not get a match gets a warning, with or without -verbose:

```
[Warning ld0265]: 'unknown.dxe' from -NoFinalTag switch resulted in no matches.
```

Programming Memory on a Target Board

Use the CCES Device Programmer utility (cldp) for programming the memory on a target board.

In the building a dual-core application example above, DualCoreApp.ldr was built for boot mode SPIO master with format hex.

Driver:

```
ADSP-BF609_EZBoard/Examples/Device_Programmer/serial/w25q32bv_dpia/w25q32bv_dpia.dxe
```

```
cldp -proc ADSP-BF609 -emu HPUSB -driver w25q32bv_dpia.dxe
-cmd prog -erase affected -offset 0 -format hex -file DualCoreApp.ldr
```

Note:

You can save the device programmer commands to a file:

```
cldp -@ myPath/SPI_Flash_Programming.txt
```

See the **Device Programmer** section in the CCES online help for more information (search help for "device programmer").

CRC32 Protection

ADSP-BF60x CRC32 protection is implemented in hardware. The boot kernel provides mechanisms to allow each block to be verified using a 32-bit CRC.

When building a LDR file for CRC32 protection, use the -CRC32 <PolynomialCoefficient> switch.

-CRC32 (PolynomialCoefficient)

The -CRC32 switch directs the loader to generate CRC32 checksums. It uses the polynomial coefficient if specified, otherwise uses the default coefficient (0xD8018001).

Block Sizes

The loader creates blocks with payload and fill blocks using default maximum size and alignment that meets the requirements of the target hardware. Switches are available to override the defaults.

Table 21. ADSP-BF60x Processor Block Sizes

Switch	Description	Default	Requirements
`-MaxBlockSize #`	Specify the maximum block byte count	Loader uses maximum block size 0x7FFFFFF0 as default	The maximum block size is limited to `0xFFFFFFFC` bytes and must be a multiple of 4.
`-MaxFillBlockSize #`	Specify the maximum fill block byte count	Loader uses maximum fill block size 0x7FFFFFF0 as default	The maximum fill block size is limited to `0xFFFFFFFC` bytes and must be a multiple of 4.

ADSP-BF60x Processor Loader Guide

The loader utility post processes executable (`.dxe`) files and generates loader (`.ldr`) files. A loader file can be formatted as binary, ASCII or Intel hex style. An `.ldr` file contains the boot stream in a format expected by the on-chip boot kernel.

Loader utility operations depend on the loader properties, which control how the utility processes executable files. You select features, such as boot modes, boot kernels, and output file formats via the properties. The properties are specified on the loader utility's command line or the **Tool Settings** dialog box in the IDE (**CrossCore Blackfin Loader** pages). The default loader settings for a selected processor are preset in the IDE.

Note:

The IDE's **Tool Settings** correspond to switches displayed on the command line.

These sections describe how to produce a bootable (single and multiple) or non-bootable loader file:

- *CCES Loader and Splitter Interface for ADSP-BF60x Processors*
- *ROM Splitter Capabilities for ADSP-BF60x Processors*
- *ADSP-BF60x Loader Collateral*

CCES Loader and Splitter Interface for ADSP-BF60x Processors

Once a project is created in the CrossCore Embedded Studio IDE, you can change the project's output (artifact) type.

The IDE invokes the `elfloader.exe` utility to build the output loader file. To modify the default loader properties, use the project's **Tool Settings** dialog box. The controls on the pages correspond to the loader command-line switches and parameters. Refer to the online help for more information.

ROM Splitter Capabilities for ADSP-BF60x Processors

Note:

The readall feature is available for the automatic merging of fixed-position ROM data with code blocks within the bootable loader stream and typically supersedes the use of the legacy romsplitter feature described below.

When the loader utility is invoked with the splitter capabilities, it does not format the application data when transforming a `.dxe` file to an `.ldr` file. The splitter emits raw data only. Whether data and/or instruction sections are processed by the loader or by the splitter depends upon the LDF's `TYPE()` command. Sections declared with `TYPE(RAM)` are consumed by the loader, and sections declared by `TYPE(ROM)` are consumed by the splitter.

The contents of the ROM memory segments are extracted from the `.dxe`. The contents of the ROM segments get written to the `.ldr` file in raw format, each segment preceded by header words. The header consists of the following four 32-bit words written in unprefixed hex format:

Address	Start address of ROM memory segment (as defined in LDF)
Length	# of bytes extracted from the DXE for this segment
Control Word	32 bit control word 00 xx address multiply xx logical width xx physical width
Reserved word	All zeros

Example - ASCII Formatted Splitter .ldr File

This is an example of the header preceding the raw content extracted from the `.dxe` for segment `MEM1`.

Assume 256 bytes were written to the `.ldr` file and `MEM1` was defined in the LDF as:

```
  MEM1 { TYPE(ROM) WIDTH(8) START(0xB0000000) END( 0xB3FFFFFF) }

The -romsplitter .ldr output will be:
```

```

B0000000
00000100
00010101
00000000
00         <- content starts here
01
02
03
...
```

ADSP-BF60x Loader Collateral

The CrossCore Embedded Studio installation contains additional files and projects to assist with development and debugging of ADSP-BF60x applications which rely on booting functionality.

ROM Code

ROM code is available in `Blackfin/ldr/rom_code` of the CrossCore Embedded Studio installation.

Init Code

The sources and prebuilt executables for the init codes for the ADSP-BF609 EZ-KIT Lite are available in `Blackfin/ldr/init_code` of the CrossCore Embedded Studio installation.

The list of boot kernel symbols can be found in the booting chapter of the *ADSP-BF60x Blackfin Processor Hardware Reference*. Instructions for loading symbols can be found in the online help (search for "debugging the boot process").

Configuration information can be found in the `init_platform.h` file located in the corresponding part/silicon revision project. For example, for a ADSP-BF609 processor, `-si-revision 0.1`, see `Blackfin/ldr/init_code/BF609_init/BF609_init_v01/src/include/init_platform.h`.

You can include an `init code dxe` into the `*.ldr` file built for your application . To do so, use the **Initialization file** option in the IDE or the `-init` *`filename`* switch on the command line. Multiple *`-init`* switches are supported.

ROM Programming

ROM API headers for Blackfin processors, including the ADSP-BF609, are available in the CrossCore Embedded Studio installation. Build macros automatically configure `bfrom.h` (`Blackfin/include/bfrom.h`) for your build target processor.

The boot programming model is documented in the *ADSP-BF60x Blackfin Processor Hardware Reference*.

6

Loader/Splitter for ADSP-BF70x Blackfi Processors

This chapter explains how the loader/splitter utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable or non-bootable files for the ADSP-BF70x Blackfin processors.

Refer to the Introduction chapter for the loader utility overview. Loader operations specific to the ADSP-BF70x Blackfin processors are detailed in the following sections.

- *ADSP-BF70x Processor Booting*

  Provides general information on various boot modes.
- *ADSP-BF70x Processor Loader Guide*

  Provides information on how to build loader files.

ADSP-BF70x Processor Booting

For detailed information on the boot loader stream and modes for the ADSP-BF70x processors, refer to the booting chapter of the *ADSP-BF70x Blackfin+ Processor Hardware Reference*.

Refer to the processor's data sheet and hardware reference manual for detailed information on system configuration, peripherals, registers, and operating modes.

- Blackfin processor data sheets can be found at:

  http://www.analog.com/en/embedded-processing-dsp/blackfin/processors/data-sheets/resources/index.html.

The **ADSP-BF70x Part Numbers** table lists the part numbers that currently comprise the ADSP-BF70x family of Blackfin processors. Future releases of CrossCore Embedded Studio may support additional processors.

Table 22. ADSP-BF70x Part Numbers

ADSP-BF700	ADSP-BF701	ADSP-BF702
ADSP-BF702	ADSP-BF704	ADSP-BF706

ADSP-BF703	ADSP-BF705	ADSP-BF707

This section covers the following topics:

- *ADSP-BF70x Processor Boot Modes*
- *ADSP-BF70x BCODE Field for SPI Boot*
- *CRC32 Protection*
- *Block Sizes*

ADSP-BF70x Processor Boot Modes

Table 23. ADSP-BF70x Processor Boot Modes

ADSP-BF70x Boot Mode	Boot (-b)	Boot Code (-bcode)	Notes
SPI master	`-b SPI`	`-bcode #`[3]	The argument for the -`bcode` switch is the `SPIMCODE`, one of the supported numeric values specific to SPI master boot.
SPI slave	`-b SPISLAVE`	Not applicable	Boot code field in headers is not used for slave boot modes.
UART slave	`-b UARTSLAVE`	Not applicable	Boot code field in headers is not used for slave boot modes.

ADSP-BF70x BCODE Field for SPI Boot

A bootable loader stream is a series of boot blocks, each block beginning with a block header. Bits `0:3` of the ADSP-BF70x block header is a boot mode specific code field known as `BCODE`. The `-bcode #` switch controls what value is written to the `BCODE` field in the block headers in a bootable loader stream.

For detailed information on `.ldr` block headers, see the booting chapter of the *ADSP-BF70x Blackfin+ Processor Hardware Reference* manual.

[3] Values for the `-bcode #` switch can be found in the booting chapter of the *ADSP-BF70x Blackfin+ Processor Hardware Reference*.

The loader requires an explicit `BCODE` value when creating loader streams for master boot modes. For the ADSP-BF70x processors, this is SPI master boot mode. When used in the context of SPI mater boot mode, `BCODE` is referred to by its boot-specific name, `SPIMCODE`.

Note:

The `-bcode` switch is not used for slave boot modes. The `BCODE` field is zero for slave boot modes.

When building loader streams, explicitly specify the `BCODE` field for the `.ldr` block headers:

1. Open the **Properties** dialog box for the project.
2. Choose **C/C++ Build** > **Settings**. The `Tool Settings` page appears.
3. Click **General** under **CrossCore Blackfin Loader**. The loader **General** properties page appears.
4. In **Boot code (-bcode)**, enter the number.

   The `BCODE` is specific for that particular boot mode. For `-bcode #` valid values, see the booting chapter of the *ADSP-BF70x Blackfin+ Processor Hardware Reference*.
5. Click **OK** to close the dialog box.
6. Click **Apply**.

If you do not specify `-bcode #`, the loader reports `Error ld0274`. For example, if `-bcode #` is not present when building a loader stream for SPI boot:

```
[Error ld0274]: Missing boot code header value for target ADSP-BF707 block headers.
   The -bcode # switch is required for specifying the boot code value for
   the SPI boot modes. Consult the SPIMCODE table.
```

Secure Boot and Encrypted Images

The ADSP-BF70x processors provide security features for secure booting and encryption. Creating signed and optionally encrypted images is a multistage process using the Blackfin loader and signing utility (`signtool.exe`). The Blackfin loader builds the boot image in binary format, while the signtool protects that boot image.

1. LDR file creation:

   Build an `.ldr` file in CrossCore Embedded Studio or via the command-line.

   Note:

   For secure booting, the `.ldr` file must be built in binary format (`-f binary`). The signtool utility treats it as raw binary data with no interpretation. If the `.ldr` file is not built in binary format, the resulting secure-boot image is not be usable.
2. Secure-boot image creation:

   To sign/encrypt the boot image, use the signtool utility, either as a post-build step in the `.ldr` file build in the IDE or via the command-line. Refer to the signtool documentation in the **Utilities** appendix.

Using signtool to build a signed and optionally encrypted images in CCES:

The CCES IDE invokes the signtool utility after the main build, in a user-specified post-build step. The post-build step is helpful for the repetitive steps of creating a binary `.ldr` file and signing and optionally encrypting it:

To specify the post-build step:

Note:

The post-build step is executed only if the main build has executed successfully.

1. It is assumed that the project's **Build Artifact** type is a **Loader File**
2. Select the project in a navigation view:
   a. From the context menu, choose **Properties > C++ Build > Settings**. The **Settings** dialog box appears.
   b. Click **Build Steps**. The **Build Steps** dialog box appears.
3. In Post-build steps, type the signtool command for signing and optional encrypting:

```
signtool genkeypair -algo ecdsa224 -outfile keypair.der
```

Note:

The signtool sign step requires the keypair file as input in addition to the `.ldr` stream. Typically the keypair file is created in a one-time up-front setup operation.

4. Click **OK**

CRC32 Protection

ADSP-BF70x CRC32 protection is implemented in hardware. The boot kernel provides mechanisms to allow each block to be verified using a 32-bit CRC.

When building a LDR file for CRC32 protection, use the `-CRC32 <PolynomialCoefficient>` switch.

-CRC32 (PolynomialCoefficient)

The `-CRC32`switch directs the loader to generate CRC32 checksums. It uses the polynomial coefficient if specified, otherwise uses the default coefficient (`0xD8018001`).

Block Sizes

The loader creates blocks with payload and fill blocks using default maximum size and alignment that meets the requirements of the target hardware. Switches are available to override the defaults.

Table 24. ADSP-BF70x Processor Block Sizes

Switch	Description	Default	Requirements
`-MaxBlockSize #`	Specify the maximum block byte count	Loader uses maximum block size 0x7FFFFFF0 as default	The maximum block size is limited to `0xFFFFFFFC` bytes and must be a multiple of 4.
`-MaxFillBlockSize #`	Specify the maximum fill block byte count	Loader uses maximum fill block size 0x7FFFFFF0 as default	The maximum fill block size is limited to `0xFFFFFFFC` bytes and must be a multiple of 4.

ADSP-BF70x Processor Loader Guide

The loader utility post processes executable (`.dxe`) files and generates loader (`.ldr`) files. A loader file can be formatted as binary, ASCII or Intel hex style. An `.ldr` file contains the boot stream in a format expected by the on-chip boot kernel.

Loader utility operations depend on the loader properties, which control how the utility processes executable files. You select features, such as boot modes, boot kernels, and output file formats via the properties. The properties are specified on the loader utility's command line or the **Tool Settings** dialog box in the IDE (the **CrossCore Blackfin Loader** pages). The default loader settings for a selected processor are preset in the IDE.

Note:

The IDE's **Tool Settings** correspond to switches displayed on the command line.

These sections describe how to produce a bootable or non-bootable loader file:

- *CCES Loader and Splitter Interface for ADSP-BF70x Processors*
- *ROM Splitter Capabilities for ADSP-BF70x Processors*
- *ADSP-BF70x Loader Collateral*

CCES Loader and Splitter Interface for ADSP-BF70x Processors

Once a project is created in the CrossCore Embedded Studio IDE, you can change the project's output (artifact) type.

The IDE invokes the `elfloader.exe` utility to build the output loader file. To modify the default loader properties, use the project's **Tool Settings** dialog box. The controls on the pages correspond to the loader command-line switches and parameters. Refer to the online help for more information.

ROM Splitter Capabilities for ADSP-BF70x Processors

Note:

The readall feature is available for the automatic merging of fixed-position ROM data with code blocks within the bootable loader stream and typically supersedes the use of the legacy romsplitter feature described below.

When the loader utility is invoked with the splitter capabilities, it does not format the application data when transforming a `.dxe` file to an `.ldr` file. The splitter emits raw data only. Whether data and/or instruction sections are processed by the loader or by the splitter depends upon the LDF's `TYPE()` command. Sections declared with `TYPE(RAM)` are consumed by the loader, and sections declared by `TYPE(ROM)` are consumed by the splitter.

The contents of the ROM memory segments are extracted from the `.dxe`. The contents of the ROM segments get written to the `.ldr` file in raw format, each segment preceded by header words. The header consists of the following four 32-bit words written in unprefixed hex format:

Address	Start address of ROM memory segment (as defined in LDF)
Length	# of bytes extracted from the DXE for this segment
Control Word	32 bit control word 00 xx address multiply xx logical width xx physical width
Reserved word	All zeros

Example - ASCII Formatted Splitter .ldr File

This is an example of the header preceding the raw content extracted from the `.dxe` for segment `MEM1`.

Assume 256 bytes were written to the `.ldr` file and `MEM1` was defined in the LDF as:

```
  MEM1 { TYPE(ROM) WIDTH(8) START(0xB0000000) END( 0xB3FFFFFF) }

The -romsplitter .ldr output will be:

B0000000
00000100
00010101
```

```
00000000
00        <- content starts here
01
02
03
...
```

ADSP-BF70x Loader Collateral

The CrossCore Embedded Studio installation contains additional files and projects to assist with development and debugging of ADSP-BF70x applications which rely on booting functionality.

ROM Code

The sources for ROM code are not available in the CrossCore Embedded Studio to protect the ADSP-BF70x secure booting and encryption details.

Init Code

The sources and prebuilt executables for the init codes for the ADSP-BF707 EZ-KIT Lite are available in `Blackfin/ldr/init_code` of the CrossCore Embedded Studio installation.

The list of boot kernel symbols can be found in the booting chapter of the *ADSP-BF70x Blackfin+ Processor Hardware Reference*. Instructions for loading symbols can be found in the online help (search for "debugging the boot process").

Configuration information can be found in the `init_platform.h` file located in the corresponding part/silicon revision project. For example, for a ADSP-BF707 processor, `-si-revision 0.0`, see `Blackfin/ldr/init_code/BF707_init/BF707_init_v00/src/include/init_platform.h`.

You can include an `init code dxe` into the `.ldr` file built for your application . To do so, use the **Initialization file** option in the IDE or the `-init` *`filename`* switch on the command line. Multiple `-init` switches are supported.

ROM Programming

Note:

The ADSP-BF707 sample init code example is provided for non-secure booting. Init code or any form of callback is not supported in secure boot because the code is inherently insecure.

ROM API headers for Blackfin processors, including the ADSP-BF707 processors, are available in the CrossCore Embedded Studio installation. Build macros automatically configure `bfrom.h` (`Blackfin/include/bfrom.h`) for your build target processor.

The boot programming model is documented in the *ADSP-BF70x Blackfin+ Processor Hardware Reference*.

7

Loader for ADSP-21160 SHARC Processors

This chapter explains how the loader utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable files for the ADSP-21160 SHARC processors.

Refer to the Introduction chapter for the loader utility overview; the introductory material applies to all processor families. Refer to Loader for ADSP-21161 SHARC Processors chapter for information about the ADSP-21161 processors. Refer to Loader for ADSP-2126x/2136x/2137x/214xx SHARC Processors chapter for information about the ADSP-2126x and ADSP-2136x processors.

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

- *ADSP-21160 Processor Booting*

  Provides general information about various booting modes, including information about boot kernels.
- *ADSP-21160 Processor Loader Guide*

  Provides reference information about the loader utility's graphical user interface, command-line syntax, and switches.

ADSP-21160 Processor Booting

The processors support three boot modes: EPROM, host, link port, and no-boot (see the **ADSP-21160 Boot Mode Pins** and **ADSP-21160 Boot Modes** tables in *Boot Mode Selection*). 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.

Note:

The 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 processor reads boot data from an 8-bit external EPROM.
- When booting from a host processor through the external port, the processor accepts boot data from a 8- or 16-bit host microprocessor.

- When booting through the link port, the processor receives boot data as 4-bit wide data in link buffer 4.
- In no-boot mode, the processor begins executing instructions from external memory.

