CrossCore Embedded Studio 2.2.0

Assembler and Preprocessor Manual

Revision 1.6, February 2016

Part Number
82-100121-01

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

Contents

Preface

Assembler

Preprocessor

1 Preface

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

Purpose of This Manual

The *Assembler and Preprocessor Manual* contains information about the assembler and preprocessor utilities for the following Analog Devices processor families: Blackfin (ADSP-BFxxx) and SHARC (ADSP-21xxx/SC5xx).

The manual describes how to write assembly programs for these processors and provides reference information about related development software. It also provides information on new and legacy syntax for assembler and preprocessor directives and comments, as well as command-line switches.

Intended Audience

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

Manual Contents

The manual consists of:

- Assembler

  Provides an overview of the process of writing and building assembly programs. It also provides information about assembler switches, expressions, keywords, and directives.

- Preprocessor

  Provides procedures for using preprocessor commands within assembly source files as well as the preprocessor's command-line interface options and command sets.

What's New in This Manual

This is Revision 1.6 of the *Assembler and Preprocessor Manual*, supporting CrossCore Embedded Studio (CCES) 2.2.0. Modifications and corrections based on errata reports against this manual have been made. For future revisions, this section will document assembler and preprocessor functionality that is new to CCES updates, including support for new Blackfin and/or SHARC processors.

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

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

Supported Processors

The CCES assembler and preprocessor support the following processor families from Analog Devices.

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

Refer to the CCES online help for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices website and the CCES online help.

Analog Devices Website

The Analog Devices website, 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 website 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, Inc. 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, can 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`.

Example	*Description*
`[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 `letter gothic` font and 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.
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 ***ATTENTION:*** appears instead of this symbol.

2 Assembler

This chapter provides information on how to use the assembler to develop and assemble programs for SHARC (ADSP-21xxx/ADSP-SCxxx) and Blackfin (ADSP-BFxxx) processors.

The chapter contains the following sections:

- Assembler Guide

  Describes how to develop new programs using the processors' assembly language.

- Assembler Syntax Reference

  Provides the assembler rules and conventions of syntax used to define symbols (identifiers), expressions, and to describe different numeric and comment formats.

- Assembler Command-Line Reference

  Provides reference information on the assembler's switches and conventions.

Assembler Guide

Using CCES, you can run the assembler drivers for each processor family from the Integrated Development Environment (IDE) or from an operating system command line. The assembler processes assembly source, data, and header files to produce an object file. Assembler operations depend on two types of controls: assembler directives and assembler switches.

CCES contains the following assembler drivers.

- `easm21k.exe` (for SHARC processors)
- `easmblkfn.exe` (for Blackfin processors)

This section describes how to develop new programs in the Analog Devices processor assembly language. It provides information on how to assemble your programs from the operating system's command line.

Software developers using the assembler should be familiar with these topics:

- Writing Assembly Programs
- Using Assembler Support for C Structs

- Preprocessing a Program
- Using Assembler Feature Macros
- Generating Make Dependencies
- Reading a Listing File
- Specifying Assembler Options

For information about a processor's architecture, including the instruction set used when writing assembly programs, refer to the *Hardware Reference* and *Programming Reference* for the appropriate processor.

Assembler Overview

The assembler processes data from assembly source (`.asm`), data (`.dat`), and header (`.h`) files to generate object files in Executable and Linkable Format (ELF), an industry-standard format for binary object files. The object file has a `.doj` extension.

In addition to the object file, the assembler can produce a listing file (`.lst`) that shows the correspondence between the binary code and the source.

Assembler switches are specified from the IDE or from the command line used to invoke the assembler. These switches allow you to control the assembly process of source, data, and header files. Use these switches to enable and configure assembly features, such as search paths, output file names, and macro preprocessing. For more information, see Assembler Command-Line Reference.

You can also set assembler options via the assembler pages of the ***Tool Settings*** dialog box in the IDE. For more information, see Specifying Assembler Options.

Writing Assembly Programs

Assembler directives are coded in assembly source files. The directives allow you to define variables, set up hardware features, and identify program sections for placement within processor memory. The assembler uses directives for guidance as it translates a source program into object code.

Write assembly language programs using the IDE editor or any editor that produces text files. Do not use a word processor that embeds special control codes in the text. Use an `.asm` extension to source file names to identify them as assembly source files.

The ***Assembler Input and Output Files*** figure shows a graphical overview of the assembly process. The figure shows the preprocessor processing the assembly source (`.asm`) and header (`.h`) files.

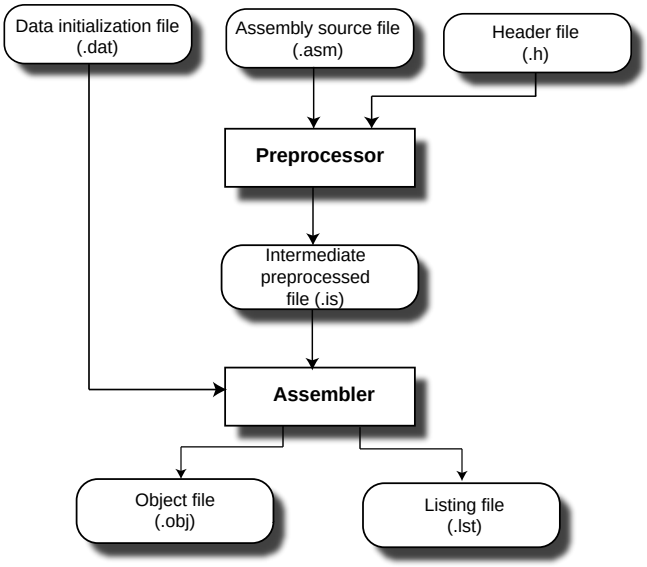

Figure 2-1: Assembler Input and Output Files

Assemble your source files from the IDE or using any mechanism, such as a batch file or makefile, that supports invoking an appropriate assembler driver with a specified command-line command. By default, the assembler processes an intermediate file to produce a binary object file (`.doj`) and an optional listing file (`.lst`).

Object files produced by the processor assembler may be used as input to the linker and archiver. You can archive the output of an assembly process into a library file (`.dlb`), which can then be linked with other objects into an executable. Use the linker to combine separately assembled object files and objects from library files to produce an executable file. For more information about the linker and archiver, refer to the *Linker and Utilities Manual* .

A binary object file (`.doj`) and an optional listing (`.lst`) file are final results of the successful assembly .

The assembler listing file is a text file read for information on the results of the assembly process. The listing file also provides information about the imported C data structures. The listing file tells which imports were used within the program, followed by a more detailed section; see the .IMPORT, Provide Structure Layout Information directive. The file shows the name, total size, and layout with offset for the members. The information appears at the end of the listing. You must specify the -l filename switch to produce a listing file.

The assembly source file may contain preprocessor commands, such as `#include`, that cause the preprocessor to include header files (`.h`) into the source program. The preprocessor's only output, an intermediate source file (`.is`), is the assembler's primary input. In normal operation, the preprocessor output is a temporary file that is deleted during the assembly process.

Program Content

Assembly source file statements include assembly instructions, assembler directives, and preprocessor commands.

Assembly Instructions

Instructions adhere to the processor's instruction set syntax, which is documented in the processor's *Programming Reference*. Each instruction line must be terminated by a semicolon (`;`). The ***Assembly Code File Structure for SHARC Processors*** figure in Code File Structure for SHARC Processors shows a sample assembly source file.

To mark the location of an instruction, place an address label at the beginning of an instruction line or on the preceding line. End the label with a colon (`:`) before beginning the instruction. Your program can then refer to this memory location using the label instead of an address. The assembler places no restriction on the number of characters in a label.

Labels are case sensitive. The assembler treats “outer” and “Outer” as unique labels. For example (in Blackfin processors),

```
outer:  [I1] = R0;
Outer:  R1 = 0X1234;
JUMP outer; // jumps back 2 instructions
```

Assembler Directives

Assembler directives begin with a period (`.`) and end with a semicolon (`;`). The assembler does not differentiate between directives in lowercase or uppercase.

NOTE: This manual prints directives in uppercase to distinguish them from other assembly statements.

Example (Blackfin processors):

```
.SECTION data1;
.BYTE2 sqrt_coeff[2] = 0x5D1D, 0xA9ED;
```

For details, see Assembler Directives.

Preprocessor Commands

Preprocessor commands begin with a pound sign (`#`) and extend to the end of the current line. The pound sign must be the first non-white space character on the line containing the command. If the command is longer than one line, use a backslash (`\`) at the end of the line to continue the command onto the next line.

Do not put any characters between the backslash and the end of the line. Unlike assembler directives, preprocessor commands are case sensitive and must be lowercase. For example,

```
#include "string.h"
#define MAXIMUM 100
```

For more information, see Writing Preprocessor Commands. For a list of the preprocessor commands, see Preprocessor Command Reference.

Program Structure

An assembly source file defines code (instructions) and data. It also organizes the instructions and data to allow the use of the linker description file (`.ldf`) to describe how code and data are mapped into the memory on your target processor. The way you structure your code and data into memory should follow the memory architecture of the target processor.

Use the `.SECTION` directive to organize the code and data in assembly source files. The `.SECTION` directive defines a grouping of instructions and data that occupies contiguous memory addresses in the processor. The name given in a `.SECTION` directive corresponds to an input section name in the linker description file.

The ***Suggested Input Section Names for a SHARC .ldf File*** and ***Suggested Input Section Names for a Blackfin.ldf File*** tables show suggested input section names for data and code that can be used in your assembly source for various processors. Using these predefined names in your sources makes it easier to take advantage of the default Linker Description File, LDF, (with extension `.ldf`) included in your installation. However, you may also define your own sections. For information on `.ldf` files, refer to the *Linker and Utilities Manual*.

Table 2-1: Suggested Input Section Names for a SHARC .ldf File

`.SECTION` *Name*	*Description*
`seg_pmco`	A section in program memory that holds code
`seg_dmda`	A section in data memory that holds data
`seg_pmda`	A section in program memory that holds data
`seg_rth`	A section in program memory that holds system initialization code and interrupt service routines
`seg_swco`	A section in short word memory that holds instructions encoded for execution from short word memory **NOTE**: Applies to the ADSP-214xx processors only.

Table 2-2: Suggested Input Section Names for a Blackfin .ldf File

`.SECTION` Name	Description
`data1`	A section that holds data
`program`	A section that holds code
`constdata`	A section that holds global data (which is declared as constant) and literal constants such as strings and array initializers

Use sections in a program to group elements to meet hardware constraints. For example, the ADSP-BF533 processor has a separate program and data memory in Level 1 memory only. Level 2 memory and external memory are not separated into instruction and data memory.

To group the code that resides in off-chip memory, declare a section for that code and place that section in the selected memory with the linker.

The *Assembly Code File Structure for SHARC Processors* figure in Code File Structure for SHARC Processors and the *Assembly Source File Structure for Blackfin Processors* figure in Code File Structure for Blackfin Processors describe the assembly code file structure for each processor family. They show how a program divides into sections that match the memory segmentation of a DSP system. Notice that an assembly source may contain preprocessor commands (such as `#include` to include other files in source code), `#ifdef` (for conditional assembly), or `#define` (to define macros). Assembler directives, such as `.VAR` (or `.BYTE` for Blackfin processors), appear within sections to declare and initialize variables.

Code File Structure for SHARC Processors

The ***Assembly Code File Structure for SHARC Processors*** figure demonstrates assembly code file structure for SHARC processors.

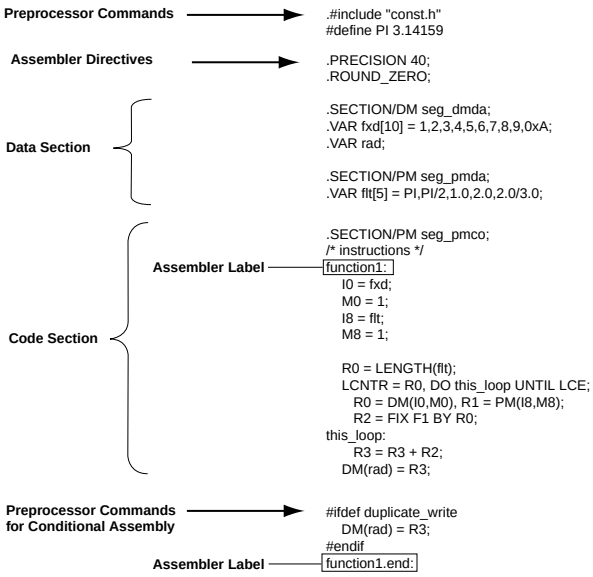

Figure 2-2: Assembly Code File Structure for SHARC Processors

Looking at figure, notice that the `.PRECISION` and `.ROUND_ZERO` directives inform the assembler to store floating-point data with 40-bit precision and to round a floating-point value to a closer-to-zero value if it does not fit in the 40-bit format.

Code File Structure for Blackfin Processors

The ***Assembly Code File Structure for Blackfin Processors*** figure demonstrates the Blackfin processor's assembly code file structure and shows how a program divides into sections that match the memory segmentation of Blackfin processors.

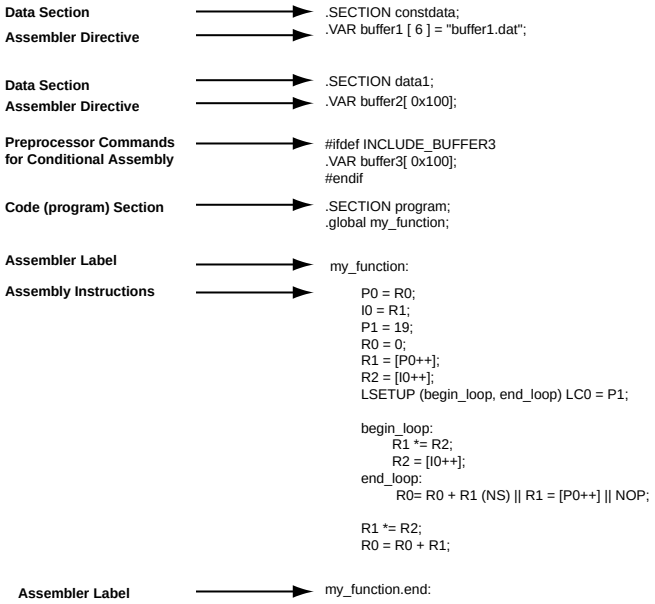

Figure 2-3: Assembly Source File Structure for Blackfin Processors

Program Interfacing Requirements

You can interface your assembly program with a C or C++ program. The C/C++ compiler supports two methods for mixing C/C++ and assembly language:

- Embedding assembly code in C or C++ programs
- Linking together C or C++ and assembly routines

To embed (inline) assembly code in your C or C++ program, use the `asm()` construct. To link together programs that contain C/C++ and assembly routines, use assembly interface macros. These macros facilitate the assembly of mixed routines. For more information about these methods, see the *C/C++ Compiler and Library Manual* for the appropriate target processor.

When writing a C or C++ program that interfaces with assembly, observe the same rules that the compiler follows as it produces code to run on the processor. These rules for compiled code define the compiler's run-time environment. Complying with a run-time environment means following rules for memory usage, register usage, and variable names.

The definition of the run-time environment for the C/C++ compiler is provided in the *C/C++ Compiler and Library Manual* for the appropriate target processor, which also includes a series of examples to demonstrate how to mix C/C++ and assembly code.

Using Assembler Support for C Structs

The assembler supports C `typedef/struct` declarations within assembly source. These assembler data directives and built-ins provide high-level programming features with C structs in the assembler.

Data Directives:

- `.IMPORT` - see .IMPORT, Provide Structure Layout Information
- `.EXTERN STRUCT` - see .EXTERN STRUCT, Refer to a Struct Defined Elsewhere
- `.STRUCT` - see .STRUCT, Create a Struct Variable

C Struct in Assembly Built-Ins:

- `OFFSETOF(struct/typedef,field)` - see OFFSETOF Built-In Function
- `SIZEOF(struct/typedef)` - see SIZEOF Built-In Function

Struct References:

- `struct->field` - see Struct References

For more information on C struct support, refer to the `-flags-compiler` command-line switch and Reading a Listing File.

C structs in assembly features accept the full set of legal C symbol names, including those that are otherwise reserved in the appropriate assembler. For example,

- In the SHARC assembler, `I1`, `I2`, and `I3` are reserved keywords, but it is legal to reference them in the context of the C struct in assembly features.
- In the Blackfin assembler, as an example, "`X`" and "`Z`" are reserved keywords, but it is legal to reference them in the context of the C struct in assembly features.

The examples below show how to access the parts of the struct defined in the header file, but they are not complete programs on their own. Refer to your DSP project files for complete code examples.

Blackfin Example:

```
.IMPORT "Coordinate.h";
      /* typedef struct Coordinate {
           int X;
           int Y;
           int Z;
       } Coordinate;*/
.SECTION data1;
.STRUCT Coordinate Coord1 = {
      X = 1,
      Y = 4,
      Z = 7
      };
.SECTION program;
      P0.l = Coord1->X;
      P0.h = Coord1->X;
      P1.l = Coord1->Y;
      P1.h = Coord1->Y;
      P2.l = Coord1->Z;
```

```
      P2.h = Coord1->Z;
      P3.l = Coord1+OFFSETOF(Coordinate,Z);
      P3.h = Coord1+OFFSETOF(Coordinate,Z);
```

SHARC Example:

```
.IMPORT "Samples.h";
      /*   typedef struct Samples {
             int I1;
             int I2;
             int I3;
        } Samples; */
.SECTION/DM seg_dmda;
.STRUCT Samples Sample1 ={
      I1 = 0x1000,
      I2 = 0x2000,
      I3 = 0x3000
      };
.SECTION/PM seg_pmco;
      doubleMe:
      /* The code may look confusing, but I2 can be used both
         as a register and a struct member name */
      B2 = Sample1;
      M2 = OFFSETOF(Sample1,I2);
      R0 = DM(M2,I2);
      R0 = R0+R0;
      DM(M2,I2) = R0;
```

NOTE: For better code readability, avoid using `.STRUCT` member names that have the same spelling as assembler keywords. This may not always be possible if your application needs to use an existing set of C header files.

Preprocessing a Program

The assembler utilizes a preprocessor that allows the use of C-style preprocessor commands in your assembly source files. The preprocessor automatically runs before the assembler unless you use the assembler's `-sp` (skip preprocessor) switch. The ***Preprocessor Command Summary*** table, Preprocessor Commands and Operators in the Preprocessor chapter lists preprocessor commands and provides a brief description of each command.

You can see the command line that the assembler uses to invoke the preprocessor by adding the -v[erbose] switch to the assembler command line or by selecting a project in any project navigation view and choose ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***CrossCore Assembler*** > ***General*** > ***Generate verbose output (-v)***. See Specifying Assembler Options.

Use preprocessor commands to modify assembly code. For example, you can use the `#include` command to include other source files that fill memory, load configuration registers, or set up processor parameters. You can use the `#define` command to define constants and aliases for frequently used instruction sequences. The preprocessor replaces each occurrence of the macro reference with the corresponding value or series of instructions.

For example, the `MAXIMUM` macro from `#define MAXIMUM 100` is replaced with the number `100` during preprocessing.

For more information on the preprocessor command set, see the ***Preprocessor Command Reference***, Preprocessor Command-Line Reference in the Preprocessor chapter. For more information on the preprocessor usage, see -flags-pp -opt1[, -opt2...].

NOTE: There is one important difference between the assembler preprocessor and compiler preprocessor. The assembler preprocessor treats the "." character as part of an identifier. Thus, `.EXTERN` is a single identifier and does not match a preprocessor macro `EXTERN`. This behavior can affect how macro expansion is done for some instructions.

For example,

```
#define EXTERN ox123
.EXTERN Coordinate;     /* EXTERN not affected by macro */

#define MY_REG P0
MY_REG.1 = 14;          /* MY_REG.1 is not expanded; */
                        /* "." is part of token */
```

Using Assembler Feature Macros

The assembler automatically defines preprocessor macros for properties such as the source language, the architecture, and the specific processor. These *feature macros* allow programmers to use preprocessor conditional commands to configure the source for assembly based on the context.

The ***Feature Macros for Blackfin Processors*** table lists the set of feature macros for Blackfin processors. The ***Feature Macros for SHARC Processors*** table lists the set of feature macros for SHARC processors.

Table 2-3: Feature Macros for Blackfin Processors

Macro	*Definition*
`-D_LANGUAGE_ASM=1`	Always present
`-D__ADSPBLACKFIN__=1`	Always present
`-D__ADSPLPBLACKFIN__`	Defined to `1` for ADSP-BF5xx processors, `0x110` for ADSP-BF60x processors, and `0x220` for ADSP-BF70x processors
`-D__NUM_CORES__`	Defined to the number of cores on the target processor, e.g. `1` for ADSP-BF532 and `2` for ADSP-BF561

`-D__ADSPBF504__=1`
`-D__ADSPBF50x__=1`
`-D__ADSPBF5xx__=1`
`-D__ADSPBF506F_FAMILY__=1`
	Present when running `easmblkfn -proc ADSP-BF504`

`-D__ADSPBF504F__=1`	Present when running `easmblkfn -proc ADSP-BF504F`

Table 2-3: Feature Macros for Blackfin Processors (Continued)

Macro	*Definition*
-D__ADSPBF50x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF506F_FAMILY__=1	
-D__ADSPBF506F__=1 -D__ADSPBF50x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF506F_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF506F`
-D__ADSPBF512__=1 -D__ADSPBF51x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF518_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF512`
-D__ADSPBF514__=1 -D__ADSPBF51x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF518_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF514`
-D__ADSPBF516__=1 -D__ADSPBF51x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF518_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF516`
-D__ADSPBF518__=1 -D__ADSPBF51x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF518_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF518`
-D__ADSPBF522__=1 -D__ADSPBF52x__=1 -D__ADSPBF52xLP__=1 -D__ADSPBF5xx__=1 -D__ADSPBF526_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF522`
-D__ADSPBF523__=1 -D__ADSPBF52x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF527_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF523`
-D__ADSPBF524__=1 -D__ADSPBF52x__=1	Present when running `easmblkfn -proc ADSP-BF524`

Table 2-3: Feature Macros for Blackfin Processors (Continued)

Macro	*Definition*
-D__ADSPBF52xLP__=1 -D__ADSPBF5xx__=1 -D__ADSPBF526_FAMILY__=1	
-D__ADSPBF525__=1 -D__ADSPBF52x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF527_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF525`
-D__ADSPBF526__=1 -D__ADSPBF52x__=1 -D__ADSPBF52xLP__=1 -D__ADSPBF5xx__=1 -D__ADSPBF526_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF526`
-D__ADSPBF527__=1 -D__ADSPBF52x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF527_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF527`
-D__ADSPBF531__=1 -D__ADSP21531__=1 -D__ADSPBF53x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF533_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF531`
-D__ADSPBF532__=1 -D__ADSP21532__=1 -D__ADSPBF53x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF533_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF532`
-D__ADSPBF533__=1 -D__ADSP21533__=1 -D__ADSPBF53x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF533_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF533`
-D__ADSPBF534__=1 -D__ADSPBF53x__=1 -D__ADSPBF5xx__=1	Present when running `easmblkfn -proc ADSP-BF534`

Table 2-3: Feature Macros for Blackfin Processors (Continued)

Macro	*Definition*
-D__ADSPBF537_FAMILY__=1	
-D__ADSPBF536__=1 -D__ADSPBF53x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF537_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF536`
-D__ADSPBF537__=1 -D__ADSPBF53x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF537_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF537`
-D__ADSPBF538__=1 -D__ADSPBF53x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF538_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF538`
-D__ADSPBF539__=1 -D__ADSPBF53x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF538_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF539`
-D__ADSPBF542__=1 -D__ADSPBF54x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF542`
-D__ADSPBF542M__=1 -D__ADSPBF54x__=1 -D__ADSPBF54xM__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548M_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF542M`
-D__ADSPBF544__=1 -D__ADSPBF54x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF544`
-D__ADSPBF544M__=1 -D__ADSPBF54x__=1 -D__ADSPBF54xM__=1 -D__ADSPBF5xx__=1	Present when running `easmblkfn -proc ADSP-BF544M`

Table 2-3: Feature Macros for Blackfin Processors (Continued)

Macro	*Definition*
-D__ADSPBF548M_FAMILY__=1	
-D__ADSPBF547__=1 -D__ADSPBF54x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF547`
-D__ADSPBF547M__=1 -D__ADSPBF54x__=1 -D__ADSPBF54xM__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548M_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF547M`
-D__ADSPBF548__=1 -D__ADSPBF54x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF548`
-D__ADSPBF548M__=1 -D__ADSPBF54x__=1 -D__ADSPBF54xM__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548M_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF548M`
-D__ADSPBF549__=1 -D__ADSPBF54x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF549`
-D__ADSPBF549M__=1 -D__ADSPBF54x__=1 -D__ADSPBF54xM__=1 -D__ADSPBF5xx__=1 -D__ADSPBF548M_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF549M`
-D__ADSPBF561__=1 -D__ADSPBF56x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF561_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF561`
-D__ADSPBF592A__=1 -D__ADSPBF592__=1	Present when running `easmblkfn -proc ADSP-BF592-A`

Table 2-3: Feature Macros for Blackfin Processors (Continued)