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

- *Power-Up Booting Process*
- *Boot Mode Selection*
- *ADSP-21160 Boot Modes*
- *ADSP-21160 Boot Kernels*
- *ADSP-21160 Interrupt Vector Table*
- *ADSP-21160 Multi-Application (Multi-DXE) Management*
- *ADSP-21160 Processor ID Numbers*

Power-Up Booting Process

The 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 boot 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 various processor models are shown in the **Processor DMA Channels** table.

Table 25. Processor DMA Channels

Processor	PROM Booting	Host Booting	Link Booting
ADSP-21160	`DMAC10`[4]	`DMAC10`	`DMAC8`

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 from location `0x40000` (ADSP-21160). The start addresses and reset vector addresses are summarized in the **Processor Start Addresses** table.

[4] See the **DMA Settings for ADSP-21160 EPROM Booting** table in *EPROM Boot Mode*.

Table 26. Processor Start Addresses

Processor	Start Address	Reset Vector Address[5]
ADSP-21160	0x40000	0x40004

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. See the **ADSP-21160 Boot Mode Pins** and **ADSP-21160 Boot Modes** tables.

Table 27. 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 port 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.
$\overline{\text{BMS}}$	I/O/T[6]	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.

[5] 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.

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

Table 28. 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

ADSP-21160 Boot Modes

The processors support these boot modes: EPROM, host, and link. The following sections describe each of the modes.

- *EPROM Boot Mode*
- *Host Boot Mode*
- *Link Port Boot Mode*
- *No-Boot Mode*

For multiprocessor booting, refer to *ADSP-21160 Multi-Application (Multi-DXE) Management*.

EPROM Boot Mode

The processor is configured for EPROM boot through the external port when the `EBOOT` pin is high and the `LBOOT` pin is low. These settings cause the $\overline{\text{BMS}}$ pin to become an output, serving as chip select for the EPROM. The **PROM Connections to Processors** table lists all PROM-to-processor connections.

Table 29. PROM Connections to Processors

Processor	Connection
ADSP-21160	PROM/EPROM connects to `DATA39-32` pins
ADSP-21xxx	Address pins of PROM connect to lowest address pins of any processor

Processor	Connection
ADSP-21xxx	Chip select connects to the $\overline{\texttt{BMS}}$ pin
ADSP-21160	Output enable connects to $\overline{\texttt{RDH}}$ 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 the **DMA Settings for ADSP-21160 EPROM Booting** table. 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 30. DMA Settings for ADSP-21160 EPROM Booting

DMA Setting	ADSP-21160 Processor
BMS space	8M x 8-bit
DMA channel	`DMAC10 = 0x4A1`
`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 boot undergoes the following boot-loading sequence:

1. The processor $\overline{\texttt{BMS}}$ 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 the **Processor Start Addresses** table in *Power-Up Booting Process*).
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 the **DMA Settings for ADSP-21160 EPROM Booting** table. 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 D39-32 external port data bus lines.
6. Data transfers begin and increment after each access. The external address lines (ADDR31-0) start at 0x80 0000.
7. The processor $\overline{\text{RD}}$ 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, theBSO bit is cleared in the SYSCON register, deactivating the $\overline{\text{BMS}}$ pin 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 BSO bit set (allowing the processor to gain access to the boot EPROM), and a third with the BSO bit cleared.

   When BSO=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 BSO is set.) When an external port DMA channel is being used in conjunction with the BSO bit, none of the other three channels may be used. In this mode, $\overline{\text{BMS}}$ 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 ($\overline{\text{Cx}}$ and $\overline{\text{ECx}}$) decrement after each EPROM transfer. When both counters reach zero, DMA transfer has stopped and RTI returns the program counter to the address where the kernel starts.

Note:

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 *ADSP-21160 Processor ID Numbers*).

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 Boot Mode

The ADSP-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 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 **Host Connections to ADSP-21160 Processors** table lists host connections to processors.

Table 31. Host Connections to ADSP-21160 Processors

Processor	Connection/Data Bus Pins
ADSP-21160	Host connected toDATA63-32 or DATA47-31 pins (based on HPM bits)
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 0x40004.

The parameter registers for the external port DMA channel (0, 6, or 10) are initialized as shown in the **DMA Settings for ADSP-21160 EPROM Booting** table (in *EPROM Boot Mode*), except that registers EIx, EMx and ECx are not initialized and no DMA transfers start.

The DMA channel control register (DMAC10) for the ADSP-21160 processor is initialized, which allows external port DMA enable and selects DTYPE for instruction words, PMODE for 16- to 48-bit word packing, 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 setting. The host boot file created by the loader utility requires the host processor to perform the following sequence of actions:

1. The host initiates the synchronous booting operation by asserting the processor HBR input pin, informing the processor that the default 8-/16-bit bus width is used. The host may optionally assert the CS chip select input to allow asynchronous transfers.
2. After the host receives the 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 the **Host Connections to ADSP-21160 Processors** table.

3. The host continues to write 16-bit words 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 of the 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 the **Processor Start Addresses** table in *Power-Up Booting Process*) 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 *Boot Kernels*.
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 the **Processor Start Addresses** table in *Power-Up Booting Process*) 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 Port Boot Mode

When link-boot the 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 8).

Link port 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.

No-Boot Mode

No-boot mode causes the processor to start fetching and executing instructions at address `0x800004` in external memory space for ADSP-21160 processors. All DMA control and parameter registers are set to their default initialization values. The loader utility is not intended to support no-boot mode.

ADSP-21160 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 `160_ldr.ldf`. The files are stored in the `SHARC/ldr` directory of CCES. The files shipped are listed in the **Default Boot Kernel Files** table.

Table 32. Default Boot Kernel Files

Processor	PROM Booting	Link Booting	Host Booting
ADSP-21160	`160_prom.asm`	`160_link.asm`	`160_host.asm`

Once the boot 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.

Processor Boot Steams

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 utility outputs the block body, which includes the actual data or instructions for placement in the processor memory. The loader utility, however, does not output a block

body if the actual data or instructions are all zeros in value. This type of block called a zero block. The **Boot Block Format** table describes the block header and body formats.

Table 33. 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 utility identifies the data type in the block header with a 16-bit tag that precedes the block. Each type of initialization has a unique tag number. The tag numbers and block types are shown in the **ADSP-21160 Processor Loader Block Tags** table.

Table 34. ADSP-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`
`0x0005`	`init dm32`	`0x0010`	`init dm64`
`0x0007`	`zero pm16`	`0x0011`	`zero pm64`
`0x0008`	`zero pm32`	`0x0012`	`init pm64`
`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 the block is a zero block.

The final section, which is identified by a tag of `0x0`, is called the final initialization section. This section has self-modifying code that, when executed, facilitates a DMA over the kernel, replacing it with 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 utility 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 mode) contains an `RTI` instruction. This `RTI` is inserted automatically by the loader utility 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 Modificatio 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 boot kernel is used.

When producing a boot-loadable file, the loader utility reads a processor executable file and uses information in it to initialize the memory. However, the loader utility 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 IDE (refer to online help for details), or rebuild the boot kernel file from the command line.

Note:

Specify the name of the modified kernel executable in the **Kernel file (-l)** field on the **CrossCore SHARC Loader > General** page of the **Tool Settings** dialog box.

If you modify the boot kernel for EPROM, host, or link boot modes, ensure that the `seg_ldr` memory segment is defined in the `.ldf` file. Refer to the source of the segment in the `.ldf` file located in the `<install_path>/SHARC/ldr` directory.

The loader utility generates a warning when vector address 0x40004 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 boot 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 instruction at this location.

The boot kernel project can be rebuilt from the IDE. The command-line can also be used to rebuild various default boot kernels for the processors.

EPROM Booting. The default boot kernel source file for the ADSP-21161 EPROM booting is 161_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 -21161 my_prom.asm
```

or

```
easm21k -proc ADSP-21161 my_prom.asm
```

```
linker -T 161_ldr.ldf my_prom.doj
```

Host Booting. The default boot kernel source file for the ADSP-21161 host booting is 161_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 -21161 my_host.asm
```

or

```
easm21k -proc ADSP-21161 my_host.asm
```

```
linker -T 161_ldr.ldf my_host.doj
```

Link Port Booting. The default boot kernel source file for the ADSP-21160 link port booting is 161_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 -21161 my_link.asm
```

or

```
easm21k -proc ADSP-21161 my_link.asm
```

```
linker -T 161_ldr.ldf my_link.doj
```

Rebuilding Boot Kernels

To rebuild the PROM boot kernel for the 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
```

ADSP-21160 Interrupt Vector Table

If an 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 boot 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-189: Link Port Tips and Tricks for ADSP-2116x* on the Analog Devices Web site for more information.

ADSP-21160 Multi-Application (Multi-DXE) Management

Currently, the loader utility generates single-processor loader files for host and link port boot modes. As a result, the loader utility supports multiprocessor EPROM boot mode only. The application code must be modified for a multiprocessor system boot in host and link port modes.

The loader utility can produce boot-loadable files that permit the SHARC processors in a multiprocessor system to boot from a single EPROM. In such a system, the $\overline{\texttt{BMS}}$ 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 boot of multiple processors is similar to a single-processor EPROM boot.

When booting a multiprocessor system through a single EPROM:

- Connect all $\overline{\texttt{BMS}}$ pins to EPROM.
- Processor with ID# of `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 **CrossCore SHARC Loader > Multiprocessor** page of the **Tool Settings** dialog box or specify the `-id1exe=` switch on the loader command line. These options specify the executable file targeted for a specific processor.

Do not use the `-id1exe=` 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.

ADSP-21160 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-21160 -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 `0` to `6`.

In the following example, the loader utility 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-21160 -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-21160 -bprom -id1exe=Input.dxe -id2ref=1
```

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

Processor Loader Guide

Loader utility operations depend on the loader properties, which control how the utility processes executable files. You select features, such as boot modes, boot kernels, and output file formats via the properties. The properties are specified on the loader utility's command line or the **Tool Settings** dialog box in the IDE (**CrossCore Blackfin Loader** pages). The default loader settings for a selected processor are preset in the IDE.

Note:

The IDE's **Tool Settings** correspond to switches displayed on the command line.

For detailed information about the processor loader properties page, refer to the online help.

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

- *Loader Command Line for ADSP-21160 Processors*
- *CCES Loader Interface for ADSP-21160 Processors*

Loader Command Line for Processors

The loader utility uses the following command-line syntax for the ADSP-21160 SHARC processors.

```
elfloader inputfile -proc part_number -switch [-switch]
```

where:

- *`inputfile`* - Name of the executable (`.dxe`) file 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` *`part_number`* - Part number of the processor (for example, `-proc ADSP-21160`) 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 utility. A list of all switches and their descriptions appear in *Loader Command-Line Switches for ADSP-21160 Processors*.

Note:

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

The following command line,

```
elfloader p0.dxe -bprom -fhex -l 160_prom.dxe -proc ADSP-21160
```

runs the loader utility 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 160_prom.exe` - Specifies `160_prom.exe` as the boot kernel file to be used in the boot-loadable file.
- `-proc ADSP-21160` - Identifies the processor model as ADSP-21160.

Loader Command-Line Switches for Processors

The **Loader Command-Line Switches** table is a summary of the loader switches for the ADSP-21160 processors.

Table 35. Loader Command-Line Switches

Switch	Description

`-bprom`
`-bhost`
`-blink`
`-bJTAG`
	Specifies the boot mode. The `-b` switch directs the loader utility to prepare a boot-loadable file for the specified boot mode. Valid boot modes include PROM, host, and link.
	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 utility automatically selects a default boot kernel based on the selected boot type (see *ADSP-21160 Boot Kernels*).

`-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 intended to boot a multiprocessor system. 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`.

Switch	Description
	In most cases, it is not necessary to use the `-e` switch: the loader utility processes the `.sm` files efficiently-includes a single copy of the code and data from each `.sm` file in a loader file.

`-fhex`
`-fASCII`
`-fbinary`
`-finclude`
`-fS1`
`-fS2`
`-fS3`
	Specifies the format of the boot-loadable file (Intel hex-32, ASCII, S1, S2, S3, binary, or 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.

`-h`
or
`-help`
	Command-line help. Outputs a list of the command-line switches to standard out and exits. Type `elfloader -proc ADSP-21xxx -h`, where `xxx` is `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 utility to use the processor ID (`#`) for the corresponding executable file (*`filename`* parameter) when producing a boot-loadable file for a multiprocessor system. This switch is used 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 *ADSP-21160 Processor ID Numbers*.
`-id#ref=`*`N`*	Points the processor ID (#) loader jump table entry to the ID (*`N`*) 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 *ADSP-21160 Processor ID Numbers*.

Switch	Description
`-l` *`kernelfile`*	Directs the loader utility 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 utility searches in the processor-specific loader directory for the boot kernel file as described in *ADSP-21160 Boot Kernels*.
`-o` *`filename`*	Directs the loader utility to use the specified *`filename`* as the name for the loader output file. If not specified, the default name is *`inputfile`*`.ldr`.
`-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 utility starts the EPROM file at address `0x0`; this EPROM address corresponds to `0x800000` on ADSP-21160 processors.
`-proc` *`processor`*	Specifies the processor. This a mandatory switch.
`-si-revision [none|any|`*`x.x`*`]`	Sets revision for the build, with *`x.x`* being the revision number for the processor hardware. If `-si-revision` is not used, the target is a default revision from the supported revisions.
`-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.
`-use32bitTagsfor ExternalMemoryBlocks`	Directs the loader utility 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

Switch	Description
	if the external memory sections are packed by the linker and do not need the loader utility to pack them again.
`-v`	Outputs verbose loader utility messages and status information as the utility processes files.
`-version`	Directs the loader utility 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-21160 -version` to display version information of both loader drive and SHARC loader utility.

CCES Loader Interface for Processors

Once a project is created in the CrossCore Embedded Studio IDE, you can change the project's output (artifact) type.

The IDE invokes the `elfloader.exe` utility to build the output loader file. To modify the default loader properties, use the project's **Tool Settings** dialog box. The controls on the pages correspond to the loader command-line switches and parameters (see *Loader Command-Line Switches for ADSP-21160 Processors*).

The loader pages (also called *loader properties pages*) show the default loader settings for the project's target processor. Refer to the CCES online help for information about the loader interface.

The CCES splitter interface for the ADSP-21160 processors is documented in the Splitter for SHARC Processors chapter.

8

Loader for ADSP-21161 SHARC Processors

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

Refer to the Introduction chapter for the loader utility overview; the introductory material applies to all processor families. Refer to the Loader for ADSP-21160 SHARC Processors chapter for information about the ADSP-21160 processors. Refer to the Loader for ADSP-2126x/2136x/2137x/214xx SHARC Processors chapter for information about the ADSP-2126x, ADSP-2136x, ADSP-2137x, ADSP-2146x, ADSP-2147x, and ADSP-2148x processors.

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

- *ADSP-21161 Processor Booting*

  Provides general information about various boot modes, including information about boot kernels.
- *ADSP-21161 Processor Loader Guide*

  Provides reference information about the loader utility's graphical user interface, command-line syntax, and switches.

ADSP-21161 Processor Booting

The processors support five boot modes: EPROM, host, link port, SPI port, and no-boot (see tables **ADSP-21161 Boot Mode Pins** and **ADSP-21161 Boot Mode Pin States** in *Boot Mode Selection*). 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, the ADSP-21161 processors begin executing instructions from external memory.

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

- *Power-Up Booting Process*
- *Boot Mode Selection*
- *ADSP-21161 Processor Boot Modes*
- *ADSP-21161 Processor Boot Kernels*
- *Boot Kernel Modification and Loader Issues*
- *ADSP-21161 Processor Interrupt Vector Table*
- *ADSP-21161 Multi-Application (Multi-DXE) Management*

Power-Up Booting Process

The 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 boot kernel. When executed, the boot kernel facilitates booting of user application code. The combination of the boot kernel and application code comprises the boot-loadable (`.ldr`) file.

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

1. Based on the boot mode, 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 mode 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 processor's boot mode. The **ADSP-21161 Boot Mode Pins** and **ADSP-21161 Boot Mode Pin States** tables show how the pin states correspond to the modes.

Table 36. 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 $\overline{\texttt{BMS}}$ pins determine booting mode.

Pin	Type	Description
LBOOT	I	Link port 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.
$\overline{\text{BMS}}$	I/O/T[7]	Boot memory select - when boot-loading from EPROM (EBOOT=1 and LBOOT=0), the 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.

Table 37. ADSP-21161 Boot Mode Pin States

EBOOT	LBOOT	BMS	Booting Mode
1	0	Output	EPROM (connects $\overline{\text{BMS}}$ 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)

ADSP-21161 Processor Boot Modes

The processors support these boot modes: EPROM, host, link, and SPI. The following section describe each of the modes.

- *EPROM Boot Mode*
- *Host Boot Mode*
- *Link Port Boot Mode*
- *SPI Port Boot Mode*
- *No-Boot Mode*

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

Note:

For multiprocessor booting, refer to *ADSP-21161 Multi-Application (Multi-DXE) Management*.

EPROM Boot Mode

EPROM boot via the external port is selected when the EBOOT input is high and the LBOOT input is low. These settings cause the 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. CrossCore Embedded Studio includes loading routines (boot kernels) that can load entire programs; see *ADSP-21161 Processor Boot Kernels* for more information.

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

Note:

Be aware that DMA channel differences between the ADSP-21161 and previous SHARC processors account for boot differences. Even with these differences, the ADSP-21161 processor supports the same boot capability and configuration as previous SHARC processors. The DMACx 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 previous SHARC processors, the ADSP-21161 processor boots from DATA23—16.

The processor determines the booting mode at reset from the EBOOT, LBOOT, andBMS pin inputs. When EBOOT=1 and LBOOT=0, the processor boots from an EPROM through the external port and uses BMS as the memory select output. For information on boot mode selection, see the boot memory select pin descriptions in tables **ADSP-21161 Boot Mode Pins** and **ADSP-21161 Boot Mode Pin States** (in *Boot Mode Selection*).

Note:

When using any of the power-up boot modes, 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 boot (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 BMS, and its output enable should be connected to RD.

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

Note:

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 theWAIT register are initialized to perform asynchronous access and generate seven wait states (8 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.

The **DMA Channel 10 Parameter Registers for EPROM Booting** table 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 ($\overline{\text{BMS}}$ 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.

Table 38. 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 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

The following sequence occurs at system start-up, when the processor $\overline{\text{RESET}}$ 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 the **DMA Channel 10 Parameter Registers for EPROM Booting** table.
3. The BMS pin 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 RD strobe asserts as in a normal memory access with seven wait states (8 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 BMS pin; other memory select pins are disabled.

The master DMA internal and external count registers (CEP0)ECEP0/ decrement 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. The BMS pin 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 boot and is executing instructions normally. The first instruction at the EP0Iinterrupt 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 Boot Mode

The processor can boot from a host processor through the external port. Host booting is selected when theEBOOT and LBOOT inputs are low and 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. CrossCore Embedded Studio includes loading routines (boot kernels) that can load entire programs; refer to *ADSP-21161 Processor Boot Kernels* for more information.

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

Note:

DMA channel differences between the ADSP-21161 and previous SHARC family processors account for boot differences. Even with these differences, the ADSP-21161 processors support the same boot capability and configuration as previous SHARC processors. The DMAC10 register default values differ because the ADSP-21161 processor 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 $\overline{\text{BMS}}$ pin inputs. When EBOOT=0, LBOOT=0, and BMS=1, the processor boots from a host through the external port. Refer to tables **ADSP-21161 Boot Mode Pins** and **ADSP-21161 Boot Mode Pin States** in *Boot Mode Selection* for boot mode selections.

When using any of the power-up boot modes, address 0x40004 should not contain a valid instruction. Because it is not executed during the boot 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.

The **DMA Channel 10 Parameter Register for Host Boot** table shows how the DMA channel 10 parameter registers are initialized at reset for host boot. 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.

Table 39. DMA Channel 10 Parameter Register for Host Boot

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

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

At system start-up, when the processor 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 the **DMA Channel 10 Parameter Register for Host Boot** table.
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 boot mode and is executing instructions normally. The first instruction at the EP0Iinterrupt 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 Boot Mode

Link port boot 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. CrossCore Embedded Studio includes loading routines (boot kernels) that load an entire program through the selected port; refer to *ADSP-21161 Processor Boot Kernels* for more information.

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

Note:

DMA channel differences between the ADSP-21161 and previous SHARC family processors account for boot differences. Even with these differences, the ADSP-21161 processors support the same boot capabilities and configuration as the previous SHARC processors.

The processor determines the boot mode at reset from the EBOOT, LBOOT and 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 tables **ADSP-21161 Boot Mode Pins** and **ADSP-21161 Boot Mode Pin States** in *Boot Mode Selection*.

Note:

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

In link port boot, 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.

The **DMA Channel 8 Parameter Register for Link Port Boot** table 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 boot to allow link buffer 0 to receive 48-bit data.

Table 40. DMA Channel 8 Parameter Register for Link Port Boot

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

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 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 Boot Mode

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 boot 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. CrossCore Embedded Studio includes loading routines (boot kernels) which load an entire program through the selected port. See *ADSP-21161 Processor Boot Kernels* for more information.

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

The processor determines the boot mode at reset from the EBOOT, LBOOT, and $\overline{\text{BMS}}$ 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 tables **ADSP-21161 Boot Mode Pins** and **ADSP-21161 Boot Mode Pin States** in *Boot Mode Selection*.

Note:

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

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

The **DMA Channel 8 Parameter Register for SPI Port Boot** table 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) boot 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 41. DMA Channel 8 Parameter Register for SPI Port Boot

Parameter Register	Initialization Value
IISRX	0x0004 0000
IMSRX	Uninitialized (increment by 1 is automatic)
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 utility does not produce the code for no-boot execution.

ADSP-21161 Processor 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 are shipped with CrossCore Embedded Studio; refer to the **Default Boot Kernel Files** table.

Table 42. Default Boot Kernel Files

PROM Booting	Link Booting	Host Booting	SPI Booting
`161_PROM.dxe`	`161_LINK.dxe`	`161_HOST.dxe`	`161_SPI.dxe`

Boot kernels are loaded at processor reset into the `seg_ldr` memory segment, which is defined in the `161_ldr.ldf`. The file is stored in the `<install_path>/SHARC/ldr` directory.

Processor Boot Streams

The loader utility 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 mode, as follows.

`0xLLLLLLLL`	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 boot kernel examines the tag to determine the type of data or instruction being loaded. The **ADSP-21161N Processor Block Tags** table lists the ADSP-21161N processor block tags.

Table 43. ADSP-21161N Processor Block 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
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 Modificatio 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 utility is used.

If you do not specify a boot kernel file via the loader pages of the **Tool Settings** dialog box in the IDE (or via the -l *kernelfile* command-line switch), the loader utility places a default boot kernel in the loader output file (see *ADSP-21161 Processor Boot Kernels)*) based on the specified boot mode.

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 is in the 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 IDE (refer to 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 using command lines as follows.

EPROM Boot. 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 Boot. 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 Boot. 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 Boot. 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 directory of the of the target processor.

Because the loader utility uses the address of 0x40004 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 boot 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 utility generates a warning if the vector address 0x40004 does not contain NOP or IDLE.

Note:

When creating the loader file, specify the name of the customized boot kernel executable in the **Kernel file (-l)** field on the **CrossCore SHARC Loader** page of the **Tool Settings** dialog box.

ADSP-21161 Processor Interrupt Vector Table

If the 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.

ADSP-21161 Multi-Application (Multi-DXE) Management

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

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

- *Boot From a Single EPROM*
- *Sequential EPROM Boot*

Regardless of the method, the 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

Boot From a Single EPROM

The loader utility can produce boot-loadable files that permit SHARC processors in a multiprocessor system to boot from a single EPROM. The $\overline{\text{BMS}}$ 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 Boot

Set the `EBOOT` pin of the processor with ID# of `1` high for EPROM booting. The other processors should be configured for host boot (`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.

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 utility 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-21161 -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 directed by the command-line switch `-id#ref=N` 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 referenced by processor `2`.

The loader utility 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 utility operations depend on the loader properties, which control how the utility processes executable files. You select features, such as boot modes, boot kernels, and output file formats via the properties. The properties are specified on the loader utility's command line or the **Tool Settings** dialog box in the IDE (**CrossCore Blackfin Loader** pages). The default loader settings for a selected processor are preset in the IDE.

Note:

The IDE's **Tool Settings** correspond to switches displayed on the command line.

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

- *Loader Command Line for ADSP-21161 Processors*
- *CCES Loader Interface for ADSP-21161 Processors*

Loader Command Line for Processors

The loader utility uses the following command-line syntax for the ADSP-21161 SHARC processors.