Macro	*Definition*
-D__ADSPBF59x__=1 -D__ADSPBF5xx__=1 -D__ADSPBF592_FAMILY__=1	
-D__ADSPBF606__=1 -D__ADSPBF60x__=1 -D__ADSPBF6xx__=1 -D__ADSPBF609_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF606`
-D__ADSPBF607__=1 -D__ADSPBF60x__=1 -D__ADSPBF6xx__=1 -D__ADSPBF609_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF607`
-D__ADSPBF608__=1 -D__ADSPBF60x__=1 -D__ADSPBF6xx__=1 -D__ADSPBF609_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF608`
-D__ADSPBF609__=1 -D__ADSPBF60x__=1 -D__ADSPBF6xx__=1 -D__ADSPBF609_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF609`
-D__ADSPBF700__=1 -D__ADSPBF70x__=1 -D__ADSPBF7xx__=1 -D__ADSPBF707_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF700`
-D__ADSPBF701__=1 -D__ADSPBF70x__=1 -D__ADSPBF7xx__=1 -D__ADSPBF707_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF701`
-D__ADSPBF702__=1 -D__ADSPBF70x__=1 -D__ADSPBF7xx__=1 -D__ADSPBF707_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF702`
-D__ADSPBF703__=1 -D__ADSPBF70x__=1 -D__ADSPBF7xx__=1	Present when running `easmblkfn -proc ADSP-BF703`

Table 2-3: Feature Macros for Blackfin Processors (Continued)

Macro	*Definition*
-D__ADSPBF707_FAMILY__=1	
-D__ADSPBF704__=1 -D__ADSPBF70x__=1 -D__ADSPBF7xx__=1 -D__ADSPBF707_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF704`
-D__ADSPBF705__=1 -D__ADSPBF70x__=1 -D__ADSPBF7xx__=1 -D__ADSPBF707_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF705`
-D__ADSPBF706__=1 -D__ADSPBF70x__=1 -D__ADSPBF7xx__=1 -D__ADSPBF707_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF706`
-D__ADSPBF707__=1 -D__ADSPBF70x__=1 -D__ADSPBF7xx__=1 -D__ADSPBF707_FAMILY__=1	Present when running `easmblkfn -proc ADSP-BF707`

Table 2-4: Feature Macros for SHARC Processors

Macro	**Definition**
-D_LANGUAGE_ASM=1	Always present
-D__ADSP21000__=1	Always present
-D__NUM_CORES__=1	Defined to the number of cores on the target processor
-D__ADSP21160__=1 -D__2116x__=1 -D__ADSP2116x__=1 -D__ADSP211xx__=1 -D__ADSP21160_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21160`
-D__ADSP21161__=1 -D__2116x__=1 -D__ADSP2116x__=1 -D__ADSP211xx__=1 -D__ADSP21161_FAMILY__=1	Present when running `easm21K -proc ADSP-21161`

Table 2-4: Feature Macros for SHARC Processors (Continued)

Macro	Definition
-D__ADSPSHARC__=0x110	
-D__ADSP21261__=1 -D__2126x__=1 -D__ADSP2126x__=1 -D__ADSP212xx__=1 -D__ADSP21266_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21261`
-D__ADSP21262__=1 -D__2126x__=1 -D__ADSP2126x__=1 -D__ADSP212xx__=1 -D__ADSP21266_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21262`
-D__ADSP21266__=1 -D__2126x__=1 -D__ADSP2126x__=1 -D__ADSP212xx__=1 -D__ADSP21266_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21266`
-D__ADSP21362__=1 -D__2136x__=1 -D__213xx__=1 -D__ADSP2136x__=1 -D__ADSP213xx__=1 -D__ADSP21362_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21362`
-D__ADSP21363__=1 -D__2136x__=1 -D__213xx__=1 -D__ADSP2136x__=1 -D__ADSP213xx__=1 -D__ADSP21362_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21363`
-D__ADSP21364__=1 -D__2136x__=1	Present when running `easm21K -proc ADSP-21364`

Table 2-4: Feature Macros for SHARC Processors (Continued)

Macro	Definition
-D__213xx__=1 -D__ADSP2136x__=1 -D__ADSP213xx__=1 -D__ADSP21362_FAMILY__=1 -D__ADSPSHARC__=0x110	
-D__ADSP21365__=1 -D__2136x__=1 -D__213xx__=1 -D__ADSP2136x__=1 -D__ADSP213xx__=1 -D__ADSP21362_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21365`
-D__ADSP21366__=1 -D__2136x__=1 -D__213xx__=1 -D__ADSP2136x__=1 -D__ADSP213xx__=1 -D__ADSP21362_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21366`
-D__ADSP21367__=1 -D__2136x__=1 -D__213xx__=1 -D__ADSP2136x__=1 -D__ADSP213xx__=1 -D__ADSP21367_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21367`
-D__ADSP21368__=1 -D__2136x__=1 -D__213xx__=1 -D__ADSP2136x__=1 -D__ADSP213xx__=1 -D__ADSP21367_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21368`
-D__ADSP21369__=1 -D__2136x__=1	Present when running `easm21K -proc ADSP-21369`

Table 2-4: Feature Macros for SHARC Processors (Continued)

Macro	Definition
-D__213xx__=1 -D__ADSP2136x__=1 -D__ADSP213xx__=1 -D__ADSP21367_FAMILY__=1 -D__ADSPSHARC__=0x110	
-D__ADSP21371__=1 -D__2137x__=1 -D__213xx__=1 -D__ADSP2137x__=1 -D__ADSP213xx__=1 -D__ADSP21371_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21371`
-D__ADSP21375__=1 -D__2137x__=1 -D__213xx__=1 -D__ADSP2137x__=1 -D__ADSP213xx__=1 -D__ADSP21371_FAMILY__=1 -D__ADSPSHARC__=0x110	Present when running `easm21K -proc ADSP-21375`
-D__ADSP21467__=1 -D__2146x__=1 -D__214xx__=1 -D__ADSP2146x__=1 -D__ADSP214xx__=1 -D__ADSP21469_FAMILY__=1 -D__ADSPSHARC__=0x140	Present when running `easm21K -proc ADSP-21467`
-D__ADSP21469__=1 -D__2146x__=1 -D__214xx__=1 -D__ADSP2146x__=1 -D__ADSP214xx__=1 -D__ADSP21469_FAMILY__=1 -D__ADSPSHARC__=0x140	Present when running `easm21K -proc ADSP-21469`
-D__ADSP21477__=1 -D__2147x__=1	Present when running `easm21K -proc ADSP-21477`

Table 2-4: Feature Macros for SHARC Processors (Continued)

Macro	Definition
-D__214xx__=1 -D__ADSP2147x__=1 -D__ADSP214xx__=1 -D__ADSP21479_FAMILY__=1 -D__ADSPSHARC__=0x147	
-D__ADSP21478__=1 -D__2147x__=1 -D__214xx__=1 -D__ADSP2147x__=1 -D__ADSP214xx__=1 -D__ADSP21479_FAMILY__=1 -D__ADSPSHARC__=0x147	Present when running `easm21K -proc ADSP-21478`
-D__ADSP21479__=1 -D__2147x__=1 -D__214xx__=1 -D__ADSP2147x__=1 -D__ADSP214xx__=1 -D__ADSP21479_FAMILY__=1 -D__ADSPSHARC__=0x147	Present when running `easm21K -proc ADSP-21479`
-D__ADSP21483__=1 -D__2148x__=1 -D__214xx__=1 -D__ADSP2148x__=1 -D__ADSP214xx__=1 -D__ADSP21479_FAMILY__=1 -D__ADSPSHARC__=0x147	Present when running `easm21K -proc ADSP-21483`
-D__ADSP21486__=1 -D__2148x__=1 -D__214xx__=1 -D__ADSP2148x__=1 -D__ADSP214xx__=1 -D__ADSP21479_FAMILY__=1 -D__ADSPSHARC__=0x147	Present when running `easm21K -proc ADSP-21486`
-D__ADSP21487__=1 -D__2148x__=1	Present when running `easm21K -proc ADSP-21487`

Table 2-4: Feature Macros for SHARC Processors (Continued)

Macro	Definition
-D__214xx__=1 -D__ADSP2148x__=1 -D__ADSP214xx__=1 -D__ADSP21479_FAMILY__=1 -D__ADSPSHARC__=0x147	
-D__ADSP21488__=1 -D__2148x__=1 -D__214xx__=1 -D__ADSP2148x__=1 -D__ADSP214xx__=1 -D__ADSP21479_FAMILY__=1 -D__ADSPSHARC__=0x147	Present when running easm21K -proc ADSP-21488
-D__ADSP21489__=1 -D__2148x__=1 -D__214xx__=1 -D__ADSP2148x__=1 -D__ADSP214xx__=1 -D__ADSP21479_FAMILY__=1 -D__ADSPSHARC__=0x147	Present when running easm21K -proc ADSP-21489
-D__NORMAL_WORD_CODE__=1	Present when running easm21K for ADSP-214xx processors, with the -nwc switch
-D__SHORT_WORD_CODE__=1	Present when running easm21K for ADSP-214xx processors, without the -nwc switch
-D__SIMDSHARC__=1	Always present

For .IMPORT headers, the assembler calls the compiler driver with the appropriate processor option, and the compiler sets the machine constants accordingly (and defines -D_LANGUAGE_C=1). This macro is present when used for C compiler calls to specify headers. It replaces -D_LANGUAGE_ASM.

For example,

```
easm21k -proc ADSP-21262  assembly ->
cc21K -proc ADSP-21262

easmblkfn -proc ADSP-BF533 assembly ->
ccblkfn -proc ADSP-BF533
```

NOTE: Use the -verbose switch to verify what macros are defined by default. Refer to Chapter 1 in the *C/C++ Compiler and Library Manual* of the appropriate target processor for more information.

__CCESVERSION__ Predefined Macro

The __CCESVERSION__ predefined macro provides product version information for CCES. The macro allows a preprocessing check to be placed within code and is used to differentiate between releases and updates. This macro applies to all Analog Devices processors.

The preprocessor defines this macro to be an eight-digit hexadecimal representation of the CCES release, in the form 0xMMmmUUPP, where:

- MM is the major release number
- mm is the minor release number
- UU is the update number
- PP is the patch release number

For example, CrossCore Embedded Studio 1.1.0.0 defines __CCESVERSION__ as 0x01010000.

The 0x*MMmmUUPP* information is obtained from the *<install_path>*/System/cces.ini file.

If an unexpected problem occurs while trying to locate cces.ini or while extracting information from the cces.ini file, the __CCESVERSION__ macro is not encoded to the CCES version. Instead, it is set to 0xffffffff, as illustrated in the example below.

Code Example (Error Check):

```
#if __CCESVERSION__ == 0xffffffff
#error Unexpected build problems, unknown tools version
#endif
```

Code Examples (Assembly):

```
#if __CCESVERSION__ == 0x01010000
/* Building with CrossCore_Embedded_Studio 1.1.0 */
.VAR VersionBuildString[] = `Building with CrossCore_Embedded_Studio 1.1.0';
#else
/* Building with unknown tools version */
.VAR VersionBuildString[] = 'Building with unknown CrossCore_Embedded_Studio version?';
#endif
```

Generating Make Dependencies

The assembler can generate *make dependencies* for a file, allowing the IDE and other makefile-based build environments to determine when to rebuild an object file due to changes in the input files. The assembly source file and any files identified in the #include commands, .IMPORT directives, or buffer initializations (in .VAR and .STRUCT directives) constitute the make dependencies for an object file.

When you request make dependencies for the assembly, the assembler produces the dependencies from buffer initializations. The assembler also invokes the preprocessor to determine the make dependency from #include commands, and the compiler to determine the make dependencies from the .IMPORT headers.

For example,

```
easmblkfn -proc ADSP-BF533 -MM main.asm
   "main.doj": ".../Blackfin/include/defBF532.h"
   "main.doj": ".../Blackfin/include/defBF533.h"
   "main.doj": ".../include/def_LPBlackfin.h"
   "main.doj": "main.asm"
   "main.doj": "input_data.dat"
```

The original source file, `main.asm`, is as follows:

```
...
#include "defBF533.h"
...
.GLOBAL input_frame;
.BYTE input_frame[N] = "input_data.dat"; /* load in 256 values from a test file */
...
```

In this case, `defBF533.h` includes `defBF532.h`, which also includes `def_LPBlackfin.h`.

Reading a Listing File

A listing file (`.lst`) is an optional output text file that lists the results of the assembly process. Listing files provide the following information:

- Address - the first column contains the offset from the `.SECTION`'s base address.
- Opcode - the second column contains the hexadecimal opcode that the assembler generates for the line of assembly source.
- Line - the third column contains the line number in the assembly source file.
- Assembly Source - the fourth column contains the assembly source line from the file.

The assembler listing file provides information about the imported C data structures. It tells which imports were used within the program, followed by a more detailed section. It shows the name, total size, and layout with offset for the members. The information appears at the end of the listing. You must specify the -l filename option to produce a listing file.

Assembler Syntax Reference

When developing a source program in assembly language, include preprocessor commands and assembler directives to control the program's processing and assembly. You must follow the assembler rules and syntax conventions to define symbols (identifiers) and expressions, and to use different numeric and comment formats.