```
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 utility. A list of all switches and their descriptions can be found in *Loader Command-Line Switches for ADSP-21161 Processors*.

Note:

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 utility 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 utility 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 of `1` (see *Processor ID Numbers)*).
- `-id2exe=Input2.dxe` - Identifies `Input2.dxe`. as the executable file to process into a boot-loadable file for a processor with ID of `2` (see *Processor ID Numbers)*).

Loader Command-Line Switches for ADSP-21161 Processors

The **ADSP-21161 Loader Command Line Switches** table is a summary of the loader switches for the ADSP-21161processors.

Table 44. ADSP-21161 Loader Command Line Switches

Switch	Description

`-bprom`
`-bhost`
`-blink`
`-bspi`
	Specifies the boot mode. The `-b` switch directs the loader utility to prepare a boot-loadable file for the specified boot mode. The valid modes (boot types) are PROM, host, link, and SPI. If the switch does not appear on the command line, the default is `-bprom`. To use a custom boot kernel, the boot mode selected with the `-b` switch must correspond with the boot kernel selected with the `-l` *`kernelfile`* switch. Otherwise, the loader utility automatically selects a default boot kernel based on the selected boot type (see *ADSP-21161 Processor Boot Kernels*).

`-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 intended to boot a multiprocessor system. To omit multiple `.sm` files, repeat the switch and its parameter multiple times on the command line. For example, to omit two files, use: `-efileA.SM -efileB.SM`. In most cases, it is not necessary to use the `-e` switch: the loader utility processes the `.sm` files efficiently (includes a single copy of the code and data from each `.sm` file in a loader file).

`-fhex`
`-fASCII`
`-fbinary`
`-finclude`
`-fS1`
`-fS2`
`-fS3`
	Specifies 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 mode selection (`-b` switch):
	• For a PROM boot, select a hex-32, S1, S2, S3, ASCII, or include format.
	• For host or link boot, select an ASCII, binary, or include format.
	• For SPI boot, select an ASCII or binary format.

Switch	Description

`-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 the ADSP-21161 processors. By default, the `-h` switch alone provides help for the loader driver.

`-hostwidth {8|16|32}`	Sets up the word width for the `.ldr` file. 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 utility to use the processor ID (`#`) for the corresponding executable file (*`filename`*) when producing a boot-loadable file for EPROM boot 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*.
`-id#ref=`*`N`*	Points the processor ID (`#`) loader jump table entry to the ID (*`N`*) 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*.
`-l` *`kernelfile`*	Directs the loader utility 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 mode selected with the `-b` switch.

Switch	Description
	If the `-l` switch does not appear on the command line, the loader utility searches for a default boot kernel file. Based on the boot mode (`-b` switch), the loader utility searches in the processor-specific loader directory for the boot kernel file as described in *ADSP-21161 Processor Boot Kernels*.
`-o` *filename*	Directs the loader utility to use the specified *filename* as the name for the loader output file. If not specified, the default name is *inputfile*`.ldr`.
`-noZeroBlock`	The `-noZeroBlock` switch directs the loader utility not to build zero blocks.
`-p` *address*	Directs the loader utility to start the boot-loadable file at the specified address in the EPROM. This EPROM address corresponds to `0x8000000` on the processor. If the `-p` switch does not appear on the command line, the loader utility starts the EPROM file at address `0x0`.
`-proc ADSP-21161`	Specifies the processor. This is a mandatory switch.
`-si-revision [none|any|`*x.x*`]`	Sets revision for the build, with *x.x* being the revision number for the processor hardware. If `-si-revision` is not used, the target is a default revision from the supported revisions.
`-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 related directly 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 utility processes files.

Switch	Description
`-version`	Directs the loader utility 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.

CCES Loader Interface for Processors

Once a project is created in the CrossCore Embedded Studio IDE, you can change the project's output (artifact) type.

The IDE invokes the `elfloader.exe` utility to build the output loader file. To modify the default loader properties, use the project's **Tool Settings** dialog box. The controls on the pages correspond to the loader command-line switches and parameters (see *Loader Command-Line Switches for ADSP-21161 Processors*).

The loader pages (also called *loader properties pages*) show the default loader settings for the project's target processor. Refer to the CCES online help for information about the loader interface.

The CCES splitter interface for the ADSP-21161 processors is documented in the Splitter for SHARC Processors chapter.

9

Loader for ADSP-2126x/2136x/2137x/214xx SHARC Processors

This chapter explains how the loader utility (`elfloader.exe`) is used to convert executable (`.dxe`) files into boot-loadable files for the ADSP-2126x, ADSP- 2136x, ADSP-2137x, and ADSP-214xx SHARC processors.

Note:

For information on specific SHARC processors, refer to the product- specific hardware reference, programming reference, and data sheet.

Refer to the Introduction chapter for the loader utility overview; the introductory material applies to all processor families. Refer to the Loader for ADSP-21160 SHARC Processors chapter for information about the ADSP-21160 processors. Refer to the Loader for ADSP-21161 SHARC Processors chapter for information about the ADSP-21161 processors.

Loader operations specific to the ADSP-2126x/2136x/2137x/214xx SHARC processors are detailed in the following sections.

- *ADSP-2126x/2136x/2137x/214xx Processor Booting*

  Provides general information about various booting modes, including information about boot kernels.
- *ADSP-2126x/2136x/2137x/214xx Processor Loader Guide*

  Provides reference information about the graphical user interface, command-line syntax, and switches.

ADSP-2126x/2136x/2137x/214xx Processor Booting

ADSP-2126x, ADSP-2136x, ADSP-2137x and ADSP-214xx processors can be booted from various sources:

- The boot source is selected via the boot configuration pins during power-up.
- All processors do support 8-bit parallel flash boot mode and SPI master/slave boot modes.
- The ADSP-2146x processor does support link port boot mode.
- In no-boot mode, the processor fetches and executes instructions directly from the internal ROM memory, bypassing the boot kernel entirely. The loader utility does not produce a file supporting the no-boot mode.

- SPI master boot does support three cases: SPI master (no address), SPI PROM (16-bit address), and SPI flash (24-bit address).
- The ADSP-21367/21368/21369, ADSP-21371/21375, and ADSP-214xx processors support parallel flash multiprocessing boot by decoding the processor ID number from the boot stream.

Note:

Only the ADSP-21367/21368/21369, ADSP-21371/21375, and ADSP-214xx processors are supporting multiprocessing, so the loader can use an ID lookup table between the kernel and the rest of the application.

Note:

Upon ADSP-2126x processors, no boot mode from external memory with internal/external IVT option is no longer supported.

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

- *Power-Up Booting Process*
- *ADSP-2126x/2136x/2137x/214xx Processors Interrupt Vector Table*
- *General Boot Definitions*
- *Boot Mode Selection*
- *Boot DMA Configuration Settings*
- *ADSP-2126x/2136x/2137x/214xx Processors Boot Kernels*
- *ADSP-2126x/2136x/2137x/214xx Processor Boot Streams*

Power-Up Booting Process

The ADSP-2126x, ADSP-2136x, ADSP-2137x, ADSP-214xx 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 a boot 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 or a 256-word (48-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 (48-bit) 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. Note that the DAI/DPI pins are enabled by default for correct booting over the peripherals.

ADSP-2126x/2136x/2137x/214xx Processor Interrupt Vector Table

If the ADSP-2126x, ADSP-2136x, ADSP-2137x or ADSP-214xx processor is booted from an external source (PROM or SPI or link port), the IVT is always located in internal memory.

General Boot Definition

The boot source is determined by sampling the state of the boot configuration pins.

On the ADSP-2126x/2136x/2137x/214xx processors, the boot type is determined by sampling the state of the `BOOT_CFG1-0` pins (`BOOT_CFG2-0` pins for ADSP-214xx processors). The truth table for boot configuration pins can be found in the processor data sheet.

Note all referred RESET vector locations in this chapter are dependent on the processor type and are defined as follows:

ADSP-2126x	0x80004
ADSP-2136x/2137x	0x90004
ADSP-214xx	0x8C004

All processors operate with an interrupt vector table (IVT) located in internal memory block0 which is used to load and execute the kernel (256x48-bit words) located at the following address:

ADSP-2126x	0x80000 - 0x800FF
ADSP-2136x/2137x	0x90000 - 0x900FF
ADSP-214xx	0x8C000 - 0x8C0FF

Boot Mode Selection

Note:

On the ADSP-2126x/2136x/2137x/214xx processors, the boot type is determined by sampling the state of the `BOOTCFGx` pins, described in the **ADSP-2126x/2136x/2137x Boot Mode Selection** and **ADSP-214xx Boot Mode Selection** tables, and the selection of the corresponding boot kernel in the elfloader.

A description of each boot type follows the tables.

Table 45. ADSP-2126x/2136x/2137x Boot Mode Selection

HW Pins BOOT_CFG[1-0]	Boot Mode	SW Elfloader Settings Boot Mode 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	No boot (not available on all processors)	Does not use the loader utility

Table 46. ADSP-214xx Boot Mode Selection

HW Pins BOOT_CFG[2-0]	Boot Mode	SW Elfloader Settings Boot Mode Selection
000	SPI slave	-bspislave
001	SPI master (SPI flash, SPI PROM, or a host processor via SPI master mode)	-bspiflash -bspiprom -bspimaster
010	EPROM boot via the parallel port	-bprom
011	No boot (not available on all processors)	Does not use the loader utility
100	Link Port 0 boot	-blink

Boot DMA Configuratio Settings

All peripheral boot mode use a 256 words instruction length DMA (as described in "power-up booting process" which does load the kernel into the internal memory. At reset, the control and parameter registers settings of the peripheral's boot DMA can be found at:

- For ADSP-2126x products refer to the *ADSP-2126x SHARC Processor Hardware Reference*
- For ADSP-2136x products refer to the *ADSP-2136x SHARC Processor Hardware Reference*
- For ADSP-21367/8/9 and ADSP-2137xx products refer to the *ADSP-2137x SHARC Processor Hardware Reference*
- For ADSP-214xx products refer to the *ADSP-214xx SHARC Processor Hardware Reference*

PROM Boot Mode

All processors which support external memory typically have memory I/O size which is different to normal word of 32-bit. The linker's width command takes care about logical and physical addressing.

Packing Options for External Memory

The`WIDTH()` command in the linker specifies which packing mode should be used to initialize the external memory: `WIDTH(8)` for 8-bit memory or `WIDTH(16)` for 16-bit memory.

The loader utility packs the external memory data from the `.dxe` file according to the linker's `WIDTH()` command. The loader utility unpacks the data from the executable file and packs the data again in the loader file if the data is packed in the `.dxe` file due to the packing command in the linker description (`.ldf`) file.

The next section lists the different packing options depending on model, and data versus instruction fetch.

Multiplexed Parallel Port

The ADSP-2126x/2136x processors do use a parallel port which does multiplex the address and data (in order to save pin count). The **Data Packing Options for Parallel Port** table and following sections list the different packing options, depending on part numbers and data versus instructions.

Table 47. Data Packing Options for Parallel Port

Packing Options	WIDTH (8)	WIDTH (16)	WIDTH (32)
ADSP-2126x	Yes	Yes	No
ADSP-2136x	Yes	Yes	No

For ADSP-2126x processors, the external memory address ranges are `0x10 00000-0x2F FFFFF`. For ADSP-2136x processors, the external memory address ranges are `0x12 00000-0x1203FFF`. External instruction fetch is not supported by these processors.

AMI/SDRAM/DDR2

The ADSP-21367/8/9 processors external port is used to arbitrate between AMI and SDRAM/DDR2 access.

Table 48. Data Packing Options for External Port

Packing Options	WIDTH (8)	WIDTH (16)	WIDTH (32)
ADSP-21367/8/9	AMI	AMI/SDRAM	AMI/SDRAM
ADSP-2137x	AMI	AMI/SDRAM	AMI/SDRAM
ADSP-214xx	AMI	AMI/SDRAM/DDR2	No

For ADSP-2137x/214xx processors, the external memory address range for ISA instruction fetch (bank0 only) is `0x20 0000-0x5FFFFF`.

For ADSP-214xx processors, the external memory address range for VISA instruction fetch (bank0 only) is `0x60 0000-0xFFFFFF`.

Table 49. Instruction Fetch Packing Options for External Port

Packing Options	WIDTH (8)	WIDTH (16)	WIDTH (32)
ADSP-2137x	AMI	AMI/SDRAM	AMI/SDRAM
ADSP-214xx	AMI	AMI/SDRAM/DDR2	No

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

Both SPI boot modes support booting from 8-, 16-, or 32-bit SPI devices. In all SPI boot modes, 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 SPI shift register to generate 32-bit words least significant bit (LSB) first, which are then shifted into internal memory.

When booting, the ADSP-2126x/2136x/2137x/214xx processor expects to receive words into the `RXSPI` buffer seamlessly. This means that bits are received continuously without breaks in the $\overline{\texttt{SPIDS}}$ link. For different SPI host sizes, the processor expects to receive instructions and data packed in a least significant word (LSW) format.

SPI Slave Boot Mode

In SPI slave boot mode, the host processor initiates the booting operation by activating the SPICLK signal and asserting the 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.

Note:

The processor's $\overline{\text{SPIDS}}$ pin should not be tied low. When in SPI slave mode, including booting, the SPIDS signal is required to transition from high to low. SPI slave booting uses the default bit settings shown in the **SPI Slave Boot Bit Settings** table.

Table 50. 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

SPI Master Boot Modes

In SPI master boot mode, the ADSP-2126x/2136x/2137x/214xx processor initiates the booting operation by:

1. Activating the SPICLK signal and asserting the FLAG0 signal to the active low state
2. Writing the read command 0x03 and 24-bit address 0x00000 to the slave device

Note:

The processor's SPIDS pin should not be tied low. When in SPI slave mode, including booting, the SPIDS signal is required to transition from high to low. SPI slave booting uses the default bit settings shown in the **SPI Slave Boot Bit Settings** table (see *SPI Slave Boot Mode*).

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, received while the 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.

Note:

SPI master boot mode is used when the processor is booting from an SPI compatible serial PROM, serial flash, or slave host processor.

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 LSB format. See *Bit-Reverse Option for SPI Master Boot Modes* for details.

Also, because the processor always transmits 32 bits before it begins reading boot data from the slave device, the loader utility 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 utility 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* for details.

The **SPI Master Mode Booting Using Various Serial Devices** figure 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.

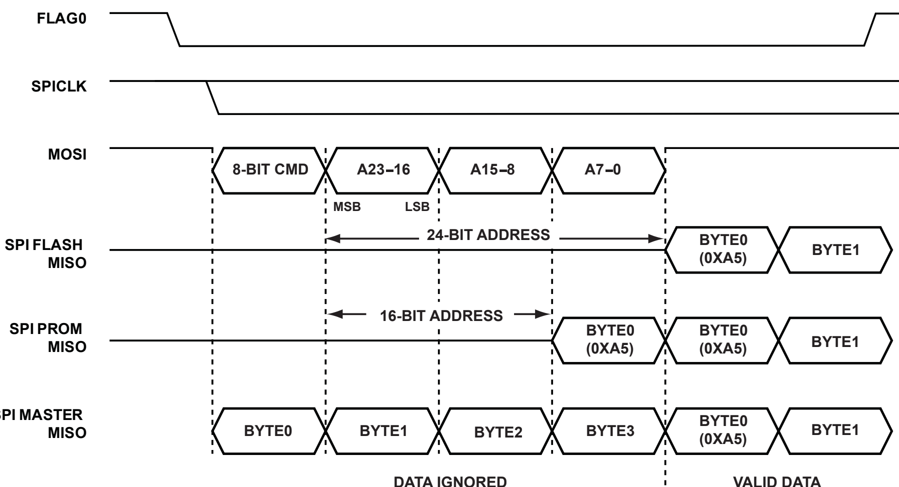


Figure 13. SPI Master Mode Booting Using Various Serial Devices

Bit-Reverse Option for SPI Master 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 of the `MSBF` bit (`SPICTL` register) is cleared which sets order to be LSB first. 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 prepend the word `0xA5` to the beginning of the byte stream. During SPI read command, the SPI port discards the first byte read from the SPI via the MISO line (see the **Initial Word for SPI Master and SPI PROM in .ldr File** table).

Table 51. Initial Word for SPI Master and SPI PROM in .ldr File

Boot Mode	Additional Word	-hostwidth		
		32	16	8
SPI master[8]	`0xA5000000`	`A5000000`	`0000`	`00`

[8] Initial word for SPI master boot type is always 32 bits. See the **SPI Master Mode Booting Using Various Serial Devices** figure in *Bit-Reverse Option for SPI Master Boot Modes* for explanation.

Boot Mode	Additional Word	-hostwidth		
		32	16	8
			A500	00
				00
				A5
SPI PROM[9]	0xA5	A5	A5	A5

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.

Note:

Initial word option is only required for SPI master/prom boot mode. The CrossCore Embedded Studio tools automatically handle this in the loader file generation process for SPI boot devices.

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 the **Default Settings for PROM and SPI Boot Modes** table).

Table 52. 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`

[9] Initial word for SPI PROM boot type is always 8 bits. See the **SPI Master Mode Booting Using Various Serial Devices** figure in *Bit-Reverse Option for SPI Master Boot Modes* for explanation

Boot Type Selection	Host Width	Output Format	Bit Reverse	Initial Word
`-bspiprom`	8	Intel hex	Yes	`0xa5`

Booting From an SPI Flash (24-Bit Address)

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.

Booting From an SPI PROM (16-Bit Address)

The the **SPI Master Mode Booting Using Various Serial Devices** figure in *Bit-Reverse Option for SPI Master Boot Modes* shows the initial 32-bit word sent out from the processor from the perspective of the serial PROM device.

As shown in the figure, 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.

Booting From an SPI Host Processor (No Address)

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 the **SPI Master Mode Booting Using Various Serial Devices** figure in *Bit-Reverse Option for SPI Master Boot Modes*). 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.

Reserved (No Boot) Mode

In no boot mode, upon reset, the processor starts executing the application stored in the internal boot kernel.

ADSP-2126x/2136x/2137x/214xx Processor 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.

The **ADSP-2126x/2136x/2137x/214xx Default Boot Kernel Files** table lists the ADSP-2126x/2136x/2137x/214xx boot kernels shipped with CrossCore Embedded Studio.

Table 53. ADSP-2126x/2136x/2137x/214xx Default Boot Kernel Files

Processor	PROM	SPI Slave, SPI Flash, SPI Master, SPI PROM	Link Port Boot (ADSP-2146x)
ADSP-2126x	26x_prom.dxe	26x_spi.dxe	N/A
ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366	36x_prom.dxe	36x_spi.dxe	N/A
ADSP-21367, ADSP-21368, ADSP-21369	369_prom.dxe	369_spi.dxe	N/A
ADSP-21371	371_prom.dxe	371_spi.dxe	N/A
ADSP-21375	375_prom.dxe	375_spi.dxe	N/A
ADSP-21467, ADSP-21469	469_prom.dxe	469_spi.dxe	469_link.dxe
ADSP-21477, ADSP-21478, ADSP-21479	479_prom.dxe	479_spi.dxe	N/A
ADSP-21483, ADSP-21486, ADSP-21487, ADSP-21488, ADSP-21489	489_prom.dxe	489_spi.dxe	N/A

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 <*install_path*>/SHARC/ldr/26x_prom directory of the target processor.

Boot Kernel Modificatio 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 utility is used.

If you do not specify a boot kernel file via the **Loader > General** page of the**Tool Settings** dialog box in the IDE (or via the `-l` command-line switch), the loader utility places a default boot kernel (see the **ADSP-2126x/2136x/2137x/214xx Default Boot Kernel Files** table in *ADSP-2126x/2136x/2137x/214xx Processors Boot Kernels*) 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 (-nokernel)** box (or specify the `-nokernel` command-line switch). The loader utility places no boot kernel in the loader output file.

- To omit a boot kernel. The `-nokernel` switch denotes that a running on the processor (already booted) subroutine imports the `.ldr` file. The loader utility 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 (IVT) 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 utility 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 utility 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 utility 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 (see the **ADSP-2126x/2136x/2137x/214xx Loader Switches** table in *Loader Command-Line Switches for ADSP-2126x/2136x/2137x/214xx Processors*. 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`.

  The **Internal Booting: USER_MESG Block Header Format** listing shows a sample format for the `USER_MESG` header.

  Internal Booting: USER_MESG Block Header Format

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

Rebuilding a Boot Kernel File

If you modify the boot kernel source (.asm) file by inserting appropriate settings for your system, you must rebuild the boot kernel (.dxe) before generating the boot-loadable (.ldr) file. Note the boot kernel source file already contains default register configurations for the external memories (AMI/SDRAM/DDR2).

To Modify a Boot Kernel Source File

1. Copy the applicable boot kernel source file (26x_prom.asm, 26x_spi.asm, 36x_prom.asm, 36x_spi.asm, 369_prom.asm, 369_spi.asm).
2. Apply the appropriate changes.

Note:

Any modification requires that the RTI instruction should still be located at the required peripheral ISR, otherwise the booting may fail.

After modifying the boot kernel source file, rebuild the boot kernel (.dxe) file. Do this from within the IDE (refer to 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 the 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 the 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 utility uses the RESET vector location 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 utility generates a warning if the RESET vector location does not contain NOP or IDLE.

Note:

When creating the loader file, specify the name of the customized boot kernel executable in the **Kernel file (-l)** field on the **CrossCore SHARC Loader > General** page of the **Tool Settings** dialog box.

ADSP-2126x/2136x/2137x/214xx Processor Boot Streams

The loader utility 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/2136x/2137x/214xx 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.

Boot Stream Block Tags

The **ADSP-2126x/2136x/2137x/214xx Processor Block Tags** table details the processor block tags.

Table 54. ADSP-2126x/2136x/2137x/214xx Processor Block Tags

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 (ISA) or Short word (VISA) 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* for details.

Tag	Count 1	Address	Padding
0x4 INIT_L32	Number of 32-bit words	Logical normal word address	None
0x5 INIT_L48 2	Number of 48-bit words	Logical normal word (ISA) or Short word (VISA) address	If count is odd, pad with 48-bit zero word; see *INIT_L48 Blocks* for details.
0x6 INIT_L64	Number of 64-bit words	Logical long word address	None; see *INIT_L64 Blocks* 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
0xB MULTI_PROC for ADSP-21368, ADSP-2146x processors	Processor IDs (bits 0-7); see *Multi-Application (Multi-DXE) Management* for details.	Offset to the next processor ID in words (32 bits)	None
0x0 USR_MESG	msg_word1	msg_word2	None

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

2 40-bit floating point data and 48-bit ISA/VISA instructions words are treated identically.

The ADSP-2126x/2136x/2137x/214xx processor uses eleven block tags, a lesser number of tags compared to other SHARC predecessors.

ZERO_INIT Blocks

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_LDATA blocks of 16-bit, 32-bit, and 64-bit words. The 0x2 tag is used for ZERO_L48 blocks of 40-bit floating point data and 48-bit ISA (VISA instructions ADSP-214xx).

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*). 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. For ADSP-21367/21368/21369/2137x and ADSP-214xx processors, tag `INIT_L32` also is used for all external 32-bit blocks.

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 utility must append one additional 48-bit `NOP` 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 the **INIT_L48 Block Packing and Zero-Padding (ASCII Format)** table.

Table 55. 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

Original Data	Packed into an Even Number of 32-bit Words	-hostwidth		
		32	16	8
			0000	44
			0000	44
				CC
				CC
				BB

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 utility adds one additional word (all zeros) to the end of the block, as shown in the **INIT_L16 Block Packing and Zero-Padding (ASCII Format)** table. The count stored in the header is the actual number of 16-bit words. The count does not include the padded word.

Table 56. 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 the **INIT_L64 Block Packing (ASCII Format)** table.

Table 57. 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

MULT_PROC Blocks

The `0xB` tag is for multiprocessor systems, exclusively supported on ADSP-21368 and ADSP-2146x processors. The tag indicates that the header is a processor ID header with the ID values and offset values stored in the header. A block can have multiple IDs in its block header, which makes it possible to boot the block into multiple processors.

Two data tags, `USER_MESG` and `FINAL_INIT`, differ from the standard format for other SHARC data tags. The `USER_MESG` header is described in *Boot Kernel Modification and Loader Issues*, and the `FINAL_INIT` header in *FINAL_INIT Blocks*.

FINAL_INIT Blocks

The final 256-instructions of the `.ldr` file contain the instructions for the IVT. The instructions are initialized by a special self-modifying subroutine in the boot kernel (see Listing 7-2). To support the self-modifying code, the loader utility modifies the `FINAL_INIT` block as follows:

1. Places a multi-function instruction at the fifth instruction of the block: The loader utility places the instruction `R0=R0-R0, DM(I4,M5)=R9, PM(I12,M13)=R11;` at RESET vector location. 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 utility inserts an RTI instruction (opcode `0x0B3E00000000`) at the first address in the IVT entry associated with the boot-source. Unlike the multifunction instruction placed at RESET vector location which overwrites the data, the loader utility preserves the user-specified instruction which the RTI replaces. This instruction is stored in the header for `FINAL_INIT` as shown in Listing 7-2.
   - For parallel boot mode, the RTI is placed at address `0x80050` for ADSP-2126x processors, at `0x90050` for ADSP-2136x/2137x processors, and at `0x8C050` for ADSP-214xx processors.
   - For all SPI boot modes, the RTI is placed at address `0x80030` for ADSP-2126x processors, at `0x90030` for ADSP-2136x/2137x processors, and at `0x8C030` for ADSP-214xx 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 7-2 illustrates the block header for `FINAL_INIT` if, for example, the opcode `0xAABBCCDDEEFF` is assumed to be the user-intended instruction for the IVT.

FINAL_INIT Block Header Format

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

FINAL_INIT Section (ADSP-2126x)

```
/* ===================== 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 automatically
   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 0x80005 (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 vector
   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!      */
/* ========================================================= */
```

Multi-Application (Multi-DXE) Management

Up to four ADSP-21367/21368/21369/21371/21375, and two ADSP-214xx processors can be clustered together and supported by the CrossCore Embedded Studio loader utility. In PROM boot mode, all of the processors can boot from the same PROM. The loader utility assigns an input executable (`.dxe`) file to a processor ID or to a number of processor IDs, provided a corresponding loader option is selected on the properties page or on the command line.

The loader utility inserts the ID into the output boot stream using the multiprocessor tag `MULTI_PROC` (see the **ADSP-2126x/2136x/2137x/214xx Processor Block Tags** table in *Boot Stream Block Tags*). The loader utility also inserts the offset (the 32-bit word count of the boot stream built from the input executable (`.dxe`) file) into the boot stream. The `MULTI_PROC` tag enables the boot kernel to identify each section of the boot stream with the executable (`.dxe`) file from which that section was built. The **Multiprocessor Boot Stream** figure shows the multiprocessor boot stream structure.

BOOT KERNEL
1st .dxe BLOCK HEADER
1st .dxe DATA BLOCKS
2nd .dxe BLOCK HEADER
2nd .dxe DATA BLOCKS
......
......

Figure 14. Multiprocessor Boot Stream

The processor ID of the corresponding processor is indicated in a 32-bit word, which has the *`Nth`* bit set for the `.dxe` file corresponding to `ID=`*`N`*. The **Multiprocessor ID Fields** table shows possible ID fields.

Table 58. Multiprocessor ID Fields

Processor ID Number	Loader ID Field
0	0x00000001
1	0x00000002
2	0x00000004
3	0x00000008
4	0x00000010
5	0x00000020
6	0x00000040

Processor ID Number	Loader ID Field
7	0x00000080
1 && 4	0x00000012
6 && 7	0x000000C0

The multiprocessor tag, processor ID, and the offset are encapsulated in a multiprocessor header. The multiprocessor header includes three 32-bit words: the multiprocessor tag; the ID (0-7) of the associated processor .dxe file in the lowest byte of a word; and the offset to the next multiprocessor tag. The loader -id#exe=filename switch is used to assign a processor ID number to an executable file. The loader -id#ref=N switch is used to share the same executable file by setting multiple bits in the ID field. The **Multiprocessor Header** figure shows the multiprocessor header structure.

0xB
PROCESSOR ID
OFFSET TO THE NEXT MULTIPROCESSOR HEADER

Figure 15. Multiprocessor Header

ADSP-2126x/2136x/2137x Processor Compression Support

Note:

Compression is not supported on the ADSP-214xx processors.

The loader utility for the ADSP-2126x/2136x/2137x processors offers a loader file (boot stream) compression mechanism known as zLib. The zLib compression is supported by a third party dynamic link library, zLib1.dll. Additional information about the library can be obtained from the *http://www.zlib.net* Web site.

The zLib1 dynamic link library is included with CrossCore Embedded Studio. The library functions perform the boot stream compression and decompression procedures when the appropriate options are selected for the loader utility.

The boot kernel with built-in decompression mechanism must perform the decompression on the compressed boot stream in a booting process. The default boot kernel with decompression functions are included with CrossCore Embedded Studio.

The loader -compression switch directs the loader utility to perform the boot stream compression from the command line. The IDE also offers a dedicated loader properties page (**Compression**) to manage the compression from the graphical user interface.

The loader utility takes two steps to compress a boot stream. First, the utility generates the boot stream in the conventional way (builds data blocks), then applies the compression to the boot stream. The decompression initialization is the reversed process: the loader utility decompresses the compressed stream first, then loads code and data into memory segments in the conventional way.

The loader utility compresses the boot stream on the `.dxe`-by-`.dxe` basis. For each input `.dxe` file, the utility compresses the code and data together, including all code and data from any associated shared memory (`.sm`) files. The loader utility, however, does not compress automatically any data from any associated overlay files. To compress data and code from the overlay file, call the utility with the `-compressionOverlay` switch, either from the properties page or from the command line.

Compressed Streams

The basic structure of a loader file with compressed streams is shown in the **Loader File With Compressed Streams** figure.

KERNEL WITH DECOMPRESSION ENGINE
1st .dxe COMPRESSED STREAM
1st .dxe UNCOMPRESSED STREAM
2nd .dxe COMPRESSED STREAM
2nd .dxe UNCOMPRESSED STREAM
......
......

Figure 16. Loader File With Compressed Streams

The kernel code with the decompression engine is on the top of the loader file. This section is loaded into the processor first and is executed first when a boot process starts. Once the kernel code is executed, the rest of the stream is brought into the processor. The kernel code calls the decompression routine to perform the decompression operation on the stream, and then loads the decompressed stream into the processor's memory in the same manner a conventional kernel does when it encounters a compressed stream.

The **Compressed Block** figure shows the structure of a compressed boot stream.

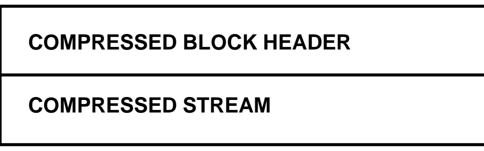


Figure 17. Compressed Block

Compressed Block Headers

A compressed stream always has a header, followed by the payload compressed stream.

The compressed block header is comprised of three 32-bit words. The structure of a compressed block header is shown in the **Compressed Block Header** figure.

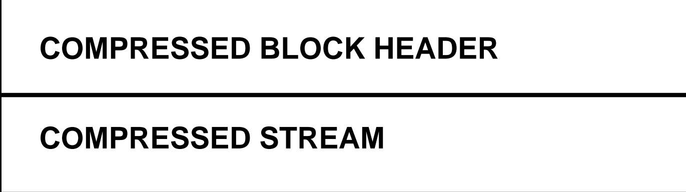


Figure 18. Compressed Block Header

The first 32-bit word of the compressed block header holds the compression flag, `0x00002000`, which indicates that it is a compressed block header.

The second 32-bit word of the compressed block header hold the size of the compression window (takes the upper 16 bits) and padded word count (takes the lower 16 bits). For the ADSP-2126x/2136x/2137x processors, the loader utility always rounds the byte count of the compressed stream to be a multiple of 4. The loader utility also pads 3 bytes to the compressed stream if the byte count of the compressed stream from the loader compression engine is not a multiple of 4. An actual padded byte count is a value between `0x0000` and `0x0003`.