Software developers who write assembly programs should be familiar with:

- Assembler Keywords and Symbols
- Assembler Expressions

- Assembler Operators
- Numeric Formats
- Comment Conventions
- Conditional Assembly Directives
- C Struct Support in Assembly Built-In Functions
- Struct References
- Assembler Directives

Assembler Keywords and Symbols

The assembler supports predefined keywords that include register and bit field names, assembly instructions, and assembler directives. The following tables list assembler keywords for supported processors. Although the keywords appear in uppercase, the keywords are case insensitive in the assembler's syntax. For example, the assembler does not differentiate between `MAX` and `max`.

Table 2-5: Blackfin Keywords

	.ALIGN, .ASCII, .ASM_ASSERT, .ASSERT, .BSS, .BYTE, .DATA, .ELIF, .ELSE, .END, .ENDIF, .EXTERN, .FILE, .FILE_ATTR, .GLOBAL, .GLOBL, .IF, .IMPORT, .INC, .INCBIN, .LEFTMARGIN, .LIST, .LIST_DATA, .LIST_DATFILE, .LIST_DEFTAB, .LIST_LOCTAB, .LIST_WRAPDATA, .LONG, .MESSAGE, .NEWPAGE, .NOLIST, .NOLIST_DATA, .NOLIST_DATFILE, .NOLIST_WRAPDATA, .PAGELENGTH, .PAGEWIDTH, .PREVIOUS, .PRIORITY, .REFERENCE, .RETAIN_NAME, .SECTION, .SET, .SHORT, .SIZE, .STRUCT, .TEXT, .TYPE, .VAR, .WEAK
A (Blackfin keywords)	A0, A0.H, A0.L, A0.W, A0.X, A1, A1.H, A1.L, A1.W, A1.X, ABORT, ABS, AC0, AC1, ALIGN16, ALIGN24, ALIGN8, AMNOP, AN, AND, AQ, ASHIFT, ASL, ASR, ASTAT, AV0, AV0S, AV1, AV1S, AZ
B (Blackfin keywords)	B, B0, B0.H, B0.L, B1, B1.H, B1.L, B2, B2.H, B2.L, B3, B3.H, B3.L, BINARY, BIT_XOR_AC, BITCLR, BITMUX, BITPOS, BITSET, BITTGL, BITTST, BP, BREV, BRF, BRT, BXOR, BXORSHIFT, BY, BYTEOP16M, BYTEOP16P, BYTEOP1NS, BYTEOP1P, BYTEOP2P, BYTEOP3P, BYTEPACK, BYTEUNPACK
C (Blackfin keywords)	CALL, CALL.A, CALL.L, CALL.X, CALL.XL, CC, CLI, CLIP, CMUL, CO, CODE, CSYNC, CYCLES, CYCLES2
D (Blackfin keywords)	DATA, DBG, DBGA, DBGAH, DBGAL, DBGCMPLX, DBGHALT, DBGX, DEPOSIT, DISALGNEXCPT, DIVQ, DIVS, DOUBLE32, DOUBLE64, DOUBLEANY, DOZE
E (Blackfin keywords)	EMUEXCPT, EXCL, EXCPT, EXPADJ, EXTRACT
F (Blackfin keywords)	FEXT, FEXTSX, FLUSH, FLUSHINV, FOR, FP, FU
G (Blackfin keywords)	GE, GF, GT
H (Blackfin keywords)	H, HI, HLT
I (Blackfin keywords)	I0, I0.H, I0.L, I1, I1.H, I1.L, I2, I2.H, I2.L, I3, I3.H, I3.L, IDLE, IF, IFLUSH, IH, INTRP, INVALIDATE, IS, ISS2, IU

Table 2-5: Blackfin Keywords (Continued)

J (Blackfin keywords)	JUMP, JUMP.A, JUMP.L, JUMP.S , JUMP.X, JUMP.XL
L (Blackfin keywords)	L, L0, L0.H, L0.L, L1, L1.H, L1.L, L2, L2.H, L2.L, L3, L3.H, L3.L, LB0, LB1, LC0, LC1, LE, LENGTH, LINES, LINK, LJUMP, LMAX, LMIN, LO, LOOP, LOOPLEZ, LOOPZ, LOOP_BEGIN, LOOP_END, LSETUP, LSETUPLEZ, LSETUPZ, LSHIFT, LT, LT0, LT1
M (Blackfin keywords)	M, M0, M0.H, M0.L, M1, M1.H, M1.L, M2, M2.H, M2.L, M3, M3.H, M3.L, MAX, MAXINT, MIN, MNOP, MUNOP
N (Blackfin keywords)	NEG, NO_INIT, NOP, NOT, NS
O (Blackfin keywords)	OFFSETOF, ONES, OR, OUTC
P (Blackfin keywords)	P0, P0.H, P0.L, P1, P1.H, P1.L, P2, P2.H, P2.L, P3, P3.H, P3.L, P4, P4.H, P4.L, P5, P5.H, P5.L, PACK, PC, PREFETCH, PRNT, PRNTX
R (Blackfin keywords)	R, R0, R0.B, R0.H, R0.L, R1, R1.B, R1.H, R1.L, R2, R2.B, R2.H, R2.L, R3, R3.B, R3.H, R3.L, R32, R4, R4.B, R4.H, R4.L, R5, R5.B, R5.H, R5.L, R6, R6.B, R6.H, R6.L, R7, R7.B, R7.H, R7.L, RAISE, RETE, RETI, RETN, RETS, RETX, RND, RND_MOD, RND12, RND20, RNDH, RNDL, ROL, ROR, ROT, ROT_L_AC, ROT_R_AC, RSDL, RTE, RTI, RTN, RTS, RTX, RUNTIME_INIT
S (Blackfin keywords)	S, S2RND, SAA, SAA1H, SAA1L, SAA2H, SAA2L, SAA3H, SAA3L, SAT, SCO, SEARCH, SEQSTAT, SIGN, SIGNBITS, SIZEOF, SKPF, SKPT, SLEEP, SP, SS, SSF, SSF_RND, SSF_RND_HI, SSF_TRUNC, SSF_TRUNC_HI, SSYNC, STI, STRUCT, SU, SYNCEXCL, SYSCFG
T (Blackfin keywords)	T, TESTPOINT, TESTSET, TFU, TH, TL, TST
U (Blackfin keywords)	UNLINK, UNRAISE, UNTIL, USP, UU
V (Blackfin keywords)	V, VALIDATE, VIT_MAX, VS
W (Blackfin keywords)	W, W32
X (Blackfin keywords)	X, XOR
Z (Blackfin keywords)	Z, ZERO_INIT

Table 2-6: SHARC Keywords

. (SHARC keywords)	.ALIGN, .ASM_ASSERT, .BB, .BF, .BYTE, .BYTE2, .BYTE4, .BYTE8, .COMPRESS, .DEF, .DIM, .DM, .EB, .EF, .ELIF, .ELSE, .END_REPEAT, .ENDEF, .ENDIF, .ENDSEG, .EOS, .EXTERN, .FILE, .FILE_ATTR, .FORCECOMPRESS, .GCC_COMPILED, .GLOBAL, .IF, .IMPORT, .INC, .INCBIN, .LEFTMARGIN, .LINE, .LIST, .LIST_DATA, .LIST_DATFILE, .LIST_DEFTAB, .LIST_LOCTAB, .LIST_WRAPDATA, .LN, .MESSAGE, .MESSAGE_DEFAULT, .MESSAGE_POP, .MESSAGE_RESTORE, .MESSAGE_RESTORE_CL, .NEWPAGE, .NO_TRANSFORM_C_RETURN, .NOCOMPRESS, .NOLIST, .NOLIST_DATA, .NOLIST_DATFILE, .NOLIST_WRAPDATA, .PAGELENGTH, .PAGEWIDTH, .PM, .PORT, .PRECISION, .PREVIOUS, .PRIORITY, .REPEAT, .RETAIN_NAME, .ROUND_MINUS, .ROUND_NEAREST, .ROUND_PLUS, .ROUND_ZERO, .SCL, .SECTION, .SEGMENT, .SET, .SIZE, .STRUCT, .SWF_OFF, .SWF_ON, .TAG, .TYPE, .VAL, .VAR, .WEAK
A (SHARC keywords)	ABS, AC, ACT, ADDRESS, AND, ASHIFT, ASTAT, ASTATX, ASTATY, AV

Table 2-6: SHARC Keywords (Continued)

B (SHARC keywords)	B0, B1, B10, B11, B12, B13, B14, B15, B2, B2W, B3, B4, B5, B6, B7, B8, B9, BCLR, BFFWRP, BINARY, BIT, BITDEP, BITEXT, BITREV, BM, BSET, BTGL, BTST, BW, BWSE, BY, BYTE_ADDRESS
C (SHARC keywords)	CA, CACHE, CALL, CH, CI, CJUMP, CL, CLIP, CLR, CODE, COMP, COMPU, COPYSIGN, COS, CURLCNTR
D (SHARC keywords)	DADDR, DATA, DATA64, DB, DM, DM_CACHE, DMA1E, DMA1S, DMA2E, DMA2S, DMADR, DMAONLY, DMBANK1, DMBANK2, DMBANK3, DMWAIT, DO, DOUBLE32, DOUBLE64, DOUBLEANY, DOVL
E (SHARC keywords)	ECE, ELSE, EMUCLK, EMUCLK2, EMUIDLE, EMUN, EQ, EX, EXP, EXP2, EXTERNAL
F (SHARC keywords)	F0, F1, F10, F11, F12, F13, F14, F15, F2, F3, F4, F5, F6, F7, F8, F9, FADDR, FDEP, FEXT, FIX, FLAG0_IN, FLAG1_IN, FLAG2_IN, FLAG3_IN, FLAGS, FLOAT, FLUSH, FMERG, FOR, FOREVER, FPACK, FR, FUNPACK
G (SHARC keywords)	GE, GT
I (SHARC keywords)	I0, I1, I10, I11, I12, I13, I14, I15, I2, I3, I4, I5, I6, I7, I8, I9, I_CACHE, IDLE, IDLE16, IF, IMASK, IMASKP, INVALIDATE, IRPTL
J (SHARC keywords)	JUMP
L (SHARC keywords)	L0, L1, L10, L11, L12, L13, L14, L15, L2, L3, L4, L5, L6, L7, L8, L9, LA, LADDR, LCE, LCNTR, LE, LEFTO, LEFTZ, LENGTH, LINE, LINES, LIRPTL, LOAD, LOG2, LOGB, LOOP, LR, LSHIFT, LT, LW
M (SHARC keywords)	M0, M1, M10, M11, M12, M13, M14, M15, M2, M3, M4, M5, M6, M7, M8, M9, MANT, MAX, MIN, MMASK, MODE1, MODE1STK, MODE2, MR0B, MR0F, MR1B, MR1F, MR2B, MR2F, MRB, MRF, MS, MV
N (SHARC keywords)	NE, NO_INIT, NOFO, NOFZ, NOP, NOPSPECIAL, NOT, NU, NW
O (SHARC keywords)	OFFSETOF, OR
P (SHARC keywords)	P20, P24, P32, P40, PACK, PAGE, PASS, PC, PCSTK, PCSTKP, PEX, PEY, PM, PMADR, PMBANK1, PM_CACHE, PMCODE, PMDAE, PMDAS, PMDATA, PMWAIT, POP, POVL0, POVL1, PSA1E, PSA1S, PSA2E, PSA2S, PSA3E, PSA3S, PSA4E, PSA4S, PUSH, PX, PX1, PX2
R (SHARC keywords)	R0, R1, R10, R11, R12, R13, R14, R15, R2, R3, R4, R5, R6, R7, R8, R9, READ, RECIPS, RFRAME, RND, ROT, RSQRTS, RTI, RTS, RUNTIME_INIT
S (SHARC keywords)	S0, S1, S10, S11, S12, S13, S14, S15, S2, S3, S4, S5, S6, S7, S8, S9, SAT, SCALB, SE, SET, SF, SF0, SF1, SF10, SF11, SF12, SF13, SF14, SF15, SF2, SF3, SF4, SF5, SF6, SF7, SF8, SF9, SI, SIN, SIZEOF, SQR, SR, SSF, SSFR, SSI, SSIR, ST, STEP, STKY, STKYX, STKYY, STRUCT, STS, SUF, SUFR, SUI, SUIR, SV, SW, SWSE, SYNC, SZ
T (SHARC keywords)	TCOUNT, TF, TGL, TPERIOD, TRUE, TRUNC, TST
U (SHARC keywords)	UF, UI, UNPACK, UNTIL, UR, USF, USFR, USI, USIR, USTAT1, USTAT2, USTAT3, USTAT4, UUF, UUFR, UUI, UUIR
W (SHARC keywords)	W2B, WITH, WORD_ADDRESS, WRITEBACK
X (SHARC keywords)	XOR
Z (SHARC keywords)	ZERO_INIT

Extend these sets of keywords with symbols that declare sections, variables, constants, and address labels. When defining symbols in assembly source code, follow these conventions:

- Define symbols that are unique within the file in which they are declared. If you use a symbol in more than one file, use the `.GLOBAL` assembly directive to export the symbol from the file in which it is defined. Then use the `.EXTERN` assembly directive to import the symbol into other files.

  If you use a symbol in more than one file, use the `.GLOBAL` assembly directive to export the symbol from the file in which it is defined. Then use the `.EXTERN` assembly directive to import the symbol into other files.

- Begin symbols with alphabetic characters.

  Symbols can use alphabetic characters (`A-Z` and `a-z`), digits (`0-9`), and the special characters " `$`" and " `_`" (dollar sign and underscore) as well as " `.`" (dot).

  Symbols are case sensitive; so `input_addr` and `INPUT_ADDR` define unique variables.

  The dot, point, or period " `.`" as the first character of a symbol triggers special behavior in the IDE. A symbol with a " `.`" as the first character cannot have a digit as the second character. Such symbols will not appear in the debugger. A symbol name in which the first two characters are dots will not appear even in the symbol table of the object.

  The compiler and run-time libraries prepend " `_`" to avoid using symbols in the user namespace that begin with an alphabetic character.

- Do not use a reserved keyword to define a symbol.

- Match source and LDF sections' symbols. Ensure that `.SECTION` name symbols do not conflict with the linker's keywords in the `.ldf` file. The linker uses sections' name symbols to place code and data in the processor's memory. For details, see the *Linker and Utilities Manual*. Ensure that `.SECTION` name symbols do not begin with the " `.`" (dot).

  Ensure that `.SECTION` name symbols do not conflict with the linker’s keywords in the `.ldf` file. The linker uses sections’ name symbols to place code and data in the processor’s memory. For details, see the *Linker and Utilities Manual*.

  Ensure that `.SECTION` name symbols do not begin with the “.” (dot).

- Terminate the definition of address label symbols with a colon (`:`).

- The reserved word list for processors includes some keywords with commonly used spellings; therefore, ensure correct syntax spelling.

Address label symbols may appear at the beginning of an instruction line or standalone on the preceding line.

The following disassociated lines of code demonstrate symbol usage.

```
.BYTE2 xoperand;         /* xoperand is a 16-bit variable */
.BYTE4 input_array[10];  /* input_array is a 32-bit wide  */
                         /* data buffer with 10 elements  */
sub_routine_1:           /* sub_routine_1 is a label      */
.SECTION kernel;         /* kernel is a section name      */
```

Assembler Expressions

The assembler can evaluate simple expressions in source code. The assembler supports two types of expressions: constant expressions and symbolic expressions.

Constant Expressions

A constant expression is acceptable where a numeric value is expected in an assembly instruction or in a preprocessor command. Constant expressions contain an arithmetic or logical operation on two or more numeric constants.

For example,

```
2.9e-5 + 1.29
(128 - 48) / 3
0x55 & 0x0F
0x0f7.6r - 0.8r
```

For information about fraction type support, refer to Fractional Type Support.

Symbolic Expressions

Symbolic expressions contain symbols, whose values may not be known until link-time. For example,

```
data/8
(data_buffer1 + data_buffer2) & 0xF
strtup + 2
data_buffer1 + LENGTH(data_buffer2)*2
```

Symbols in this type of expression are data variables, data buffers, and program labels. In the first three examples above, the symbol name represents the address of the symbol. The fourth example combines that meaning of a symbol with a use of the length operator (see the ***Special Assembler Operators*** table in Assembler Operators).

Assembler Operators

The ***Operator Precedence*** table lists the assembler's numeric and bitwise operators used in constant expressions and address expressions. These operators are listed in group order from highest precedence to lowest precedence. Operators with the highest precedence are evaluated first. When two operators have the same precedence, the assembler evaluates the left-most operator first. Relational operators are supported only in relational expressions in conditional assembly, as described in Conditional Assembly Directives.

Table 2-7: Operator Precedence

Operator	*Description*	*Designation*
(*expression*)	*expression* in parentheses evaluates first	Parentheses
~	Ones complement	Tilde
-	Unary minus	Minus
*	Multiply	Asterisk
/	Divide	Slash

Table 2-7: Operator Precedence (Continued)

Operator	*Description*	*Designation*
%	Modulus	Percentage
+ -	Addition Subtraction	Plus Minus
<< >>	Shift left Shift right	
&	Bitwise AND	
|	Bitwise inclusive OR	
^	Bitwise exclusive OR	

NOTE: If right-shifting a negative value, ones are shifted in from the MSB, which preserves the sign bit.

The assembler also supports special operators. The ***Special Assembler Operators*** table lists and describes special operators used in constant and address expressions.

Table 2-8: Special Assembler Operators

Operator	*Description*
BITPOS(*constant*)	Bit position (Blackfin processors only).
BYTE_ADDRESS(*expression*) WORD_ADDRESS(*expression*)	Converts symbol or expression to the byte-addressed or word-addressed alias space, respectively. (SHARC ADSP-215xx and ADSP-SC5xx processors only).
HI(*expression*) LO(*expression*)	Extracts the most significant 16 bits of expression. Extracts the least significant 16 bits of expression. **NOTE**: Used with the Blackfin assembler only. The expression in the HI and LO operators can be either symbolic or constant.
LENGTH(*symbol*)	Length of *symbol* in number of elements (in a buffer/array).

The LENGTH operator can be used with external symbols-apply it to symbols that are defined in other sections as .GLOBAL symbols.

Blackfin Processor Example

The following example demonstrates how Blackfin assembler operators are used to load the length and address information into registers.

```
#define n 20
...
.SECTION data1;           /* data section */
.VAR real_data [n];       /* n=number of input samples */

.SECTION program;         /* code section */
  P0.L = real_data;
  P0.H = real_data;
```

```
  P1=LENGTH(real_data);   /* buffer's length */
  LOOP loop1 LC0=P1;
  LOOP_BEGIN loop1;
  R0=[P0++];              /* get next sample */
  ...
LOOP_END loop1;
```

The code fragment above initializes `P0` and `P1` to the base address and length, respectively, of the `real_data` buffer. The loop is executed 20 times.

The `BITPOS()` operator takes a bit constant (with one bit set) and returns the position of the bit. Therefore, `BITPOS(0x10)` would return 4 and `BITPOS(0x80)` would return 7. For example,

```
#define DLAB 0x80
#define EPS 0x10
R0 = DLAB | EPS (z);
cc = BITSET (R0, BITPOS(DLAB));
```

SHARC Processor Example

The following code example determines the base address and length of the `real_data` circular buffer. The buffer's length value (contained in `L5)` determines when addressing wraps around to the top of the buffer (when setting up circular buffers in SHARC processors). For further information on circular buffers, refer to the *Hardware Reference* of the target processor.

```
.SECTION/DM seg_dmda;       /* data segment */
.VAR real_data[n];          /* n=number of input samples */
...

.SECTION/PM seg_pmco;       /* code segment */
   B5=real_data;            /* buffer base address */
                            /* I5 loads automatically */
   L5=LENGTH(real_data);    /* buffer's length */
   M6=1;                    /* post-modify I5 by 1 */
   LCNTR=LENGTH(real_data)
   .DO loopend UNTIL LCE;
                            /* loop counter=buffer's length */
   F0=DM(I5,M6);            /* get next sample */
...
loopend:
...
```

NOTE: Although the SHARC assembler accepts the source code written with the legacy `@` operator, it is recommended to use `LENGTH()` in place of `@`.

SHARC ADSP-215xx and ADSP-SC5xx Processor Byte/Word-Addressing Example

The following code example shows how the `BYTE_ADDRESS()` and `WORD_ADDRESS()` assembly operators can be used to convert addresses between the byte- and word-addressed aliased memory spaces.

```
.SECTION seg_dmda;              /* data segment */
.BYTE a.[4];                    /* byte-addressed 32-bit value */
p.:
```

```
.VAR = WORD_ADDRESS(a.);          /* initialize to word-address    */
                                  /* equivalent of byte-address a. */
.p..end:
extern _b;                        /* word-address _b defined in another */
                                  /* translation unit                   */
.SECTION/SW seg_swco;             /* code segment */
   R0=DM(BYTE_ADDRESS(_b)) (BW); /* load from address _b converted */
                                  /* to byte-address alias space    */
```

NOTE: The linker will report an error if the `BYTE_ADDRESS()` or `WORD_ADDRESS()` assembly operators are used to try to convert an address for which there is no equivalent in the target address space. This includes use of `WORD_ADDRESS()` on a byte-address that is not aligned on a 32-bit boundary. Use of the operators on an address that is already in the target address space leaves the address unchanged.

Numeric Formats

Depending on the processor architectures, the assemblers support binary, decimal, hexadecimal, floating-point, and fractional numeric formats (bases) within expressions and assembly instructions. The ***Numeric Formats*** table describes the notation conventions used by the assembler to distinguish between numeric formats.

Table 2-9: Numeric Formats

Convention	*Description*
`0x`*number*	The "`0x`" prefix indicates a hexadecimal number
`B#`*number* `b#`*number*	The "`B#`" or "`b#`" prefix indicates a binary number
number.number[e {+/-} number]	Entry for floating-point number
number	No prefix and no decimal point indicates a decimal number
number`r`	The "`r`" suffix indicates a fractional number

NOTE: Due to the support for `b#` and `B#` binary notation, the preprocessor stringization functionality is turned off, by default, to avoid possible undesired stringization. For more information, refer to the processor's `#(Argument)` and -stringize command-line switches, and the assembler's -flags-pp -opt1[, -opt2...] command-line switch.

Representation of Constants in Blackfin

The Blackfin assembler keeps an internal 32-bit signed representation of all constant values. Keep this in mind when working with immediate values. The immediate value is used by the assembler to determine the instruction length (16, 32 or 64 bit). The assembler selects the smallest opcode that can accommodate the immediate value.

Blackfin processors preceding Blackfin+ did not support 64-bit instructions with 32-bit immediates. When targeting such a processor, if there is no opcode that can accommodate the value, semantic error ea5003 is reported.

Examples:

```
R0 = -64;         /* 16-bit instruction: -64 fits into 7-bit immediate value */
R0 = 0xBF;        /* 32-bit instruction: 191 fits into 16-bit immediate value */
R0 = 0xFFBF;      /* 64-bit instruction: 65471 doesn't fit into 16-bit immediate */
R0 = 0xFFFFFFBF; /* 32-bit instruction: -65 fits into 16 bit immediate value */
R0 = 0x8000;      /* 64-bit instruction: 32768 doesn't fit into 16-bit immediate */
```

Fractional Type Support

Fractional (fract) constants are specially marked floating-point constants to be represented in fixed-point format. A fract constant uses the floating-point representation with a trailing "`r`", where `r` stands for `fract`.

The legal range is [`-1` `1`). This means the values must be greater than or equal to `-1` and less than `1`. Fracts are represented as signed values.

For example,

```
.VAR myFracts[] = {0.5r, -0.5e-4r, -0.25e-3r, 0.875r};
      /* Constants are examples of legal fracts */

.VAR OutOfRangeFract = 1.5r;
      /* [Error ] Fract constant '1.5r' is out of range.
         Fract constants must be greater than or equal to -1
         and less than 1. */
```

NOTE: In Blackfin processors, the 1.15 fract format is the default. Use a `/R32` qualifier (in `.BYTE4/R32` or `.VAR/R32`) to support 32-bit initialization for use with 1.31 fracts.

1.31 Fracts

Fracts supported by Analog Devices processors use the 1.31 format, which means a sign bit and "31 bits of fraction". This is `-1` to `+1 - 2 -31`. For example, 1.31 maps the constant `0.5r` to the bit pattern `0x40000000`.

The conversion formula used by processors to convert from floating-point format to fixed-point format uses a scale factor of 31.

For example,

```
.VAR/R32 myFract = 0.5r;
   // Fract output for 0.5r is 0x4000 0000
   // sign bit + 31 bits
   //  0100 0000 0000 0000  0000 0000 0000 0000
   //    4    0    0    0      0    0    0    0  = 0x4000 0000 = .5r


.VAR/R32 myFract = -1.0r;
   // Fract output for -1.0r is 0x8000 0000
   // sign bit + 31 bits
   //  1000 0000 0000 0000  0000 0000 0000 0000
   //    8     0     0     0      0     0     0     0  = 0x8000 0000 = -1.0r


.VAR/R32 myFract = -1.72471041E-03r;
```

```
// Fract output for -1.72471041E-03 is 0xFFC77C15
// sign bit + 31 bits
//  1111 1111 1100 0111 0111 1100 0001 0101
//    F    F    C    7   7    C     1    5
```

1.0r Special Case

`1.0r` is out-of-the-range `fract`. Specify `0x7FFF FFFF` for the closest approximation of `1.0r` within the 1.31 representation.

Fractional Arithmetic

The assembler provides support for arithmetic expressions using operations on fractional constants, consistent with the support for other numeric types in constant expressions, as described in Assembler Expressions.