The compression window size is 8-15 bits, with the default value of 9 bits. The compression window size specifies to the compression engine a number of bytes taken from the window during the compression. The window size is the 2's exponential value.

The next 32 bits of the compressed block header holds the value of the compressed stream byte count, excluding the byte padded.

A window size selection affects, more or less, the outcome of the data compression. Streams in decompression windows of different sizes are, in general, different and most likely not compatible to each other. If you are building a custom decompression kernel, ensure the same compression window size is used for both the loader utility and the kernel. In general, a bigger compression window size leads to a smaller outcome stream. However, the benefit of a big window size is marginal in some cases. An outcome of the data compression depends on a number of factors, and a compression window size selection is only one of them. The other important factor is the coding structure of an input stream. A compression window size selection can not cause a much smaller outcome stream if the compression ability of the input stream is low.

Uncompressed Streams

Following the compressed streams, the loader utility file includes the uncompressed streams. The uncompressed streams include application codes, conflicted with the code in the initialization blocks in the processor's memory spaces, and a final block. The uncompressed stream includes only a final block if there is no conflicted code. The

final block can have a zero byte count. The final block indicates the end of the application to the initialization code.

Overlay Compression

The loader utility compresses the code and data from the executable `.dxe` and shared memory `.sm` files when the `-compression` command-line switch is used alone, and leaves the code and data from the overlay (`.ovl`) files uncompressed. The `-compressionOverlay` switch directs the loader utility to compress the code and data from the `.ovl` files, in addition to compressing the code and data from the `.dxe` and `.sm` files.

The `-compressionOverlay` switch must be used in conjunction with `-compression`.

Booting Compressed Streams

The **ADSP-2126x/2136x/2137x Compressed Loader Stream: Booting Sequence** figure shows the booting sequence of a loader file with compressed streams. The loader file is prestored in the flash memory.

1. A a booting process is initialized by the processor.
2. The processor brings the 256 words of the boot kernel from the flash memory to the processor's memory for execution.
3. The decompression engine is brought in.
4. The compressed stream is brought in, then decompressed and loaded into the memory.
5. The uncompressed stream is brought and loaded into memory, possibly to overwrite the memory spaces taken by the compressed code.
6. The final block is brought and loaded into the memory to overwrite the memory spaces taken by the boot kernel.

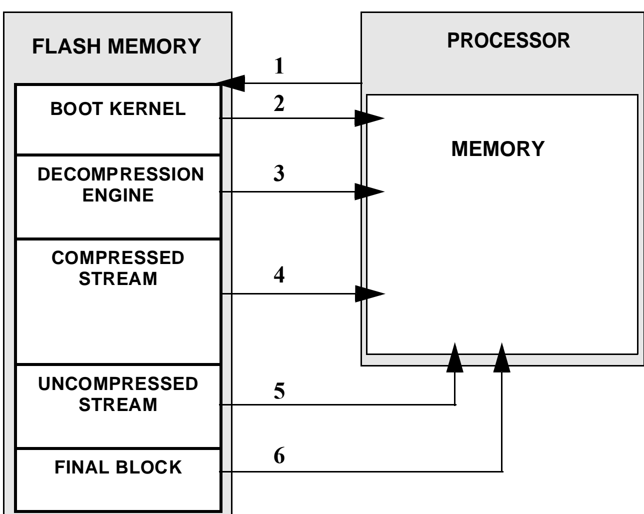


Figure 19. ADSP-2126x/2136x/2137x Compressed Loader Stream: Booting Sequence

Decompression Kernel File

As stated before, a decompression kernel .dxe file must be used when building a loader file with compressed streams. The decompression kernel file has a built-in decompression engine to decompress the compressed streams from the loader file.

A decompression kernel file can be specified from the loader properties page or from the command line via the -l *userkernel* switch. CrossCore Embedded Studio includes the default decompression kernel files, which the loader utility uses if no other kernel file is specified. If building a custom decompression kernel, ensure that you use the same decompression function, and use the same compression window size for both the kernel and the loader utility.

The default decompression kernel files are stored in the <*install_path*>/SHARC/ldr/zlib directory of CrossCore Embedded Studio. The loader utility uses the window size of 9 bits to perform the compression operation. The compression window size can be changed through the loader properties page or the -compressWS # command-line switch. The valid range for the window size is from 8 to 15 bits.

ADSP-2126x/2136x/2137x/214xx Processor Loader Guide

Loader utility operations depend on the loader properties, which control how the utility processes executable files. You select features, such as boot modes, boot kernels, and output file formats via the properties. The properties are specified on the loader utility's command line or the **Tool Settings** dialog box in the IDE (**CrossCore Blackfin Loader** pages). The default loader settings for a selected processor are preset in the IDE.

Note:

The IDE’s **Tool Settings** correspond to switches displayed on the command line.

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

- *Loader Command Line for ADSP-2126x/2136x/2137x/214xx Processors*
- *CCES Loader Interface for ADSP-2126x/2136x/2137x/214xx Processors*

Loader Command Line for ADSP-2126x/2136x/2137x/214xx Processors

The loader utility uses the following command-line syntax for the ADSP-2126x, ADSP-2136x, ADSP-2137x, and ADSP-214xx SHARC processors.

```
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 utility. A list of all switches and their descriptions appear in *Loader Command-Line Switches ADSP- 2126x/ 2136x/2137x/214xx Processors*.

Note:

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 utility 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.

Loader Command-Line Switches for ADSP-2126x/2136x/2137x/214xx Processors

The **ADSP-2126x/2136x/2137x/214xx Loader Switches** table is a summary of the loader switches for the ADSP-2126x, ADSP-2136x, ADSP-2137x, and ADSP-214xx processors.

Table 59. ADSP-2126x/2136x/2137x/214xx Loader Switches

Switch	Description

`-bprom`
`-bspislave|-bspi`
`-bspimaster`
`-bspiprom`
`-bspiflash`
`-blink`
	Specifies the boot mode. The `-b` switch directs the loader utility to prepare a boot-loadable file for the specified boot mode. The valid modes (boot types) are PROM, SPI slave, SPI master, SPI PROM, SPI flash, and link port (ADSP-2146x processors).
	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 utility automatically selects a default boot kernel based on the selected boot type (see *ADSP-2126x/2136x/2137x/214xx Processors Boot Kernels*).
	Note:
	Do not use with the `-nokernel` switch.

`-compression`	Directs the loader utility to compress the application data and code, including all data and code from the application-associated shared memory files (see *ADSP-2126x/2136x/2137x Processors Compression Support*). The data and code from the overlay files are not compressed if this switch is used alone (see `-compressionOverlay`).

Switch	Description
-compressionOverlay	Directs the loader utility to compress the application data and code from the associated overlay files (see *Overlay Compression*). **Note:** This switch must be used with -compression.
-compressWS #	The -compressWS # switch specifies a compression window size in bytes. The number is a 2's exponential value to be used by the compression engine. The valid values are [8-15], with the default of 9.
-fhex -fASCII -fbinary -fbyte -finclude -fs1 -fs2 -fs3	Specifies the format of a boot-loadable file (Intel hex-32, ASCII, binary, byte, 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. • The byte format is used with -splitter only. The byte format is not available for bootable loader files.
-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 the ADSP-21262 processor.
-hostwidth [8|16|32]	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 Motorola S-records formats. • 8, 16, or 32 for ASCII, binary, and include formats. • 8, 16, or 32 for byte format when building with -splitter *section_name*.

Switch	Description
`-id# exe=filename`	Specifies the processor ID. Directs the loader utility to use the processor ID (#) for a corresponding executable file (the `filename` parameter) when producing a boot-loadable file. This switch is used to produce a boot-loadable file to boot multiple processors. Valid values for # are `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`. Do not use this switch for single-processor systems. For single-processor systems, use `filename` as a parameter without a switch. **Note:** This switch is applicable to the ADSP-21367/21368/21369/21371/21375 and ADSP-214xx processors only.
`-id#ref=N`	Directs the loader utility to share the boot stream for processor `N` with processor #. If the executable file of the # processor is identical to the executable of the `N` processor, the switch can be used to set the start address of the processor with ID of # to be the same as that of the processor with ID of `N`. This effectively reduces the size of the loader file by providing a single copy of the file to two or more processors in a multiprocessor system. The ADSP-21367/21378/21369/21371/213755] and ADSP-214xx processors support 8 processors, and the valid processor ids are 0, 1, 2, 3, 4, 5, 6, 7. **Note:** This switch is applicable to the ADSP-21367/21368/21369/21371/21375 and ADSP-214xx processors only.
`-l userkernel`	Directs the loader utility 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 utility searches for a default boot kernel file in the installation directory, (see *ADSP-2126x/2136x/2137x/214xx Processors Boot*

Switch	Description
	Kernels). For kernels with the decompression engine, see *Decompression Kernel File*. **Note:** The loader utility does not search for any kernel file if -nokernel is selected.
-nokernel [*message1, message2*]	Supports internal boot mode. The -nokernel switch directs the loader utility: • 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 *Boot Kernel Modification and Loader Issues*.
-o *filename*	Directs the loader utility 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.
-noZeroBlock	The -noZeroBlock switch directs the loader utility not to build zero blocks.
-p *address*	Specifies the PROM offset start address. This PROM address corresponds to 0x80000 (ADSP-2126x processors) or to external bank MS1 for ADSP-2136x/2137x/214xx processors. The -p switch starts the boot-loadable file at the specified offset address in the EPROM. If the p switch does not appear on the command line, the loader utility starts the EPROM file at offset address 0x0.
-proc *processor*	Specifies the processor. This is a mandatory switch. The *processor* argument is one of the following: ADSP-21261 ADSP-21262 ADSP-21266 ADSP-21362 ADSP-21363 ADSP-21364

Switch	Description
	ADSP-21365 ADSP-21366 ADSP-21367 ADSP-21368 ADSP-21369 ADSP-21371 ADSP-21375 ADSP-21467 ADSP-21469 ADSP-21477 ADSP-21478 ADSP-21479 ADSP-21483 ADSP-21486 ADSP-21487 ADSP-21488 ADSP-21489
`-retainSecondStageKernel`	Directs the loader utility to retain the decompression code in the memory at runtime. **Note:** The `-retainSecondStageKernel` switch must be used with `-compression`.
`-si-revision [none|any|`*`x.x`*`]`	Sets revision for the build, with *`x.x`* being the revision number for the processor hardware. If `-si-revision` is not used, the target is a default revision from the supported revisions.
`-splitter` *`section_name`*	The `-splitter` *`section_name`* switch provides for selectively extracting a section (*`section_name`*) from the `DXE` and writing it to a non-bootable `.ldr` file. The section name is a required argument for `-splitter`. It specifies what section the loader is to extract content from. All other sections are ignored. **Note:** This switch is provided for the ADSP-214xx processors only. The `-splitter` *`section_name`* provides support for SW (VISA) sections or NW (normal-word).
`-v`	Outputs verbose loader messages and status information as the loader utility processes files.
`-version`	Directs the loader utility 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.

Switch	Description
`-u` *`value`*	Specify a *`value`* for the content of the user flag field in a BYTE format header. The *`value`* range is `0x0-0xFF`. If no `-u` switch is specified, the user flag field is zero. **Note:** Use with non-bootable files built with the `-fbyte` and `-splitter` *`section_name`* switches only.

CCES Loader Interface for ADSP-2126x/2136x/2137x/214xx Processors

Once a project is created in the CrossCore Embedded Studio IDE, you can change the project's output (artifact) type.

The IDE invokes the `elfloader.exe` utility to build the output loader file. To modify the default loader properties, use the project's **Tool Settings** dialog box. The controls on the pages correspond to the loader command-line switches and parameters (see *Loader Command-Line Switches ADSP- 2126x/2136x/2137x/214xx Processors*). The loader utility for the ADSP-214xx SHARC processors also acts as a ROM splitter when evoked with the corresponding switches.

The loader pages (also called *loader properties pages*) show the default loader settings for the project's target processor. Refer to the CCES online help for information about the loader interface.

The CCES splitter interface for the ADSP-2126x/2136x/2137x processors is documented in the Splitter for SHARC Processors chapter.

10

Splitter for SHARC Processors

This chapter explains how the splitter utility (`elfspl21k.exe`) is used to convert executable (`.dxe`) files into non-bootable files for the ADSP-21xxx SHARC processors. Non-bootable PROM image files execute from external memory of a processor. For SHARC processors, the utility creates a 64-/48-/40-/32-bit image file or an image file to match a physical memory size.

Attention:

Users who are migrating from VisualDSP++

VisualDSP++ legacy splitter projects cannot be imported into the CrossCore Embedded Studio IDE. There is no SHARC splitter build artifact in the IDE. If attempting to import a VisualDSP++ legacy splitter project, a status of "Not Converted" appears along with the following error messages:

The legacy SHARC splitter `elfspl21k.exe` is available with CrossCore Embedded Studio for command-line usage.

Splitter functionality for SHARC processors, beginning with the ADSP-214xx family, is available through the SHARC loader instead of the legacy splitter utility.

For SHARC processors, the splitter utility also properly packs the external memory data or code to match the specified external memory widths if the logical width of the data or code is different from that of the physical memory.

In most instances, developers working with SHARC processors use the loader utility instead of the splitter. One of the exceptions is a SHARC system that can execute instructions from external memory. Refer to the Introduction chapter for the splitter utility overview; the introductory material applies to both processor families.

Splitter Command Line

Use the following syntax for the SHARC splitter command line.

```
elfspl21k [-switch] -pm &|-dm &|-64 &| -proc part_number inputfile
```

or

```
 elfspl21k [-switch] -s section_name 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` 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 for the splitter utility. Switches may be used in any order. A list of the splitter switches and their descriptions can be found in *Splitter Command-Line Switches*.
- `-pm &| -dm &| -64` - For SHARC processors, the `&|` symbol between the switches indicates `AND/OR`. The splitter command line must include one or more of `-pm`, `-dm`, or `-64` (or the `-s` switch). The `-64` switch corresponds to `DATA64` memory space.
- `-s section_name` - 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.

Note:

Most items in the splitter command line are not case sensitive; for example, `-pm` and `-PM` are interchangeable. However, the names of memory sections must be identical, including case, to the names used in the executable.

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
```

runs the splitter utility 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.
	Because these are the only switches used to identify the memory source, the specified sources are `PM`, `DM`, or `DATA64` memory sections. Because no other content switches appear on these command lines, the output file format defaults to a Motorola 32-bit format, and the PROM word width of the output defaults to 8 bits for all PROMs.

`-o pm_stuff`
`-o dm_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_#`. The splitter utility adds the `.s_#` file extension to the output files; # is a number that differentiates one output file from another.

`my_proj.dxe`	Specifies the name of the input (`.dxe`) file to be processed into non-bootable PROM image files.

Splitter File Searches

File searches are important in the splitter process. The splitter utility supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as described in **Loader File Searches** in the Introduction chapter.

Splitter Output File Extensions

The splitter utility follows the conventions shown in the **Splitter Output File Extensions** table for output file extensions.

Table 60. Splitter 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 **Loader Output Files in Motorola S-Record Format** in the File Formats appendix.
`.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** in the File Formats appendix.
`.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** in the File Formats appendix.

Splitter Command-Line Switches

A list of the splitter switches appears in the **Splitter Command-Line Switches** table.

Table 61. Splitter Command-Line Switches

Item	Description
`-64`	The `-64` (include `DATA64` memory) switch directs the splitter utility to extract all sections declared as 64-bit memory sections from the input `.dxe` file. The 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 utility 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.

`-f h`
`-f s1`
`-f s2`
`-f s3`
`-f b`
	The `-f` (PROM file format) switch directs the splitter utility 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** in the File Formats appendix.

`-norom`	The `-norom` (no ROM in PROM) switch directs the splitter utility 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.
`-o` *`imagefile`*	The `-o` (output file) switch directs the splitter utility 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.
`-pm`	The `-pm` (include program memory) switch directs the splitter utility to extract memory sections declared program memory ROM from the input `.dxe` file. The `-pm` switch influences the

Item	Description
	operation of the -ram and -norom switches, adding program memory as the target.
-proc *part_number*	Specifies the processor type to the splitter utility. Valid processors are: • ADSP-21160, ADSP-21161 • ADSP-21261, ADSP-21262, ADSP-21266 • ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368, ADSP-21369, • ADSP-21371, ADSP-21375
-r#[#]	The -r (PROM widths) switch specifies the number of PROM files and their width in bits. The splitter utility 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` 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 utility 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.
-s *section_name*	The -s (include memory section) switch directs the splitter utility to extract the content of one memory section (*section_name*) from the executable. The *section_name* argument is case sensitive and must exactly match the name as it appears in the LDF for the executable.

Item	Description
	You must also specify the switch `-dm` or `-pm` or `-64` for the memory type. Rerun the splitter for any additional sections that are required, changing the memory type switch and output file as needed for each invocation. **Note:** Short-word sections are not supported in the legacy SHARC splitter. To split a SW section into a raw (non-bootable) format, use the new `-splitter` *section_name* switch in the SHARC ADSP-214xx loader.
`-si-revision [none|any|`*x.x*`]`	Sets revision for the build, with *x.x* being the revision number for the processor hardware. If `-si-revision` is not used, the target is a default revision from the supported revisions.
`-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 utility 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 utility interprets the number as hexadecimal. For more information, see **Splitter Output Files in Byte-Stacked Format** in the File Formats appendix.
`-version`	Directs the splitter utility to show its version information.

11

File Formats

CrossCore Embedded Studio supports 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*
- *Build Files*
- *Debugger Files*

Most of the development tools use industry-standard file formats. These formats are described in their respective format specifications.

Source Files

This section describes the following source (input) file formats.

- *C/C++ Source Files*
- *Assembly Source Files*
- *Assembly Initialization Data Files*
- *Header Files*
- *Linker Description Files*
- *Linker Command-Line Files*

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[10] 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 *C/C++ Compiler and Library Manual*.

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 *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 the **Numeric Formats** table to distinguish between numeric formats.

Table 62. Numeric Formats

Convention	Description
`0x`*number* `H#`*number* `h#`*number*	Hexadecimal number
number `D#`*number*	Decimal number

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

Convention	Description
d#*number*	
B#*number* b#*number*	Binary number
O#*number* o#*number*	Octal number

For all numeric bases, the assembler uses words of different sizes for data storage. The word size varies by the processor family.

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 *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 *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 *Linker and Utilities Manual.*

Build Files

Build files are produced by CrossCore Embedded Studio while building a project. This section describes the following build file formats.

- *Assembler Object Files*
- *Library Files*
- *Linker Output Files*
- *Memory Map Files*
- *Bootable Loader Output Files*
- *Non-Bootable Loader Output Files in Byte Format*
- *Splitter Output Files*

Assembler Object Files

Assembler output object files (`.doj`) are binary object and linkable files (ELF). Object files contain relocatable code and debugging information for a DSP program's memory segments. The linker processes object files into an executable file (`.dxe`). For information on the object file's ELF format, see the TIS Committee specification.

Library Files

Library files (`.dlb`), the output of the archiver, are binary, object 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 the ELF specification.

Note:

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 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 specification.

The loaders/splitter utilities 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 splitter utility. 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 utility 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.

Bootable Loader Output Files

- *Loader Output Files in Intel Hex-32 Format*
- *Loader Output Files in Include Format*
- *Loader Output Files in Binary Format*
- *Loader Output Files in Motorola S-Record Format*

Loader Output Files in Intel Hex-32 Format

The loader utility 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 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.

The **Extended Linear Address Record Example** table shows an extended linear address record.

Table 63. 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

The **Data Record Example** table shows the organization of a data record.

Table 64. 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

The **End-of-File Record Example** table shows an end-of-file record.

Table 65. 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

CrossCore Embedded Studio includes a utility program to convert an Intel hexadecimal file to Motorola S-record or data file. Refer to **hexutil - Hex-32 to S-Record File Converter** in the Utilities appendix for details.

Loader Output Files in Include Format

The loader utility can output include format files (`.ldr`). These files permit the inclusion of the loader file in a C program.

The word width (8- or16-bit) of the loader file depends on the specified boot type. Similar to Intel hex-32 output, the loader output in include format have some basic parts in the following order.

1. Initialization code (some Blackfin processors)
2. Boot kernel (some Blackfin and SHARC processors)
3. User application code
4. Saved user code in conflict with the initialization code (some Blackfin processors)
5. Saved user code in conflict with the kernel code (some Blackfin and SHARC processors)

The initialization code is an optional first part for some Blackfin processors, while the kernel code is the part for some Blackfin and SHARC processors. 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 (on SHARC processors). 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[0];
```

The `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 utility can output binary format files (`.ldr`) to support a variety of PROM and microcontroller storage applications.