The internal (intermediate) representation for expression evaluation is a double floating-point value. Fract range checking is deferred until the expression is evaluated. For example,

```
#define fromSomewhereElse   0.875r
.SECTION data1;
.VAR localOne = fromSomewhereElse + 0.005r;
                    // Result .88r is within the legal range
.VAR xyz = 1.5r -0.9r;
                    // Result .6r is within the legal range
.VAR abc = 1.5r;    // Error: 1.5r out of range
```

Mixed Type Arithmetic

The assembler does not support arithmetic between fracts and integers. For example,

```
.SECTION data1;
.VAR myFract = 1 - 0.5r;
     [Error ea1998] "fract.asm":2 User Error: Illegal
      mixing of types in expression.
```

Comment Conventions

The assemblers support C and C++ style formats for inserting comments in assembly sources. The assemblers do not support nested comments. The ***Comment Conventions*** table lists and describes assembler comment conventions.

Table 2-10: Comment Conventions

Convention	*Description*
`/* comment */`	A "`/* */`" string encloses a multiple-line comment
`// comment`	A pair of slashes "`//`" begin a single-line comment

Conditional Assembly Directives

Conditional assembly directives are used for evaluation of assembly-time constants using relational expressions. The expressions may include relational and logical operations.

The conditional assembly directives include:

- `.IF` *`constant-relational-expression`*`;`
- `.ELIF` *`constant-relational-expression`*`;`
- `.ELSE;`
- `.ENDIF;`

Conditional assembly blocks begin with an `.IF` directive and end with an `.ENDIF` directive. The ***Relational Operators for Conditional Assembly*** table shows examples of conditional directives.

Table 2-11: Relational Operators for Conditional Assembly

Operator	*Purpose*	*Conditional Directive Examples*
`!`	Not	`.IF !0;`
`>`	Greater than	`.IF ( SIZEOF(myStruct) > 16 );`
`>=`	Greater than or equal to	`.IF ( SIZEOF(myStruct) >= 16 );`
`<`	Less than	`.IF ( SIZEOF(myStruct) < 16 );`
`<=`	Less than or equal to	`.IF ( SIZEOF(myStruct) <= 16 );`
`==`	Equality	`.IF ( 8 == SIZEOF(myStruct) );`
`!=`	Not equal	`.IF ( 8 != SIZEOF(myStruct) );`
`||`	Logical OR	`.IF (2 !=4 ) || (5 == 5);`
`&&`	Logical AND	`.IF (SIZEOF(char) == 2 && SIZEOF(int) == 4);`

Optionally, any number of `.ELIF` directives and a final `.ELSE` directive may appear within a pair of `.IF` and `.ENDIF` directives. The conditional directives are each terminated with a semi-colon "`;`", same as other assembler directives. Conditional directives do not have to appear alone on a line. These directives are in addition to the C-style `#if`, `#elif`, `#else`, and `#endif` preprocessing directives.

The `.IF` conditional assembly directive must be used to query about C structs in assembly using the `SIZEOF()` and/or `OFFSETOF()` built-in functions. These built-ins are evaluated at assembly time, so they cannot appear in expressions in `#if` preprocessor directives.

In addition, the `SIZEOF()` and `OFFSETOF()` built-in functions (see C Struct Support in Assembly Built-In Functions) can be used in relational expressions. Different code sequences can be included based on the result of the expression.

For example, `SIZEOF(struct/typedef/C_base_type)` is permitted.

The assembler supports nested conditional directives. The outer conditional result propagates to the inner condition, just as it does in C preprocessing.

Assembler directives are distinct from preprocessor directives, as follows:

- The `#` directives are evaluated during preprocessing by the preprocessor. Therefore, preprocessor `#if` directives cannot use assembler built-ins (see C Struct Support in Assembly Built-In Functions).
- The conditional assembly directives are processed by the assembler in a later pass. Therefore, you are able to write a relational or logical expression whose value depends on the value of a `#define`.

  For example,

```
.IF tryit == 2;
    <some code>
.ELIF tryit >= 3;
    <some more code>
.ELSE;
    <some more code>
.ENDIF;
```

- If you have `#define tryit 2`, the code `<some code>` is assembled, and `<some more code>` is not assembled.
- There are no parallel assembler directives for C-style directives `#define`, `#include`, `#ifdef`, `#if defined(name)`, `#ifndef`, and so on.

C Struct Support in Assembly Built-In Functions

The assemblers support built-in functions that enable you to pass information obtained from the imported C struct layouts. The assemblers currently support two built-in functions: `OFFSETOF()` and `SIZEOF()`.

OFFSETOF Built-In Function

The `OFFSETOF()` built-in function is used to calculate the offset of a specified member from the beginning of its parent data structure.

```
OFFSETOF(struct/typedef, memberName);
```

where:

`struct/typedef` - a struct `VAR` or a `typedef` can be supplied as the first argument

`memberName` - a member name within the `struct` or `typedef` (second argument)

NOTE: For SHARC processors, `OFFSETOF()` units are in words. For Blackfin processors, `OFFSETOF()` units are in bytes

SIZEOF Built-In Function

The `SIZEOF()` built-in function returns the amount of storage associated with an imported C struct or data member. It provides functionality similar to its C counterpart.

```
SIZEOF(struct/typedef/C_base_type);
```

where:

The `SIZEOF()` function takes a symbolic reference as its single argument. A symbolic reference is a name followed by none or several qualifiers to members.

The `SIZEOF()` function gives the amount of storage associated with:

- An aggregate type (structure)
- A C base type (`int`, `char`, and so on)
- A member of a structure (any type)

For example (Blackfin processor code),

```
.IMPORT "Celebrity.h";
.EXTERN STRUCT Celebrity StNick;
L3 = SIZEOF(Celebrity);            // typedef
L3 = SIZEOF(StNick);               // struct var of typedef Celebrity
L3 = SIZEOF(char);                 // C built-in type
L3 = SIZEOF(StNick->Town);      // member of a struct var
L3 = SIZEOF(Celebrity->Town);   // member of a struct typedef
```

NOTE: The `SIZEOF()` built-in function returns the size in the units appropriate for its processor. For SHARC processors, units are in words. For Blackfin processors, units are in bytes.

When applied to a structure type or variable, `SIZEOF()` returns the actual size, which may include padding bytes inserted for alignment. When applied to a statically dimensioned array, `SIZEOF()` returns the size of the entire array.

Struct References

A reference to a `struct VAR` provides an absolute address. For a fully qualified reference to a member, the address is offset to the correct location within the struct. The assembler syntax for `struct` references is "`->`".

The following example references the address of `Member5` located within `myStruct`.

```
myStruct->Member5
```

If the struct layout changes, there is no need to change the reference. The assembler recalculates the offset when the source is reassembled with the updated header.

Nested `struct` references are supported. For example,

```
myStruct->nestedRef->AnotherMember
```

NOTE: Unlike `struct` members in C, `struct` members in the assembler are always referenced with "->" (not ".") because "." is a legal character in identifiers in assembly and is not available as a `struct` reference. The "->" does not indicate pointer dereferencing as it does in C.

References within nested structures are permitted. A nested struct definition can be provided in a single reference in assembly code, and a nested `struct` via a pointer type requires more than one instruction. Use the `OFFSETOF()` built-in function to avoid hard-coded offsets that may become invalid if the struct layout changes in the future.

Following are two nested `struct` examples for `.IMPORT "CHeaderFile.h".`

Example 1: Nested Reference Within the Struct Definition with Appropriate C Declarations

C Code

```
struct Location {
      char Town[16];
      char State[16];
};

struct myStructTag {
      int field1;
      struct Location NestedOne;
};
```

Assembly Code (for Blackfin Processors)

```
.EXTERN STRUCT myStructTag _myStruct;
P3.L = LO(_myStruct->NestedOne->State);
P3.H = HI(_myStruct->NestedOne->State);
```

Example 2: Nested Reference When Nested via a Pointer with Appropriate C Declarations

When nested via a pointer, `myStructTagWithPtr` (which has `pNestedOne`) uses pointer register offset instructions.

C Code:

```
// from C header
struct Location {
      char Town[16];
      char State[16];
};

struct myStructTagWithPtr {
      int field1;
      struct Location *pNestedOne;
};
```

Assembly Code (for Blackfin Processors):

```
// in assembly file
.EXTERN STRUCT myStructTagWithPtr _myStructWithPtr;
```

```
P1.L = LO(_myStructWithPtr->pNestedOne);
P1.H = HI(_myStructWithPtr->pNestedOne);
P0   = [P1 + OFFSETOF(Location,State)];
```

Assembler Directives

Directives in an assembly source file control the assembly process. Unlike assembly instructions, directives do not produce opcodes during assembly. Use the following general syntax for assembler: directives:

```
.directive[/qualifiers|arguments];
```

Each assembler directive starts with a period (`.`) and ends with a semicolon (`;`). Some directives take qualifiers and arguments. A directive's qualifier immediately follows the directive and is separated by a slash (`/`); arguments follow qualifiers. Assembler directives can be uppercase or lowercase; uppercase distinguishes directives from other symbols in your source code.

The ***Assembler Directive Summary*** table lists all currently supported assembler directives. A description of each directive appears in the following sections.

Table 2-12: Assembler Directive Summary

Directive	*Description*
`.ALIGN`	Specifies an alignment requirement for data or code. Refer to .ALIGN, Specify an Address Alignment.
`.ASCII`	Initializes ASCII strings. Refer to .ASCII. **NOTE:** Blackfin processors only.
`.BSS`	Equivalent to `.SECTION/zero_init bsz;`. Refer to .SECTION, Declare a Memory Section. **NOTE:** Blackfin processors only.
`.BYTE .BYTE2 .BYTE4`	Defines and initializes one-, two-, and four-byte data objects, respectively. Refer to .BYTE, Declare a Byte Data Variable or Buffer. **NOTE:** Blackfin processors only.
`.COMPRESS`	Starts compression. Refer to .COMPRESS, Start Compression. **NOTE:** ADSP-214xx SHARC processors only.
`.DATA`	Equivalent to `.SECTION data1;`. Refer to .SECTION, Declare a Memory Section. **NOTE:** Blackfin processors only.
`.ELSE`	Conditional assembly directive. Refer to Conditional Assembly Directives.
`.ENDIF`	Conditional assembly directive. Refer to Conditional Assembly Directives.
`.EXTERN`	Allows reference to a global symbol. Refer to .EXTERN, Refer to a Globally Available Symbol.
`.EXTERN STRUCT`	Allows reference to a global symbol (`struct`) that was defined in another file. Refer to .EXTERN STRUCT, Refer to a Struct Defined Elsewhere.

Table 2-12: Assembler Directive Summary (Continued)

Directive	*Description*
`.FILE`	Overrides `filename` given on the command line. Refer to .FILE, Override the Name of a Source File.
`.FILE_ATTR`	Creates a file attribute in the generated object file. Refer to .FILE_ATTR, Create an Attribute in the Object File.
`.FORCECOMPRESS`	Compresses the next instruction. Refer to .FORCECOMPRESS, Compress the Next Instruction. **NOTE:** ADSP-214xx SHARC processors only.
`.GLOBAL`	Changes a symbol's scope from local to global. Refer to .GLOBAL, Make a Symbol Available Globally.
`.GLOBL`	Equivalent to `.GLOBAL`. Refer to .GLOBAL, Make a Symbol Available Globally. **NOTE:** Blackfin processors only.
`.IF`	Conditional assembly directive. Refer to Conditional Assembly Directives.
`.IMPORT`	Provides the assembler with structure layout (C struct) information. Refer to .IMPORT, Provide Structure Layout Information.
`.INC/BINARY`	Includes the content of a file at the current location. Refer to .INC/BINARY, Include Contents of a File.
`.INCBIN`	Equivalent to `.INC/BINARY`. Refer to .INC/BINARY, Include Contents of a File. **NOTE:** Blackfin processors only.
`.LEFTMARGIN`	Defines the width of the left margin of a listing. Refer to .LEFTMARGIN, Set the Margin Width of a Listing File.
`.LIST/.NOLIST`	Starts listing of source lines. Refer to .LIST/.NOLIST, Listing Source Lines and Opcodes.
`.LIST_DATA`	Starts listing of data opcodes. Refer to .LIST_DATA/.NOLIST_DATA, Listing Data Opcodes.
`.LIST_DATFILE`	Starts listing of data initialization files. Refer to .LIST_DATFILE/.NOLIST_DATFILE, List Data Init Files.
`.LIST_DEFTAB`	Sets the default tab width for listings. Refer to .LIST_DEFTAB/.LIST_LOCTAB, Set Tab Widths for Listings.
`.LIST_LOCTAB`	Sets the local tab width for listings. Refer to .LIST_DEFTAB/.LIST_LOCTAB, Set Tab Widths for Listings.
`.LIST_WRAPDATA`	Starts wrapping opcodes that don't fit listing column. Refer to .LIST_WRAPDATA/.NOLIST_WRAPDATA.
`.LONG`	Supports four-byte data initializer lists for GNU compatibility. Refer to .LONG, Define and Initialize 4-Byte Data Objects. **NOTE:** Blackfin processors only.
`.MESSAGE`	Alters the severity of an error, warning or informational message generated by the assembler. Refer to .MESSAGE, Alter the Severity of an Assembler Message.
`.NEWPAGE`	Inserts a page break in a listing. Refer to .NEWPAGE, Insert a Page Break in a Listing File.