Binary format files use less space than other loader file formats. Binary files have the same contents as the corresponding ASCII file, but in binary format.

Loader Output Files in Motorola S-Record Format

The loader and splitter utilities can output Motorola S-record format files (.s_#), which conform to the Intel standard. The three file formats supported by the loader and PROM splitter utilities 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)

The **Header Record Example** table shows the organization of a header record.

Table 66. Header Record Example

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

The **S1 Data Record Example** table shows the organization of an S1 data record.

Table 67. S1 Data Record Example

Field	Purpose
S10D00043C4034343426142226084C	Example record
S1	Record type

Field	Purpose
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. The **S1 Termination Record Example** table shows the organization of an S1 termination record.

Table 68. S1 Termination Record Example

Field	Purpose
S903000DEF	Example record
S9	Start character
03	Byte count of this record
000D	Address
EF	Checksum

The S2 termination record has the same format, except that the start character is S8 and the address field is six characters wide.

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 - Hex-32 to S-Record File Converter** in the Utilities appendix.

Non-Bootable Loader Output Files in Byte Format

The loader utility can output non-bootable loader files (`.ldr`) in byte format. This format is only available when the `-splitter` *`section-name`* switch is used.

The non-bootable file in BYTE format has these characteristics:

- A one-line header
- A block of one or more lines of section data from the `.dxe` file
- A zero header that signals the end of the file

The **Byte Format File Example** table shows a sample byte-format file created by the loader utility.

Table 69. Byte Format File Example

Field	Purpose
`200688AB0012435D00000768`	Example header record (the first line of file)
`20`	Width of address and length fields (in bits) Addresses are 32-bit width.
`06`	Reserved field in use by ADI for versioning. The loader is currently setting this to Version 6. **Note:** The `elfspl21k` utility is currently setting this to Version 5 for `.stf` files).
`88`	Flags (88 = SW, 80 = PM, 00 = DM)This shows a build with `-splitter` *`section_name`* that is a SW section
`AB`	User-defined flags (loaded with `-u` *`value`* switch). This build shows the result of a build with `-u 0xAB`. If no `-u` switch is present, the user-defined flag field is `00`.
`0012435D`	Start address of the data block
`00000768`	Number of bytes of data that follow
`0f14000b20010fb40000`	Lines of section data. The `-hostwidth [8|16|32]` switch determines the number of bytes per line. This example shows the content from a SW section for a build using `-hostwidth 16`.

Field	Purpose
0000000000000000000000000	Example header record (signals the end of file)

To produce a byte-formatted file in the CCES IDE:

1. Open the **Properties** dialog box for the project.
2. Choose **C/C++ Build > Settings**. The **Tool Settings** page appears.
3. Click **Additional Options** under **CrossCore SHARC Loader**. The loader **Additional Options** properties page appears.
4. Click **Add** (+). The **Enter Value** dialog box appears.
5. In **Additional Options**, type in `-splitter my_sw_section -fBYTE -u 0xAB`.
6. Click **OK** to close the dialog box.
7. Click **Apply**.

For information about the byte-stacked format (`.stf`) files produced by the legacy `elfspl21k.exe` utility, see *Splitter Output Files in Byte-Stacked Format*.

Splitter Output Files

- *Splitter Output Files in Intel Hex-32 Format*
- *Splitter Output Files in Byte-Stacked Format*
- *Splitter Output Files in ASCII Format*
- *Splitter Output Files in Motorola S-Record Format*

Splitter Output Files in Intel Hex-32 Format

The splitter utility 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*.

The splitter utility 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 utility 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.

The **Header Record in Byte-Stacked Format Example** table shows a header record in byte-stacked format.

Table 70. Header Record in Byte-Stacked Format Example

Field	Purpose
20008000000000080000001E	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 non- zero.

A block of data records follows its header record, five bytes per line for data memory, and six byte per line for program memory or in other physical memory width. For example:

Program Memory Section (Code or Data)

```
3C4034343426
```

```
142226083C15
```

Data Memory Section

```
3C40343434
```

```
2614222608
```

DATA64 Memory Section

```
1122334455667788
```

```
99AABBCCDDEEFF00
```

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 splitter utility is invoked as a splitter utility, 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.

Data Memory (DM) Example:

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

The above DM section results in the following code.

```
00010000      /* 32-bit logical address field */
00000004      /* 32-bit logical length field */
00020201      /* 32-bit control word: 2x address multiply */
              /* 02 bytes logical width, 01 byte physical width */
00000000      /* reserved */
0x12          /* 1st data word, DM data is 8 bits */
0x56
0x9A
0xDE          /* 4th (last) data word */
CRC16         /* optional, controlled by the -checksum switch */
```

Splitter Output Files in Motorola S-Record Format

The splitter utility can output Motorola S-record format files (.s_#). See *Loader Output Files in Motorola S-Record Format* for more information.

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 a 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 (the **SPORT Data File Example** table), 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 71. 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

12

Utilities

CrossCore Embedded Studio includes several utility programs, some of which run from a command line only.

This appendix describes the following utilities.

- *hexutil – Hex-32 to S-Record File Converter*
- *elf2dyn – ELF to Dynamically-Loadable Module Converter*
- *elf2elf – ELF to ELF File Converter*
- *dyndump – Display the Contents of Dynamically-Loadable Modules*
- *dynreloc – Relocate Dynamically-Loadable Modules*
- *signtool – Sign and Encrypt boot streams for secure booting*

Other CrossCore Embedded Studio utilities, for example, the ELF file dumper, are described in the *Linker and Utilities Manual* (search the online help).

Attention:

VisualDSP++ executables are not upwardly compatible with CrossCore Embedded Studio executables. The ELF format has changed.

hexutil - Hex-32 to S-Record File Converter

The hex-to-S file converter (`hexutil.exe`) transforms a loader (`.ldr`) file in Intel hexadecimal 32-bit format to Motorola S-record format or produces an unformatted data file.

Syntax

`hexutil.exe` is a command-line utility. It has the following syntax:

```
hexutil input_file [-s1|s2|s3|StripHex] [-o file_name]
```

where:

`input_file` is the name of the `.ldr` file generated by the CrossCore Embedded Studio splitter utility.

The **hexutil Command-Line Switches** table lists the optional switches used with the `hexutil` command.

Table 72. hexutil Command-Line Switches

Switch	Description
`-s1`	Specifies Motorola output format S1.
`-s2`	Specifies Motorola output format S2.
`-s3`	Specifies the default output format - Motorola S3. That is, when no switch appears on the command lines, the output file format defaults to S3.
`-StripHex`	Generates an unformatted data file.
`-o`	Names the output file; in the absence of the `-o` switch, causes the output file name to default to `input_file.s`.

The Intel hex-32 and Motorola S-record file formats are described in the File Formats chapter.

elf2dyn - ELF to Dynamically-Loadable Module Converter

The ELF to dynamically-loadable module converter (`elf2dyn.exe`) accepts an executable (`.dxe`) file produced by the CrossCore Embedded Studio linker and converts the file from ELF (Executable and Linkable Format) to a "lighter-weight" format suitable for loading into an existing application at runtime. In short, `elf2dyn` produces *Dynamically-Loadable Modules* (DLMs).

By default, a `.dxe` file converted to DLM format cannot be used immediately. Some extra steps must be taken, which typically involves producing a specialized `.ldf` file. For details, refer to the *System Runtime Documentation* section of the help system.

Dynamically-Loadable Modules

A dynamically-loadable module - a DLM - is intended to be loaded by another, `main' application at runtime. The DLM can contain executable code, data, or both.

DLM files have the following characteristics:

- DLMs are not standalone applications. A DLM does not set up the stack, configure the processor, etc. All such initialization are done by the main application prior to loading the DLM.
- DLMs are self-contained. Although a DLM can consist of multiple elements of code and/or data, which can reference each other, the DLM cannot make symbolic references to code elements outside of the DLM.
- DLMs are not yet relocated. This means that when the linker produced the `.dxe`, from which the DLM has been derived, the linker was instructed not to resolve cross-references within the DLM to final, absolute addresses. Instead, all internal references still are expressed in relative terms. After loading the DLM, a process known as *relocation* must be carried out before the DLM's contents can be used.

- DLMs export one or more symbols, which can be used by the main application to reference the contents of the DLM.

Note:

Because of the noted differences, most `.dxe` files are not suitable for conversion to DLMs.

For information on how to construct `.dxe` files that can be converted to DLMs, load DLMs, and access DLM contents, refer to the *System Runtime Documentation* section in help.

Syntax

The `elf2dyn.exe` utility is invoked from the command-line or CCES IDE, as a post-build step specified in your project's **Settings > Build Steps** dialog box.

The command-line syntax for the utility is:

```
elf2dyn [-h|l|r|v] [-a name=num] [-e sym] [-S|R|W|E errnum] [-o outfile] elfinfile
```

where:

`elfinfile` is the name of the `.dxe` file in ELF format to convert into a DLM file for Blackfin or SHARC processors. The file name must appear at the end of the command line. In order to be useful as a DLM, it must contain relocations, as described in the *System Runtime Documentation* help section.

The **elf2dyn Command-Line Switches** table lists the optional switches used with the `elf2dyn` command. Switches must appear before *`elfinfile`* on the command line.

Table 73. elf2dyn Command-Line Switches

Switch	Description
`-h`	Displays a list of accepted switches.
`-l`	Creates a `lite' format output file; see *File Formats and the-l Switch*.
`-r`	Emits remarks (if any apply).
`-v`	Emits version information.
`-a` *`name=num`*	Forces section *`name`* to have alignment *`num`*; see *Section Alignment*.
`-e` *`symbol`*	Exports *`symbol`* so that it can be referenced by the main application; see *Exported Symbols*.
`-o` *`outfile`*	Use *`outfile`* as the name of the DLM to create. The switch specifies where `elf2dyn` creates the DLM file. If `-o` is omitted, `elf2flt` creates `elfinfile.dyn`,

Switch	Description
	which means that the output file has the same name as the input file with the `.dyn` suffix appended.
`-S` *errnum*	Suppresses diagnostic *errnum*.
`-R` *errnum*	Makes diagnostic *errnum* a remark.
`-W` *errnum*	Makes diagnostic *errnum* a warning.
`-E` *errnum*	Makes diagnostic *errnum* an error.

File Formats and -l Switch

`elf2dyn` supports two formats, a **full** format and a **lite** format. By default, DLMs are created using the full format, the lite format can be enabled by specifying the `-l` switch.

Both ELF files and DLM files are divided into sections, some of which specify content which makes up the code and/or data of the application, and some of which specify metadata about the code and/or data of the application. For example, the debug information of an application is used by a debugger, which controls the processor, but not downloaded to the processor by the debugger. For the purposes of this discussion, the following sections use the terms **application section** and *metadata* section to make this distinction.

The lite format is closely modeled on the bFLT file format, which in turn is influenced by the UNIX `a.out` file format. It imposes the following constraints on the `.dxe` input file:

1. The ELF file must contain exactly three application sections with the following names and uses:

Note:

The lite format is supported for Blackfin processors only because of the mentioned constraints.

   - `.text`, which contains only executable code
   - `.data`, which contains only initialized data (initialized to any value)
   - `.bss`, which contains only zero-initialized data
2. All sections must contain 8-bit byte-addressed values.
3. Exported symbols are impossible, so the code entry point must be the first instruction at the start of the `.text` section, and any data entry point must be the first location at the start of the `.data` section.

In contrast, the full DLM format supports the following:

- The ELF file can have any number of application sections, from 1 to 255.
- Application sections can have any valid section name.[11]

[11] If exported symbols are being used, two names are reserved: `.expsym` and `.expstr`.

- Application sections can use any word size that is valid for the target processor. Note that not all sections need be the same word size.
- Symbols may be exported, so that entry points need not be at the start of a particular section.

Note:

The full DLM format is proprietary to Analog Devices, Inc. Although more complex than the bFLT-inspired lite format, it is still lighter than the full ELF standard.

Exported Symbols

The `-e` switch specifies particular symbols for exporting. The symbol table of the ELF file is a metadata section, so it is not available to the application when the application runs. When a symbol is exported by `elf2dyn`, the symbol is placed into an exported symbol table, which is in an application rather than a metadata section. This means that the table is available to the running application, and the application can locate DLM contents by searching for the symbol in the table.

The exported symbol table is two new application sections that are added to the DLM by `elf2dyn` when the `-e` switch is being used:

- `.expstr` is the **exported string table**; it contains `NULL`-terminated strings which are the exported symbol names. These strings are in the native target processor data format.
- `.expsym` is the **exported symbol table**; it contains one or more records, where each record consists of two pointers - to the symbol's name in the exported string table, and to the symbol's location in memory, respectively. Because the record contains pointers, the exported symbol table is only accessible once the DLM has been relocated.

The `-e` switch accepts names exactly as they appear in the symbol table of the ELF input file *`elfinfile`*. No name-demangling or parsing is done by elf2dyn. This means that, for example, to export your `main()` function in your DLM, -e _main is used. If you are unsure how to specify a symbol's name for exporting, use the following command:

```
elfdump -n .symtab elfinfile
```

The `elfdump` utility displays the contents of the ELF file's symbol table; `elfdump` shows both mangled and demangled names. For more information, see *dyndump – Display the Contents of Dynamically-Loadable Modules*.

After a DLM is loaded and resolved, you can map an exported symbol's name into its address using `dyn_LookupByName()`. Refer to the *System Runtime Documentation* help section for details.

Section Alignment

By default, the `elf2elf` utility creates a DLM in which each section has the same alignment constraints as the original sections in the input ELF file. You can specify a stronger alignment for a section by using the `-a` switch. The `-a` switch requires a parameter of the form *`name=number`*, where:

- *`name`* is the name of the section in the input ELF file
- *`number`* is a positive power-of-two decimal integer.

Multiple instances of the `-a` switch are permitted on the same command-line to specify stronger alignment for more than one section.

It is not an error if the input ELF file does not contain a section called `name`. If the input file does not have a section with a matching name, the switch is ignored.

elf2elf - ELF to ELF File Converter

The ELF to ELF file converter (`elf2elf.exe`) is a command-line utility for upgrading executables built using VisualDSP++ 5.0 to the new CrossCore Embedded Studio ELF format.

Attention:

VisualDSP++ executables are not upwardly compatible to CrossCore Embedded Studio executables. The ELF format has changed.

The loaders and splitters take executable files in ELF format as input. These are files with the suffixes `dxe`, `ovl`, or `sm`. The CrossCore Embedded Studio loaders and splitters expect input in the CrossCore Embedded Studio ELF format, which has significant differences from the VisualDSP++ ELF format.

You do not need to use `elf2elf` when:

- Creating new projects in the CrossCore Embedded Studio IDE
- Importing VisualDSP++ legacy projects into the CrossCore Embedded Studio IDE and rebuilding all code from source

In both cases, CrossCore Embedded Studio creates the executables in the expected ELF format.

The following unrecoverable error is reported by the CrossCore Embedded Studio loaders and splitters if any `dxe`/`ovl`/`sm` in the build is one built with VisualDSP++:

```
[Error  ld0002]
             File in legacy ELF format created with VisualDSP++ 5.0 or earlier.
             Rebuild from source or upgrade using the elf2elf utility: <filename>
```

If you do not have sources to rebuild your code, the `elf2elf` utility is available.

Syntax

`elf2elf.exe` has the following syntax:

```
  elf2elf [switches] [infile]
```

where:

`infile` is the name of the ELF input file produced by CrossCore Embedded Studio.

The **elf2elf Command-Line Switches** table shows the optional switches used with the `elf2elf` command.

Table 74. elf2elf Command-Line Switches

Switch	Description
-o *file*	Produces the output file with a name given by *file*.
-keep	Prevents any temporary files that have been created by the ELF Conversion Tool from being deleted.
-merge	Merge the contents of the .ovl and .sm files specified in the input .dxe into the output .dxe.
-version	Displays the version number of the ELF conversion utility.

Example: `elf2elf.exe oldKernel.dxe -o newKernel.dxe`

Note:

If you are building loader files, do no use -merge; instead, do the following:

- Upgrade the .dxe file without -merge
- Upgrade each .sm or .ovl file with elf2elf

dyndump - Display the Contents of Dynamically-Loadable Modules

To display the contents of dynamically-loadable modules (DLMs) produced by elf2dyn.exe, use dyndump.exe.

Refer to *elf2dyn – ELF to Dynamically-Loadable Module Converter* for more information on elf2dyn.

Syntax

dyndump.exe is a command-line utility. It has the following syntax:

```
dyndump [-f family] dlmname
```

where:

dlmname is the name of a file generated by elf2dyn; see *elf2dyn – ELF to Dynamically-Loadable Module Converter*.

The **dyndump Command-Line Switches** table lists switches used with the dyndump command.

Table 75. dyndump Command-Line Switches

Switch	Description
-f *Family*	Select the target processor family; see *-f family*.

Switch	Description
dlmname	The file to be displayed; see *Output*.

-f Family

The dyndump utility can display files for both Blackfin and SHARC processors in full and lite DLM format. The lite format files do not include any information indicating their target processor; without this information, the dyndump utility cannot decode the target-specific relocations section. See *File Formats and the-l Switch* for more information.

The valid forms of the -f family switch are:

- -f blackfin
- -f sharc

Note:

Full-format DLM files do specify their target processor, so the -f switch is not necessary for displaying such files.

Output

The dyndump utility displays the header from the input file, followed by the metadata sections and application sections, in the order they appear in the file. In general, dyndump displays information in several ways, providing:

- The offset within a section in the native addressing of that section
- The byte offset from the start of the file
- A hexadecimal dump of the bytes in the order they appear within the file. Each target address location in an application is displayed on a separate line, so byte-addressed sections shows a single byte per line, while 64-bit addressed sections shows eight bytes per line.
- An ASCII representation of the bytes in the order they appear within the file.

File headers, section headers (where present) and relocations are displayed more explicitly.

dynreloc - Relocate Dynamically-Loadable Modules

To relocate DLMs produced by elf2dyn.exe, use dynreloc.exe.

Refer to *elf2dyn – ELF to Dynamically-Loadable Module Converter* for more information about elf2dyn.

The dynreloc utility reads a DLM file as input, performs relocations according to address mappings provided through command-line switches, and writes out a new DLM file. The new DLM file contains no relocations and can be loaded only into the locations specified to the dynreloc utility.

Syntax dynreloc.exe is a command-line utility. It has the following syntax:

```
dynreloc [-h|v] [-a sec=addr] [-m a:n:w:m][-o outfile] infile
```

where:

infile is the name of the DLM file to relocate.

The **dynreloc Command-Line Switches** table lists the optional switches used with the `dynreloc` command. Switches must appear before *infile* on the command line.

Table 76. dynreloc Command-Line Switches

Switch	Description
`-h`	Displays a list of accepted switches.
`-v`	Displays version information.
`-a` *sec=addr*	Forces section *sec* to start at address *addr*; see *Explicit Mapping*.
`-m` *a:n:w:m*	Specifies a memory range that can be used for section mapping; see *Region Mappings*.
`-o` *outfile*	Use *outfile* as the name of the file to write the relocated DLM.

Explicit Mappings

The `-a` switch specifies explicit section mappings to the dynreloc utility.

The `-a` switch requires a parameter, of the form *sec=addr*. The *sec* argument identifies the section in the input DLM, while *addr* gives the address to which the section is to be mapped in the output DLM. It is your responsibility to ensure that:

- Starting address *addr* is appropriately aligned for section *sec*.
- Starting address *addr* is in an appropriate memory space for section *sec*.
- Starting address *addr* points to a free area of memory that has sufficient space for the contents of section *sec*.

There may be multiple instances of the `-a` switch to specify mappings for more than one section. A given section may be named by only one `-a` switch.

Region Mappings

The `-m` switch specifies regions of memory which can be used for mapping sections that have not otherwise been explicitly mapped through the `-a` switch.

The `-m` switch requires a parameter of the form *a:n:w:m*. The parameter's fields have the following meanings:

Field	Description
a	Starting address of the memory region.
n	Number of addressable locations in the region.
w	Width of each addressable location (in bits).
m	Alignment; allocations from the region will be in multiples of this value.

When relocating a section `sec` of `sz` locations from the DLM, the `dynreloc` utility uses the following approach:

1. The `dynreloc` utility looks for an explicit mapping for the section `sec`, as specified by the `-a` switch. If such a mapping is found, the mapping is used.
2. If no explicit mapping is found, the `dynreloc` utility searches for a memory region that has the same memory width as section `sec`. If such a memory region is found, the `dynreloc` utility claims `sz` locations from the region and maps `sec` to the first of those locations. The remainder of the region is available for other sections.
3. If no region has been specified with the same width as `sec`, or if the region has insufficient unallocated locations to accommodate the `sz` locations required by `sec`, the `dynreloc` utility reports an error.

Note:

Only one region may be specified for each memory width.

Region mappings are a more convenient method of specifying mappings because the `dynreloc` utility ensures that sections allocated from the same region do not overlap. This is not the case with `-a` mappings, which are used without any validation.

signtool - Sign and Encrypt Boot Streams for Secure Booting

The signing utility (`signtool.exe`) supports secure booting by cryptographically signing boot stream files. It can also encrypt boot streams and provides public-key management support. The input to signtool is the boot stream file to be protected, in binary form. The output is the signed (and optionally encrypted) boot stream, ready for secure booting.

Note that `signtool` treats its input as raw, binary data, and performs no interpretation of the content. Ensure that the `-f binary` switch is specified when invoking the `elfloader` utility; otherwise, the resulting secure-boot image is not be usable.

The `signtool` utility is provided by a third party; not all functionality is used by CrossCore Embedded Studio or applicable to Analog Devices processors.

Syntax

Invoke the `signtool` utility from the command line or the CrossCore Embedded Studio IDE, as a post-build step specified in your project's **Settings** > **Build Steps**.

The command-line syntax for the `signtool` utility is:

```
signtool cmd switches
```

cmd indicates the kind of operation to perform (the choices are described in the following sections). *switches* are dependent on the choice of *cmd*.

Output Formats

When signing or encrypting a file, `signtool` generates output in one of several forms, depending on whether encryption is required, and delivers the encryption key to the processor for decryption. The different output formats are known by the abbreviations `BLp`, `BLx`, `BLw`, and `BLe`.

- The `BLp` format supports signing, not encryption. The output file includes a signature produced by a 224-bit Elliptic Curve Digital Signature Algorithm (ECDSA) private key. The corresponding public key must be separately programmed into the processor's OTP to allow the target processor to authenticate the boot stream.
- The `BLx` format supports signing and encryption. The output file is signed, as per the `BLp` format. It is also encrypted using an AES-128 key. The encryption key is not part of the output file, so it must be programmed into the processor's OTP to allow the target processor to decrypt the boot stream.
- The `BLw` format supports signing and encryption. The output file is signed, as per the `BLp` format. It is also encrypted using an AES-128 key as per the `BLx` format, but unlike `BLx`, the encryption key is also included in the output file, itself encrypted by a second AES-128 key (the "wrap" key). The wrap key must be programmed into the processor's OTP to allow the target processor to decrypt the encryption key, which then allows the processor to decrypt the boot stream.
- The `BLe` format is only intended for testing, and should not be used for production systems. It supports authentication and encryption, but the encryption key is included in the output file as-is, without any protection.

Key Generation for Signing

Boot stream files are signed using the private key from a 224-bit Elliptic Curve Digital Signature Algorithm (ECDSA) keypair, stored in DER format. The `signtool` utility's own `genkeypair` command can be used to create such a keypair. The following table describes the switches available for the `genkeypair` command.

The following is an example command for creating a keypair:

```
signtool genkeypair -algo ecdsa224 -outfile keychain.der
```

Note:

Ensure you keep your keypair safe, once you have created it, and keep it secret, as the private key therein is essential for authenticity.

Switch	Description

Key Generation for Encryption

Boot stream files are encrypted using 128-bit keys, via the AES-128 algorithm. This is a symmetric algorithm, meaning that the same key is used for encryption and decryption. The `signtool` utility does not provide built-in support for generating AES-128 keys; you can create any 16-character file and use that as your AES-128 key. Ensure that you keep the key secret.

Signing and Encrypting Boot Streams

Boot streams are signed using the private key from a 224-bitElliptic Curve Digital Signature Algorithm (ECDSA) keypair, by invoking the `sign` command of the `signtool` utility. The following table lists the switches that are relevant to the `sign` command.

Switch	Description

The following is an example of signing a boot stream file:

```
signtool sign -type BLp -prikey keychain.der -infile boot.bin -outfile secboot.bin
```

The following is an example of signing and encrypting a boot stream file, where a random encryption key is generated by `signtool`:

```
signtool sign -type BLw -prikey keychain.def -wrapkey aes.bin \\
          -infile boot.bin -outfile secboot.bin
```

Extracting Public Keys

When a processor boots using secure boot, it must authenticate the boot stream before processing it. The boot stream will have been signed using the private key from a keypair; the authentication is done using the corresponding public key from the same keypair. Use `signtool`'s `getkey` command to extract the public key so that you can store it in OTP on the processor, where it can be used during secure boot.

The switches relevant to the `getkey` command are listed in the table below.

The following is an example of extracting the public key in binary form:

```
signtool getkey -type BLKey -key keychain.der -outfile pubkey.bin
```

The following is an example of extracting the public key as a C header file, suitable for inclusion into an application:

```
signtool getkey -type BLKeyC -key keychain.der -outfile pubkey.h
```

Index

-64 splitter switch *176*
-a name=num switch (elf2dyn utility) *195*
-a sec=addr switch (dynreloc utility) *201*
-b prom|flash|spi|spislave|UART|TWI|FIFO, loader switch for ADSP-BF53x processors *72*
-b prom|flash|spi|spislave|UART|TWI|FIFO|OTP|NAND, loader switch for ADSP-BF51x/52x/54x processors *31*, *33*
-bprom|host|link|JTAG, loader switch for ADSP-21160 processors *113*
-bprom|host|link|spi, loader switch for ADSP-21161 processors *133*
-bprom|spislave|spiflash|spimaster|spiprom, loader switch for ADSP-2126x/36x/37x/46x processors *146*, *166*
-callback, loader switch for Blackfin *34*
-compression
  loader switch for Blackfin *67*, *73*
  loader switch for SHARC *161*, *164*, *166*
-compressionOverlay, loader switch for SHARC *162*, *164*, *167*
-compressWS
  loader switch for Blackfin *71*
  loader switch for SHARC *165*, *167*
-compressWS # *73*
-CRC32, loader switch for Blackfin *34*, *87*, *94*
-dm, splitter switch *176*
-dmawidth #, loader switch for Blackfin *34*, *73*
-e filename, loader switch for ADSP-21160 processors *113*
-e symbol switch (elf2dyn utility) *195*, *197*
-efilename, loader switch for SHARC *133*
-enc dll_filename, loader switch for Blackfin *73*
-f {hex|ascii|binary|include} *74*
-f family switch (elf2dlm utility) *199*
-f h|s1|s2|s3|b, splitter switch *176*
-f hex|ascii|binary|include, loader switch for Blackfin *34*
-fhex|ascii|binary|byte|include|s1|s2|s3, loader switch for SHARC *167*
-fhex|ascii|binary|include|s1|s2|s3, loader switch for SHARC *114*, *133*
-FillBlock, loader switch for Blackfin *31*, *34*, *189*
-ghc #, loader switch for Blackfin *74*
-h switch
  dynreloc utility *201*
  elf2dyn utility *195*
-h|help
  loader switch for Blackfin *35*, *74*
  loader switch for SHARC *114*, *134*, *167*
-hostwidth #, loader switch for SHARC *134*, *145*, *146*, *153*, *154*, *167*
-id#exe=filename
  loader switch for SHARC *111*, *114*, *134*, *168*
-id#exe=N, loader switch for SHARC *134*
-id#ref=N, loader switch for SHARC *114*, *168*
-init filename, loader switch for Blackfin *35*, *39*, *41*, *49*, *65*, *74*, *78*
-initcall, ADSP-BF52x/54x Blackfin loader switch *35*, *41*
-kb prom|flash|spi|spislave|UART|TWI|FIFO, loader switch for Blackfin *75*
-kb prom|flash|spi|spislave|uart|twi|fifo|otp|nand, loader switch for Blackfin *36*
-keep switch (elf2elf utility) *199*
-kenc dll_filename, loader switch for Blackfin *75*
-kf #, loader switch for Blackfin *37*
-kf hex|ascii|binary|include, loader switch for Blackfin *75*
-kp #, loader switch for Blackfin *37*, *39*, *76*, *78*
-kwidth #, loader switch for Blackfin *37*, *76*
-l switch (elf2dyn utility) *195*, *196*
-l userkernel
  loader switch for Blackfin *37*, *65*
  loader switch for SHARC *115*, *134*, *149*, *165*, *168*
-m switch (dynreloc utility) *201*
-M, loader switch for Blackfin *38*, *76*, *77*
-maskaddr #, loader switch for Blackfin *38*, *76*
-MaxBlockSize #, loader switch for Blackfin *38*, *76*, *88*, *95*
-MaxFillBlockSize #, loader switch for Blackfin *38*
-MaxZeroFillBlockSize #, loader switch for Blackfin *77*, *88*, *95*
-merge switch (elf2elf utility) *199*
-MM, loader switch for Blackfin *38*, *77*
-Mo filename, loader switch for Blackfin *38*, *77*
-Mt filename, loader switch for Blackfin *38*, *77*
-NoFillBlock, loader switch for Blackfin *31*, *38*
-nofinalblock, loader switch for Blackfin *77*
-nofinaltag, loader switch
  ADSP-BF561 processors *77*
  ADSP-BF60x processors *86*
-noinitcode, loader switch for Blackfin *39*, *77*
-nokernel
  loader switch for ADSP-2126x/36x/37x/46x processors *169*
-norom, splitter switch *176*
-nosecondstageloader, loader switch for Blackfin *77*
-nozeroblock, loader switch for SHARC *135*, *169*
-o filename
  dynreloc utility switch *201*
  elf2dyn utility switch *195*
  elf2elf utility switch *199*
  loader switch for Blackfin *39*, *77*
  loader switch for SHARC *115*, *135*, *169*
  splitter switch *176*
-o2, loader switch for Blackfin *36*, *37*, *39*, *75*, *78*
-p #, loader switch for Blackfin *39*, *78*
-paddress, loader switch for SHARC *115*, *135*, *169*
-pflag {#|PF#|PG#|PH#}, loader switch for Blackfin *78*
-pflag #|PF|PG|PH #, loader switch for Blackfin *79–81*
-pm splitter switch *176*
-proc part_number

loader switch for Blackfin 39, 78
loader switch for SHARC 115, 135, 169
splitter switch 177
-quickboot, loader switch for Blackfin 39
-r #, splitter switch 177
-ram, splitter switch 176, 177
-readall, loader switch for Blackfin 40
-retainSecondStageKernel, loader switch for SHARC 170
-romsplitter, loader switch for Blackfin 38, 40, 76, 78
-s section_name, splitter switch 177
-save section, loader switch for Blackfin 40
-ShowEncryptionMessage, loader switch for Blackfin 79
-si-revision none|any|x.x
loader switch 178
loader switch for Blackfin 41, 79
loader switch for SHARC 115, 135, 170
-t#, loader switch for SHARC 115, 135
-u
loader switch for SHARC 171
-u, splitter switch 178
-use32bitTagsforExternal Memory Blocks, loader switch for SHARC 115
-v (verbose)
loader switch for Blackfin 41, 79
loader switch for SHARC 116, 135, 170
-v switch
dynreloc utility 201
elf2dyn utility 195
-version
elf2elf utility switch 199
loader switch for SHARC 116, 136, 170
splitter switch 178
-width #, loader switch for Blackfin 41, 76, 79
-zeroPadForced #, loader switch for Blackfin 79
.asm (assembly) source files 19, 180
.bss files (DLM format) 196
.dat (data) initialization files 180
.data files (DLM format) 196
.dlb (library) files 182
.doj (object) files 182
.dxe (executable) files 23, 25, 182, 191
.h_# (Intel hex-32) file format 175, 176, 183, 189
.knl (kernel code) files 25
.ldr (loader output) files
ASCII format 181, 191
binary format 185, 188
hex-32 format 183
include format files 185
naming 39, 77
specifying host bus width 134, 167
.map (memory map) files 182
.s_# (Motorola S-record) files 175, 186, 191
.sm (shared memory) files 25, 113, 133, 182, 191
.stk (byte-stacked) files 175, 176, 178, 189
.text files (DLM lite format) 196
.txt (ASCII text) files 181
.VAR directive 180
16- to 48-bit word packing 105
4- to 48-bit word packing 106
48- to 8-bit word packing 104
8- to 48-bit word packing 104, 105, 120, 122

A

ACK pin 103, 104, 121
ADDR23-0 address lines 122
address records, linear format 183
ADSP-21160 processors
ADSP-21160 boot modes 99, 102
boot sequence 100
ADSP-21161 processors
boot modes 117, 119
boot sequence 118
multiprocessor support 130
ADSP-2126x/36x/37x/46x processors
boot modes 137, 142
boot sequence 138
ADSP-2136x/37x/4xx processors, multiprocessor support 159
ADSP-BF50x processors
boot modes 28
multi-dxe loader files 41
ADSP-BF51x processors
boot modes 29
multi-dxe loader files 41
ADSP-BF52x/54x processors
boot modes 29
multi-dxe loader files 41
ADSP-BF531/2/3/4/6/7/8/9 processors
ADSP-BF534/6/7 (only) boot modes 45
boot modes 44
boot streams 47, 48
compression support 67
memory ranges 55
multi-dxe loader files 64, 65
on-chip boot ROM 44, 46, 47, 55, 65
ADSP-BF561 processors
boot modes 57
boot streams 58–63
dual-core architecture 56
memory ranges 64
multi-dxe loader files 64, 65
multiprocessor support 63
on-chip boot ROM 56, 57, 63–65
ADSP-BF60x processors
BCODE field 85
boot modes 84
ADSP-BF70x processors
BCODE field 92
boot modes 92
application loading (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 50, 66
ADSP-BF561 processors 57, 64, 66
application loading (SHARC processors)
ADSP-21161 processors 118, 120, 122
ADSP-2126x/36x/37x processors 156
ADSP-2126x/36x/37x/46x processors 138

applications
code start address 39, 78, 100, 109, 120
development flow 18
loading, introduction to 23
multiple-dxe files 41
archive files, See library files (.dlb) 182
archiver 182
ASCII file format 34, 74, 181, 191
assembling, introduction to 19
assembly
directives 180
initialization data files (.dat) 180
object files (.doj) 182
source text files (.asm) 19, 180
asynchronous FIFO boot mode, ADSP-BF52x/54x processors 30