Table 2-12: Assembler Directive Summary (Continued)

Directive	*Description*
.NOCOMPRESS	Terminates compression. Refer to .NOCOMPRESS, Terminate Compression. **NOTE:** ADSP-214xx SHARC processors only.
.NOLIST	Stops listing of source lines. Refer to .LIST_DATA/.NOLIST_DATA, Listing Data Opcodes.
.NOLIST_DATA	Stops listing of data opcodes. Refer to .NEWPAGE, Insert a Page Break in a Listing File.
.NOLIST_DATFILE	Stops listing of data initialization files. Refer to .LIST_DATFILE/.NOLIST_DATFILE, List Data Init Files.
.NOLIST_WRAPDATA	Stops wrapping opcodes that do not fit listing column. Refer to .LIST_WRAPDATA/.NOLIST_WRAPDATA.
.PAGELENGTH	Defines the length of a listing page. Refer to .PAGELENGTH, Set the Page Length of a Listing File.
.PAGEWIDTH	Defines the width of a listing page. Refer to .PAGEWIDTH, Set the Page Width of a Listing File.
.PORT	Legacy directive. Declares a memory-mapped I/O port. Refer to .PORT, Legacy Directive. **NOTE:** SHARC processors only.
.PRECISION	Defines the number of significant bits in a floating-point value. Refer to .PRECISION, Select Floating-Point Precision. **NOTE:** SHARC processors only.
.PREVIOUS	Reverts to a previously described .SECTION. Refer to .PREVIOUS, Revert to the Previously Defined Section.
.PRIORITY	Allows prioritized symbol mapping in the linker. Refer to PRIORITY, Allow Prioritized Symbol Mapping in Linker.
.REFERENCE	Provides better information in an X-REF file. Refer to .REFERENCE, Provide Better Info in an X-REF File. **NOTE:** Blackfin processors only.
.RETAIN_NAME	Stops the linker from eliminating a symbol. Refer to .RETAIN_NAME, Stop Linker from Eliminating Symbol.
.ROUND_NEAREST	Specifies the round-to-nearest mode. Refer to .ROUND_, Select Floating-Point Rounding. **NOTE:** SHARC processors only.
.ROUND_MINUS	Specifies the round-to-negative infinity mode. Refer to .ROUND_, Select Floating-Point Rounding. **NOTE:** SHARC processors only.
.ROUND_PLUS	Specifies the round-to-positive infinity mode. Refer to .ROUND_, Select Floating-Point Rounding. **NOTE:** SHARC processors only.
.ROUND_ZERO	Specifies the round-to-zero mode. Refer to .ROUND_, Select Floating-Point Rounding.

Table 2-12: Assembler Directive Summary (Continued)

Directive	*Description*
	NOTE: SHARC processors only.
`.SECTION`	Marks the beginning of a section. Refer to .SECTION, Declare a Memory Section.
`.SET`	Sets symbolic aliases. Refer to .SET, Set a Symbolic Alias.
`.SHORT`	Supports two-byte data initializer lists for GNU compatibility. Refer to .SHORT, Defines and Initializes 2-Byte Data Objects. **NOTE:** Blackfin processors only.
`.STRUCT`	Defines and initializes data objects based on C `typedefs` from `.IMPORT` C header files. Refer to .STRUCT, Create a Struct Variable.
`.TEXT`	Equivalent to `.SECTION program;`. Refer to .SECTION, Declare a Memory Section. **NOTE:** Blackfin processors only.
`.TYPE`	Changes the default data type of a symbol; used by C compiler. Refer to .TYPE, Change Symbol Type.
`.VAR`	Defines and initializes 32-bit data objects. Refer to .VAR, Declare a Data Variable or Buffer.
`.WEAK`	Creates a weak definition or reference. Refer to .WEAK, Weak Symbol Definition and Reference.

.ALIGN, Specify an Address Alignment

The `.ALIGN` directive forces the address alignment of an instruction or data item. The assembler sets the alignment of the section to match the largest alignment requirement specified in the section and inserts padding at each alignment location to ensure that the following item has the proper offset from the start of the section to maintain the requested alignment. The linker honors the alignment specified by the assembler when placing the section in memory, thus guaranteeing the integrity of the alignment of each element aligned with a `.ALIGN` directive.

You also can use the `INPUT_SECTION_ALIGN(#number)` LDF command (in the `.ldf` file) to force all of the following input sections to the specified alignment. Refer to the *Linker and Utilities Manual* for more information on section alignment.

Syntax:

`.ALIGN` *`expression`*;

where

`expression` - evaluates to an integer. It specifies an alignment requirement; its value must be a power of 2. When aligning a data item or instruction, the assembler adjusts the address of the current location counter to the next address that can be divided by the value of *`expression`*, with no remainder. The expression set to 0 or 1 signifies no address alignment requirement.

The linker stops allocating padding for symbols aligned by 16 or more.

NOTE: In the absence of the `.ALIGN` directive, the default address alignment is 1.

Example:

In the following example, the assembler sets the alignment of the section to 4 to match the value specified in the second alignment directive. This satisfies the first alignment directive as well, since any item alignment on an address multiple of 4 is also aligned on a multiple of 2. If the target is a byte-addressed processor, such as Blackfin, there is no padding inserted between "single" and "samples" since .VAR creates a four-byte word of storage. If the target is a processor on which the .VAR directive reserves a one-address unit, such as SHARC, three words of padding follow "single" in the section produced by the assembler.

```
...
.ALIGN 1;        /* no alignment requirement */
...
.SECTION data1;
.ALIGN 2;
.VAR single;    /* aligns the data item on the word boundary,
                   at the location with the address value that can
                   be evenly divided by 2 */
.ALIGN 4;
.VAR samples1[100]="data1.dat";
                /* aligns the first data item on the double-word
                   boundary, at the location with the address value
                   that can be evenly divided by 4; advances other
                   data items consecutively */
```

NOTE: The Blackfin assembler accepts .BYTE, .BYTE2, .BYTE4, and .VAR.

.ASCII

NOTE: Used with the Blackfin processors only.

The .ASCII directive initializes a data location with one or more characters from a double-quoted ASCII string. This is equivalent to the .BYTE directive. Note that the syntax differs from the .BYTE directive as follows:

- There is no "=" sign
- The string is enclosed in double-quotes, not single quotes

Syntax:

```
.ASCII "string";
```

Example:

```
.SECTION data1;

ASCII_String:
.TYPE ASCII_String,STT_OBJECT;
     .ASCII "ABCD";
.ASCII_String.end:
```

```
Byte_String:
.TYPE Byte_String,STT_OBJECT;
     .Byte = `ABCD';
.Byte_String.end:
```

.BYTE, Declare a Byte Data Variable or Buffer

NOTE: Used with the Blackfin processors only.

The `.BYTE`, `.BYTE2`, and `.BYTE4` directives declare and optionally initialize one-, two-, and four-byte data objects, respectively. Note that the `.BYTE4` directive performs the same function as the `.VAR` directive.

Syntax:

When declaring and/or initializing memory variables or buffer elements, use one of these forms:

```
.BYTE varName1 [, varName2, ,...];

.BYTE = initExpression1, initExpression2, ,...;

.BYTE varName1 = initExpression,varName2 = initExpression2,,...;

.BYTE bufferName [] = initExpression1, initExpression2, ,...;

.BYTE bufferName []= "fileName ";

.BYTE bufferName [length ] = "fileName ";

.BYTE bufferName [length] = initExpression1, initExpression2, ,...;
```

where:

varName - user-defined symbols that name variables

bufferName - user-defined symbols that name buffers

fileName - indicates that the elements of a buffer get their initial values from the *fileName* data file. The fileName parameter can consist of the actual name and path specification for the data file. If the initialization file is in the current directory, only the *fileName* need be given inside double quotation mark (" ") characters. Note that when reading in a data file, the assembler reads in whitespace-separated lists of decimal digits or hex strings.

If the file name is not found in the current directory, the assembler looks in the directories in the processor include path. You can use the `-l filename` switch to add a directory to the processor include path.

Initializing from files is useful for loading buffers with data, such as filter coefficients or FFT phase rotation factors that are generated by other programs. The assembler determines how the values are stored in memory when it reads the data files.

Ellipsis (...) - represents a comma-delimited list of parameters.

initExpressions parameters - sets initial values for variables and buffer elements.

NOTE: The optional `[length]` parameter defines the length of the associated buffer in words. The number of initialization elements defines the *length* of an implicit-size buffer. The brackets `[ ]` that enclose the optional [*length*] are required. For more information, see the following `.BYTE` examples. In addition, use a `/R32` qualifier (`.BYTE4/R32`) to support 32-bit initialization for 1.31 fracts (see 1.31 Fracts).

The following lines of code demonstrate `.BYTE` directives:

```
Buffer1:
     .TYPE Buffer1, STT_OBJECT;
     .BYTE = 5, 6, 7;
     // initialize three 8-bit memory locations
     // for data label Buffer1
.Buffer1.end:
.BYTE samples[] = 123, 124, 125, 126, 127;
     // declare an implicit-length buffer and initialize it
     // with five 1-byte constants
.BYTE4/R32 points[] = 0.01r, 0.02r, 0.03r;
     // declare and initialize an implicit-length buffer
     // and initialize it with three 4-byte fract constants
.BYTE2 Ins, Outs, Remains;
     // declare three 2-byte variables zero-initialized by
     // default
.BYTE4 demo_codes[100] = "inits.dat";
     // declare a 100-location buffer and initialize it
     // with the contents of the inits.dat file;
.BYTE2 taps=100;
     // declare a 2-byte variable and initialize it to 100
.BYTE twiddles[10] = "phase.dat";
     // declare a 10-location buffer and load the buffer
     // with contents of the phase.dat file
.BYTE4/R32 Fract_Byte4_R32[] = "fr32FormatFract.dat";
```

When declaring or initializing variables with `.BYTE`, consider constraints applied to the `.VAR` directive. The `.VAR` directive allocates and optionally initializes 32-bit data objects. Refer to .VAR, Declare a Data Variable or Buffer for more information about the directive.

ASCII String Initialization Support

The assembler supports ASCII-string initialization. This allows the full use of the ASCII character set, including digits and special characters.

In Blackfin processors, ASCII initialization can be provided with `.BYTE`, `.BYTE2`, or `.VAR` directives. The most likely use is the `.BYTE` directive, where each `char` is represented by one byte versus a `.VAR` directive, in which each `char` needs four bytes. The characters are stored in the upper byte of 32-bit words. The LSBs are cleared.

String initialization takes one of the following forms:

```
.BYTE symbolString[length] = `initString', 0;
```

```
.BYTE symbolString[] = 'initString', 0;
```

Note that the number of initialization characters defines the optional *length* of a string (implicit-size initialization).

Example:

```
.BYTE k[13] = `Hello world!', 0;

.BYTE k[] = `Hello world!', 0;
```

The trailing zero character is optional. It simulates ANSI-C string representation.

.COMPRESS, Start Compression

NOTE: Used with the ADSP-214xx SHARC processors only.

The `.COMPRESS` directive indicates that all of the following instructions in the source file should be compressed, if possible. The directive affects sections assembled as short word. Compression can be canceled by a `.NOCOMPRESS` directive later in the source file (see .NOCOMPRESS, Terminate Compression).

`.COMPRESS` is advisory only:

- There is no guarantee that a particular instruction will be compressed.
- Instructions can be `uncompressed' if they are near the end of a `DO` loop.
- Whether a particular instruction is compressed can change due to assembler changes such as anomaly workarounds.
- There are no warnings if instructions cannot be compressed.

Therefore, it is not recommended to create code layouts (tables with fixed size entries) that depend on particular instructions being compressed.

Syntax:

```
.COMPRESS;
```

.EXTERN, Refer to a Globally Available Symbol

The `.EXTERN` directive allows a code module to reference global data structures, symbols that are declared as `.GLOBAL` in other files. For additional information, see the `.GLOBAL` directive (.GLOBAL, Make a Symbol Available Globally).

Syntax:

```
.EXTERN symbolName1[, symbolName2 , ...];
```

where:

symbolName - the name of a global symbol to import. A single `.EXTERN` directive can reference any number of symbols on one line, separated by commas.

Example:

```
.EXTERN coeffs;
        // This code declares an external symbol to reference
        // the global symbol "coeffs" declared in the example
        // code in the .GLOBAL directive description.
```

.EXTERN STRUCT, Refer to a Struct Defined Elsewhere

The `.EXTERN STRUCT` directive allows a code module to reference a struct defined in another file. Code in the assembly file can then reference the data members by name, just as if they were declared locally.

Syntax:

`.EXTERN STRUCT` *`typedef structvarName`*`;`

where:

`typedef` - the type definition for a struct VAR

`structvarName` - a struct VAR name

The `.EXTERN STRUCT` directive specifies a struct symbol name declared in another file. The naming conventions for structs are the same as for variables and arrays:

- If a struct was declared in a C file, refer to it with a leading _.
- If a struct was declared in an `.asm` file, use the name "as is", no leading underscore (_) is necessary.