B

baud rate (Blackfin processors) 59
BCODE pins, ADSP-BF60x processors 85
BCODE pins, ADSP-BF70x processors 92
BFLAG_CALLBACK block flag 34
BFLAG_QUICKBOT block flag 40
BFLAG_SAVE block flag 40
binary format files (.ldr) 34, 74, 185
bit-reverse option (SHARC processors) 145
block
byte counts (Blackfin processors) 38, 76
of application code, introduction to 23
tags 108, 128, 149, 152
block headers (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 47, 48
ADSP-BF561 processors 59, 63
ADSP-BF60x processors 85
ADSP-BF70x processors 92
block headers (SHARC processors)
ADSP-21161 processors 127
ADSP-2126x/36x/37x processors 149, 151
blocks of application code (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 47
ADSP-BF561 processors 58
ADSP-BF60x processors 88
ADSP-BF70x processors 94
blocks of application code (SHARC processors)
ADSP-21161 processors 127
ADSP-2126x/36x/37x processors 151
BMODE1-0 pins, ADSP-BF531/2/3/8/9 processors 44, 53
BMODE2-0 pins
ADSP-BF51x processors 28, 29
ADSP-BF534/6/7 processors 45
BMODE3-0 pins, ADSP-BF52x/54x processors 29
BMS pins
ADSP-21160 processors 105, 106, 111
ADSP-21161 processors 118, 120, 122, 125, 126, 130
boot
sequences, introduction to 20
boot differences (Blackfin processors) 44, 56, 58
boot differences (SHARC processors) 144, 147
boot file formats
specifying for Blackfin processors 34, 74
specifying for SHARC processors 114, 133, 167
boot mode select pins (Blackfin processors)
ADSP-BF51x processors 28, 29
ADSP-BF52x/54x processors 29
ADSP-BF531/2/3/4/6/7/8/9 processors 44
ADSP-BF60x processors 85
ADSP-BF70x processors 92
boot mode select pins (SHARC processors)
ADSP-21161 processors 118
ADSP-2116x/160 processors 101
ADSP-2126x/36x/37x processors 140
boot modes (Blackfin processors)
ADSP-BF50x processors 28
ADSP-BF51x processors 28, 29
ADSP-BF52x/54x processors 28, 29
ADSP-BF531/2/3/8/9 processors 32
ADSP-BF534/6/7 processors 44, 45
ADSP-BF561 processors 57
ADSP-BF60x processors 84
ADSP-BF70x processors 92
specifying 33, 72
boot modes (SHARC processors)
ADSP-21160 processors 99, 102
ADSP-21161 processors 117, 119
ADSP-2126x/36x/37x processors 140
specifying 113, 133, 140, 166
boot sequences (SHARC processors)
ADSP-21161 processors 118
ADSP-2116x/160 processors 100
ADSP-2126x/36x/37x/46x processors 138
boot streams (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 47, 65
ADSP-BF561 processors 58–63, 65
boot streams (SHARC processors)
ADSP-21160 processors 108
ADSP-21161 processors 127
ADSP-2126x/36x/37x processors 149, 151
boot streams, introduction to 22, 23
boot-loadable files
introduction to 19, 20
versus non-bootable file 23
boot-loading sequence 103
bootstraps 23
BSO bit 104
BUSLCK bit 106
bypass mode, See no-boot mode 44
BYTE format (non-bootable files) 188
BYTE format files (.ldr) 188
byte-stacked format files (.stk) 175, 176, 178, 189

C

C and C++ source files 19, 179
CEP0 register 121–124
CLB0 register 125
CLKPL bit 143
command line

loader for SHARC processors 112, 132, 165
loader/splitter for Blackfin processors 32, 72, 112, 132
splitter 173, 175
compilation, introduction to 19
compressed block headers
Blackfin processors 50, 68
SHARC processors 163
compressed streams
Blackfin processors 67, 70
SHARC processors 162, 164
compression support
ADSP-BF531/2/3/4/6/7/8/9 processors 67
compression window 69, 71, 163, 165
conversion utilities 193, 194, 199
count headers (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 65
ADSP-BF561 processors 59, 63, 65
CPEP0 register 121, 123
CPHASE bit 143
CPLB0 register 125
CRC32 protection, ADSP-BF60x processors 87
CRC32 protection, ADSP-BF70x processors 94
CS pin 105, 124, 142
CSRX register 126
Cx register 103, 104, 107

D

D39-32 bits 104
data
initialization files (.dat) 180
memory (dm) sections 174, 176
data banks (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 55
ADSP-BF561 processors 64
data packing (SHARC processors)
ADSP-21160 processors 104–106
ADSP-21161 processors 120, 122
ADSP-2126x/36x/37x/46x processors 141, 153, 154
data streams
encrypting from application 73
encrypting from kernel 75
DATA23-16 pins 120
DATA39-32 pins 102
DATA63-32 pins 105
DATA64 memory sections 174, 176
DataFlash devices 44
debugger file formats 19, 191
debugging targets 19
decompression
initialization files 71
kernel files 165
DEN register 121, 123
DLMs to ELF DLM converter 199
DMA (ADSP-21160 processors)
buffers 105
channel control registers 104–107
channel interrupts 106, 107
channel parameter registers 103–105, 107
controller 99, 103, 104
transfers 104–107, 109
DMA (ADSP-21161 processors)
buffers 131
channel control registers 120, 122, 123, 127
channel interrupts 122, 124
channel parameter registers 121–127
controller 120–122
transfers 118, 122, 126, 127, 130
DMA (ADSP-2126x/36x/37x/46x processors)
parameter registers 143
transfers 147
DMA (ADSP-2126x/36x/37x/46x processors): code example 156
DMA (ADSP-2126x/36x/37x/46x processors): parameter registers 156
DMAC0 channel (ADSP-21160 processors) 105
DMAC10 channels
ADSP-21160 processors 100, 103–105, 107
ADSP-21161 processors 120–122
DMAC6 channel (ADSP-21160 processors) 104, 105, 107
DMAC8 channels
ADSP-21160 processors 105, 106
ADSP-21161 processors 118, 124–126
DMISO bit 143
DTYPE register 105, 121, 124
dual-core applications
ADSP-BF561 processors 63
ADSP-BF60x processors 86
DWARF-2 debugging information 19
Dynamically Loadable Modules (DLMs) 194
dyndump utility 199
dynreloc utility 200

E

EBOOT pins
ADSP-21160 processors 101, 102, 105, 106
ADSP-21161 processors 118, 120, 122, 125, 126, 131
ECEP0 register 121–123
ECx register 103–105
EIEP0 register 121, 123
EIx register 103, 105
ELF to BFLT file converter 198
elf to DLM converter 194
elf2dyn utility 194, 195
elf2flt utility 198
elfdump utility 197
elfloader, *See* loader
EMEP0 register 121, 123
EMx register 103
encrypted images, ADSP-BF70x processors 93
encryption functions 73, 75, 79
end-of-file records 184
EP0I vector 106, 122, 124
EPB0 buffer 105
EPROM boot mode (SHARC processors)

ADSP-21160 processors 99, 101, 102, 111, 112
ADSP-21161 processors 117, 118, 120
multiprocessor systems 130
EPROM flash memory devices 22
executable and linkable format (ELF)
executable files (.dxe) 19, 20, 182
object files (.doj) 182
exported symbols 197
external
memory boot 20
resistors 103
vector tables 111
external memory (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 44, 45, 65
ADSP-BF561 processors 63, 65
multiprocessor support 65
external memory (SHARC processors)
ADSP-21160 processors 102, 104, 106, 107, 109, 111, 115
ADSP-21161 processors 119, 127, 130, 135
ADSP-2126x/36x/37x processors) 153
ADSP-2126x/36x/37x/46x processors 141
external ports (SHARC processors)
ADSP-21160 processors 101, 102, 104–107, 109, 111
ADSP-21161 processors 118, 120–124, 131

F

file formats
ASCII 34, 74, 191
binary 34, 74
byte-stacked (.stk) 175, 176, 178
debugger input files 191
hexadecimal (Intel hex-32) 34, 74, 175, 176
include 34, 74
list of 24
s-record (Motorola) 175, 176
file formatting
selecting for output 37, 75
specifying word width 79
file search rules 24
final blocks
introduction to 23
SHARC processors 109, 149, 155
FLAG pins, ADSP-21160 processors 111
flag words (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 49
ADSP-BF561 processors 59, 63
flash memory
ADSP-BF51x processors 29
ADSP-BF52x/54x processors 29
devices 19
FLG0 signal 143
frequency 106, 125

G

global header structures, See block headers 58, 59, 74
GPEP0 register 121, 123
GPLB0 register 125
GPSRX register 126

H

HBG pin 105
HBR pin 124
header files (.h) 58, 181
header records
byte-stacked format (.stk) 190
s-record format (.s_#) 186
hex-to-S converter 193
hexutil utility 193
hold time cycles 59
host boot mode (SHARC processors)
ADSP-21160 processors 99, 105, 111
ADSP-21161 processors 117, 122
ADSP-2126x/36x/37x/46x processors 147
host boot mode, introduction to 22, 23
host DMA boot mode, ADSP-BF52x/54x processors 30
HPM bit 105

I

IDLE instruction 101, 106, 109, 110, 123
ignore blocks (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 49
ADSP-BF561 processors 59
IIEP0 register 121, 123
IILB0 register 125
IISRX register 126
IIVT bit 111, 130
IIx register 103, 107
image files, See PROM, non-bootable files 173
IMASK register 106, 107
IMDW register 106, 156
IMEP0 register 121, 123
IMLB0 register 125
IMSRX register 126
IMx register 103
include file format 185
INIT_L16 blocks 154
INIT_L48 blocks 153
INIT_L64 blocks 155
initial word option (SHARC processors) 145
initialization blocks (ADSP-2126x/36x/37x/46x processors) 151, 153–156
initialization blocks (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 49, 50, 66
ADSP-BF561 processors 63, 64, 66
initialization blocks (Blackfin processors): code example 51, 66
input file formats, See source file formats 179
input files
executable (.dxe) files 25, 112, 132, 165
extracting memory sections from 176, 177
instruction SRAM (Blackfin processors)

ADSP-BF531/2/3/4/6/7/8/9 processors 55
ADSP-BF561 processors 63, 64
Intel hex-32 file format 31, 34, 74, 183
internal memory, boot-loadable file execution 20
interrupt vector location 122, 124
interrupt vector tables 111, 130, 149, 156
IOP registers 105
IRQ vector 103
IVG15 lowest priority interrupt 46, 50, 58

K

kernels (ADSP-21160 processors)
boot sequence 100, 107
default source files 107, 110
loading to processor 104, 106
modifying 109
rebuilding 110
replacing with application code 109
specifying user kernel 115
kernels (ADSP-21161 processors)
boot sequence 118
default source files 127, 129
modifying 128, 129
rebuilding 128, 129
kernels (ADSP-2126x/36x/37x/46x processors)
boot sequence 138, 147
compression/decompression 161, 162, 164, 165
default source files 147
loading to processor 143, 144
modifying 148, 150
omitting in output 149
rebuilding 150
kernels (Blackfin processors)
compression/decompression 67, 70
specifying boot mode 36, 75
specifying file format 37, 75
specifying file width 76
specifying hex address 37, 76
specifying user kernel 37

L

L1 memory (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 47, 50, 55
ADSP-BF561 processors 58, 64
L2 memory (Blackfin processors)
ADSP-BF561 processors 64
last blocks (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 50, 51
ADSP-BF561 processors 59
LBOOT pins
ADSP-21161 processors 101, 102, 105, 118, 120, 122, 125, 126
LCOM register 107
LCTL register 107, 109, 125
least significant bit first (LSB) format 144
library files (.dlb) 182
link buffers 106, 107, 124, 125
link port boot mode
ADSP-2146x SHARC processors 166
link port boot mode (SHARC processors)
ADSP-21160 processors 99, 101
ADSP-21161 processors 117, 119, 124
linker
command-line files (.txt) 181
memory map files (.map) 182
output files (.dxe, .sm, .ovl) 19, 182
linking, introduction to 19
loadable files, See boot-loadable files 19
loader
operations 20
output file formats 21, 23, 183, 185, 186, 188
loader for ADSP-21160 processors 99
loader for ADSP-21161 processors 117
loader for ADSP-2126x/36x/37x/469 processors 137
loader for ADSP-BF51x/52x/54x Blackfin (includes splitter) 27
loader for ADSP-BF53x/BF561 Blackfin (includes splitter) 43
loader for ADSP-BF60x Blackfin (includes splitter) 83
loader for ADSP-BF70x Blackfin (includes splitter) 91
loader kernels, See boot kernels 23
loader switches, See switches by name 33, 72
loading, introduction to 19

M

make files 38, 76, 77
masking EPROM address bits 38, 76
master (host) boot, introduction to 20
memory map files (.map) 182
memory programming on ADSP-BF60x processors 87
memory ranges (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 55
ADSP-BF561 processors 64
microcontroller data transfers 189
MODE1 register 106
MODE2 register 106
most significant bit first (MSB) format 144
Motorola S-record file format 186
MS bit 143
MSBF bit 143
MSWF register 121, 124
multiprocessor booting, introduction to 20
multiprocessor systems (Blackfin processors) 41, 65
multiprocessor systems (SHARC processors)
ADSP-21160 processors 112
ADSP-21161 processors 120, 130, 131

N

no-boot mode
introduction to 20, 22
selecting with -romsplitter switch 40, 78
no-boot mode (Blackfin processors)
ADSP-BF50x processors 28
ADSP-BF51x processors 29
ADSP-BF52x/54x processors 29
ADSP-BF531/2/3/4/6/7/8/9 processors 44
ADSP-BF561 processors 58

no-boot mode (SHARC processors)
- ADSP-21160 processors 99, 107
- ADSP-21161 processors 117, 119, 127

non-bootable files
- creating from command line 174
- ignoring ROM sections 176
- introduction to 20, 23
- specifying format 176
- specifying name 176
- specifying word width 174, 177

non-bootable output files 188
NOP instruction 101, 106, 109, 110, 123
numeric formats 180

O

object files (.doj) 182
on-chip boot ROM
- ADSP-BF531/2/3/4/6/7/8/9 processors 23, 44, 46, 47, 49, 55, 65
- ADSP-BF561 processors 56, 57, 63–65
- introduction to 23

OTP boot mode, ADSP-BF51x/52x/54x processors 29
output files
- generating kernel and application 39, 78
- specifying format 21, 182
- specifying name 39, 77
- specifying with -o switch 194
- specifying word width 79, 134

overlay compression 164
overlay memory files (.ovl) 25, 182, 191

P

packing boot data 117
parallel/serial PROM devices 22
PFx signals 78
pin
- ACK 105

PMODE register 104, 105, 121, 124
processor IDs
- assigning to .dxe file 114, 134, 168
- pointing to jump table 114, 134

processor type bits (Blackfin boot streams) 49
processor-loadable files, introduction to 22
program counter settings (ADSP-21160 processors) 105
program development flow 18
program memory sections (splitter) 174, 176
PROM
- boot mode, introduction to 22
- downloading boot-loadable files 20
- memory devices 147, 183

PROM (image) files
- creating from command line 174
- ignoring ROM sections 176
- specifying format 176
- specifying name 176
- specifying width 177

PROM boot mode, ADSP-2126x/36x/37x/46x processors 141, 150
PROM/flash boot mode (Blackfin processors)
- ADSP-BF531/2/3/4/6/7/8/9 processors 44, 45, 66
- ADSP-BF561 processors 63, 66

pull-up resistors 121
Px register 109, 156

R

RBAM bit 121
RBWS bit 121
RD pin 104, 122
reset
- ADSP-21160 processors 100, 103–105, 107
- ADSP-21161 processors 118, 120–123, 125, 126
- ADSP-2126x/36x/37x/46x processors 138, 148
- ADSP-BF561 processors 53, 56, 58
- Blackfin processors 28, 43, 44
- dual-core Blackfin processors 56
- processor, introduction to 22, 23
- vector addresses 100, 104, 106, 129
- vector routine 122

RESET
- interrupt service routine 46, 58, 124
- pin 103, 121

RESET pin 124
reset: vector routine 55
ROM
- memory images as ASCII text files 191
- memory sections 176

Rx registers 63, 66, 106

S

s1 (Motorola EXORciser) file format 176, 186
s2 (Motorola EXORMAX) file format 176, 186
s3 (Motorola 32-bit) file format 176, 186
scratchpad memory (Blackfin processors)
- ADSP-BF561 processors 64

SDCTL register 129
SDRAM memory (ADSP-21160 processors) 107
SDRAM memory (Blackfin processors)
- ADSP-BF531/2/3/4/6/7/8/9 processors 49, 56
- ADSP-BF561 processors 63, 64

SDRAM memory (Blackfin processors): ADSP-BF531/2/3/4/6/7/8/9 processors 51
SDRAM/DDR boot mode, ADSP-BF52x/54x processors 30
SDRDIV register 129
second-stage loader
- ADSP-BF561 processors 63

secure boot, ADSP-BF70x processors 93
SENDZ bit 143
sequential EPROM boot 131
shared memory

Blackfin processors 63, 64
file format (.sm) 25, 63, 182, 191
in compressed .ldr files 162, 164
omitting from loader file 113, 133
shift register, See RX registers 142
simulators, for boot simulation 20
single-processor systems 112, 131, 174
slave processors 20, 22, 143
software reset 22, 47, 58
source file formats
assembly text (.asm) 180
C/C++ text (.c, .cpp, .cxx) 179
SPI boot modes (SHARC processors)
ADSP-21161 processors 117, 119, 125
ADSP-2126x/36x/37x/46x processors 142, 145, 150
SPI EEPROM boot mode (Blackfin processors)
ADSP-BF561 processors 63
SPI flash boot mode (ADSP-2126x/2136x/2137x/21469 processors) 147
SPI host boot mode (ADSP-2126x/36x/37x/46x processors) 147
SPI master boot modes
ADSP-2126x/36x/37x processors 149
ADSP-2126x/36x/37x/46x processors 143, 145
ADSP-BF51x processors 29
ADSP-BF52x/54x processors 30
ADSP-BF531/2/3/8/9 processors 44
ADSP-BF534/6/7 processors 30, 45
ADSP-BF60x processors 85
ADSP-BF70x processors 92
SPI memory slave devices 144
SPI PROM boot mode (ADSP-2126x/36x/37x/46x processors) 145–147
SPI slave boot mode (ADSP-2126x/2136x/2137x/21469 processors) 143
SPI slave boot mode (ADSP-2126x/36x/37x/46x processors) 142, 145
SPI slave boot mode (Blackfin processors)
ADSP-BF51x processors 29
ADSP-BF52x/54x processors 30
ADSP-BF531/2/3/8/9 processors 44
ADSP-BF534/6/7 processors 45
SPICLK register 143, 144, 147
SPICTL register 126
SPIDS signal 143
SPIEN bit 143
SPIRCV bit 143
SPIRx register 118, 125, 126
splitter
command-line syntax 173
file extensions 175
introduction to 19–22
list of switches 175
output file formats 189, 191
SPORT hex data files 191
SRAM memory (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 55
ADSP-BF561 processors 57, 64
start addresses
ADSP-21160 application code 100
Blackfin application code 39, 78
status information 41, 79
supervisor mode (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 46
ADSP-BF561 processors 58
synchronous boot operations 105
SYSCON register (SHARC processors)
ADSP-21160 processors 105, 109, 111
ADSP-21161 processors 129, 130
SYSCR register (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 47
ADSP-BF561 processors 58
SYSCTL register 156
SYSTAT register 111
system reset configuration register, See SYSCR register 28, 43

T

termination records 187
text files 191
Tool Settings dialog box 32, 71, 88, 95, 112, 131, 149, 165
two-wire interface (TWI) boot mode
ADSP-BF2x/54x processors 30
ADSP-BF534/6/7 processors 45

U

UART slave boot mode (Blackfin processors) 29, 30, 45
UBWM register 104
uncompressed streams 70, 163
utility programs (list) 193

V

vector addresses 110, 130

W

WAIT register 104, 107, 109, 121, 129
wait states 104, 106, 121, 122
WL bit 143
word width, setting for loader output file 134

Z

zero-fill blocks (Blackfin processors)
ADSP-BF531/2/3/4/6/7/8/9 processors 49, 77
ADSP-BF561 processors 59
zero-fill blocks (SHARC processors)
ADSP-21160 processors 109
ADSP-2126x/36x/37x processors 152
zero-padding (ADSP-2126x/36x/37x/46x processors) 153, 154