The `.EXTERN STRUCT` directive optionally accepts a list, such as:

`.EXTERN STRUCT` *`typedef structvarName`* `[, STRUCT` *`typedef structvarName ...`*`]`

The key to the assembler knowing the layout is the `.IMPORT` directive and the `.EXTERN STRUCT` directive associating the *`typedef`* with the struct VAR. To reference a data structure declared in another file, use the `.IMPORT` directive with the `.EXTERN` directive. This mechanism can be used for structures defined in assembly source files as well as in C files.

The `.EXTERN` directive supports variables in the assembler. If the program references struct members, `.EXTERN STRUCT` must be used because the assembler must consult the struct layout to calculate the offset of the struct members. If the program does not reference struct members, you can use `.EXTERN` for struct VARs.

Example (SHARC Code):

```
.IMPORT "MyCelebrities.h";
   // 'Celebrity' is the typedef for struct var 'StNick'
   // .EXTERN means that '_StNick' is referenced within this
   // file, but not locally defined. This example assumes StNick
   // was declared in a C file and it must be referenced with
   // a leading underscore.
.EXTERN STRUCT Celebrity _StNick;
   // "isSeniorCitizen" is one of the members of the 'Celebrity'
   // type
P3.L = LO(_StNick->isSeniorCitizen);
```

```
P3.H = HI(_StNick->isSeniorCitizen);
```

.FILE_ATTR, Create an Attribute in the Object File

The `.FILE_ATTR` directive instructs the assembler to place an attribute in the object file which can be referenced in the `.ldf` file when linking. See the *Linker and Utilities Manual* for more information.

Syntax:

`.FILE_ATTR` *`attrName1`* `[=` *`attrVal1`*`]` `[,` *`attrName2`* `[=` *`attrVal2`*`];]`

where:

`attrName` - the name of the attribute. Attribute names must follow the same rules for naming symbols.

`attrVal` - sets the attribute to this value. If omitted, "1" is used. The value must be double-quoted unless it follows the rules for naming symbols (as described in Assembler Keywords and Symbols).

Examples:

```
.FILE_ATTR at1;
.FILE_ATTR at10=a123;
.FILE_ATTR at101=a123, at102,at103="999";
```

.FILE, Override the Name of a Source File

The `.FILE` directive overrides the name of the source file. This directive can appear in the C/C++ compiler-generated assembly source file (`.s`). The `.FILE` directive is used to ensure that the debugger has the correct file name for the source file that generated the object file.

Syntax:

`.FILE "` *`filename.ext`* `";`

where:

`filename` - the name of the source file to associate with the object file. The argument is enclosed in double quotes.

.FORCECOMPRESS, Compress the Next Instruction

NOTE: Used with the ADSP-214xx SHARC processors only.

The `.FORCECOMPRESS` directive causes the next instruction to be compressed, if possible. The directive affects sections that are assembled as short word. The directive overrides the effect of a previous `.NOCOMPRESS` directive (see .NOCOMPRESS, Terminate Compression) for one instruction. Only the immediately following assembly instruction is affected by `.FORCECOMPRESS`, while `.COMPRESS` (see .COMPRESS, Start Compression) starts a sequence of compressed instructions ended by `.NOCOMPRESS`.

The `.FORCECOMPRESS` directive can override certain conservative assumptions normally made by the assembler, such as when an immediate value is an expression containing a symbol. In this case, the assembler normally does not

generate a compressed instruction because the ultimate value of the symbolic expression may not fit in the immediate field of the compressed instruction.

The .FORCECOMPRESS directive is advisory only:

- There is no guarantee that a particular instruction will be compressed.
- An instruction can be `uncompressed' if it is near the end of a DO loop.
- Whether a particular instruction is compressed can change due to assembler changes such as anomaly workarounds.
- There are no warnings if an instruction cannot be compressed.

Therefore, it is not recommended to create code layouts (tables with fixed size entries) that depend on particular instructions being compressed.

Syntax:

```
.FORCECOMPRESS;
```

.GLOBAL, Make a Symbol Available Globally

The .GLOBAL directive changes the scope of a symbol from local to global, making the symbol available for reference in object files that are linked to the current one.

By default, a symbol has local binding, meaning the linker can resolve references to the symbol only from the local file (that is, the same file in which the symbol is defined). The symbol is visible only in the file in which it is declared. Local symbols in different files can have the same name, and the linker considers them to be independent entities. Global symbols are visible from other files; all references from other files to an external symbol by the same name will resolve to the same address and value, corresponding to the single global definition of the symbol.

Change the default scope with the .GLOBAL directive. Once the symbol is declared global, other files may refer to it with .EXTERN. For more information, refer to .COMPRESS, Start Compression. Note that .GLOBAL (or .WEAK) scope is required for symbols that appear in RESOLVE commands in the .ldf file.

NOTE: The .GLOBAL (or .WEAK) directive is required to make symbols available for placement through RESOLVE commands in the .ldf file.

Syntax:

```
.GLOBAL symbolName1 [, symbolName2 , ...];
```

where:

symbolName - the name of a global symbol. A single .GLOBAL directive may define the global scope of any number of symbols on one line, separated by commas.

Example (SHARC Code):

```
.VAR coeffs[10];          // declares a buffer
.VAR taps=100;            // declares a variable
```

```
.GLOBAL coeffs, taps;     // makes the buffer and the variable
                          // visible to other files
```

Example (Blackfin Code):

```
.BYTE coeffs[10];          // declares a buffer
.BYTE4 taps=100;           // declares a variable
.GLOBAL coeffs, taps;      // makes the buffer and the variable
                           // visible to other files
```

.IMPORT, Provide Structure Layout Information

The `.IMPORT` directive makes struct layouts visible inside an assembler program. The `.IMPORT` directive provides the assembler with the following structure layout information:

- The names of `typedefs` and `structs` available
- The name of each data member
- The sequence and offset of the data members
- Information as provided by the C compiler for the size of C base types (alternatively, for `SIZEOF()` C base types).

Syntax:

```
.IMPORT "headerfilename1" [, "headerfilename2", ...];
```

where:

`headerfilename` - one or more comma-separated C header files enclosed in double quotes.

NOTE: The system processes each `.IMPORT` directive and each file specified in an `.IMPORT` directive separately. Therefore, all type information must be available within the context for the individual file.

If `headerfile1.h` defines a type referenced in `headerfile2.h`, an attempt to import the second file into assembly fails. One solution is to have the assembler call the compiler once for the set of import statements. The compiler then has all the information it needs when processing the second header file.

In other words, create a third file to be imported in place of `headerfile2.h`. This file would simply consist of these lines:

```
#include "headerfile1.h"
#include "headerfile2.h"
```

The `.IMPORT` directive does not allocate space for a variable of this type. Allocating space requires the `.STRUCT` directive (see .STRUCT, Create a Struct Variable).

The assembler takes advantage of knowing the struct layouts. The assembly programmer can reference struct data members by name in an assembler source file, as one does in C. The assembler calculates the offsets within the structure based on the size and sequence of the data members.

If the structure layout changes, the assembly code needs no change. It needs to get the new layout from the header file, via the compiler. Make dependencies track the `.IMPORT` header files and know when a rebuild is needed. Use the `-flags-compiler` assembler switch to pass options to the C compiler for `.IMPORT` header file compilations.

An `.IMPORT` directive with one or more `.EXTERN` directives allow code in the module to refer to a struct variable that was declared and initialized elsewhere. The C struct can be declared in C-compiled code or another assembly file.

The `.IMPORT` directive with one or more `.STRUCT` directives declares and initializes variables of that structure type within the assembler section in which it appears.

For more information, refer to .EXTERN, Refer to a Globally Available Symbol and .STRUCT, Create a Struct Variable.

Example:

```
.IMPORT "CHeaderFile.h";
.IMPORT "ACME_IIir.h","ACME_IFir.h";
.SECTION program;
         // ... code that uses CHeaderFile, ACME_IIir, and
         // ACME_IFir C structs
```

.INC/BINARY, Include Contents of a File

The `.INC/BINARY` directive includes the content of file at the current location. You can control the search paths used via the -i command-line switch.

Syntax:

```
.INC/BINARY [symbol = "filename"[, skip[,count]];

.INC/BINARY [symbol[] = "filename"[, skip[,count]];
```

where:

`symbol` is the name of a symbol to associate with the data being included from the file

`filename` is the name of the file to include. The argument is enclosed in double quotes.

The *`skip`* argument skips a number of bytes from the start of the file.

The *`count`* argument indicates the maximum number of bytes to read.

Example:

```
.SECTION data1;

.VAR jim;
.INC/BINARY sym[] = "bert",10,6;
.VAR fred;
.INC/BINARY Image1[] = "photos/Picture1.jpg";
```

.LEFTMARGIN, Set the Margin Width of a Listing File

The .LEFTMARGIN directive sets the margin width of a listing page. It specifies the number of empty spaces at the left margin of the listing file (.lst), which the assembler produces when you use the -l switch. In the absence of the .LEFTMARGIN directive, the assembler leaves no empty spaces for the left margin.

The assembler compares the .LEFTMARGIN and .PAGEWIDTH values against one another. If the specified values do not allow enough room for a properly formatted listing page, the assembler issues a warning and adjusts the directive that was specified last to allow an acceptable line width.

Syntax:

.LEFTMARGIN *expression* ;

where:

expression evaluates to an integer from 0 to 100. Default is 0. Therefore, the minimum left margin value is 0 and the maximum left margin value is 100. To change the default setting for the entire listing, place the .LEFTMARGIN directive at the beginning of your assembly source file.

Example:

```
.LEFTMARGIN 9; /* the listing line begins at column 10 */
```

NOTE: You can set the margin width only once per source file. If the assembler encounters multiple occurrences of the .LEFTMARGIN directive, it ignores all of them except the last directive.

.LIST_DATFILE/.NOLIST_DATFILE, List Data Init Files

The .LIST_DATFILE/.NOLIST_DATFILE directives (off by default) turn the listing of data initialization files on and off. Nested source files inherit the current setting of this directive pair, but a change to the setting made in a nested source file will not affect the parent source file.

The .LIST_DATFILE/.NOLIST_DATFILE directives do not take any qualifiers or arguments.

Syntax:

```
.LIST_DATFILE;
```

```
.NOLIST_DATFILE;
```

These directives can appear multiple times anywhere in a source file, and their effect depends on their location in the source file. They are used in assembly source files, but not in data initialization files.

.LIST_DATA/.NOLIST_DATA, Listing Data Opcodes

The .LIST_DATA/.NOLIST_DATA directives (off by default) turn the listing of data opcodes on and off. When .NOLIST_DATA is in effect, opcodes that correspond to variable declarations do not appear in the opcode column. Nested source files inherit the current setting of this directive pair, but a change to the setting made in a nested source file does not affect the parent source file.

.LIST_DATA/.NOLIST_DATA directives do not take any qualifiers or arguments.

Syntax:

```
.LIST_DATA;
```

```
.NOLIST_DATA;
```

These directives can appear multiple times anywhere in a source file, and their effect depends on their location in the source file.

.LIST_DEFTAB/.LIST_LOCTAB, Set Tab Widths for Listings

Tab-characters in source files are expanded to blanks in listing files under the control of two internal assembler parameters that set the tab expansion width. The default tab width is normally in effect, but it can be overridden if the local tab width is explicitly set with a directive.

The .LIST_DEFTAB directive sets the default tab width, and the .LIST_LOCTAB directive sets the local tab width.

Both the default tab width and the local tab width can be changed any number of times via the .LIST_DEFTAB and .LIST_LOCTAB directives. The default tab width is inherited by nested source files, but the local tab width only affects the current source file.

Syntax:

.LIST_DEFTAB *expression*;

.LIST_LOCTAB *expression*;

where:

expression evaluates to an integer greater than or equal to 0.

In the absence of a .LIST_DEFTAB directive, the default tab width defaults to 4.

In the absence of a .LIST_LOCTAB directive, the local tab width defaults to the current setting for the default tab width.

A value of 0 sets the default tab width and the local tab width to the current setting of the default tab width.

Example:

```
    // Tabs here are expanded to the default of 4 columns
.LIST_DEFTAB 8;
    // Tabs here are expanded to 8 columns
.LIST_LOCTAB 2;
    // Tabs here are expanded to 2 columns
    // But tabs in "include_1.h" will be expanded to 8 columns
#include "include_1.h"
.LIST_DEFTAB 4;
    // Tabs here are still expanded to 2 columns
    // But tabs in "include_2.h" will be expanded to 4 columns
 #include "include_2.h"
```

.LIST_WRAPDATA/.NOLIST_WRAPDATA

The .LIST_WRAPDATA/.NOLIST_WRAPDATA directives control the listing of opcodes that are too big to fit in the opcode column. By default, the .NOLIST_WRAPDATA directive is in effect.

This directive pair applies to any opcode that does not fit, but in practice, such a value almost always is the data (alignment directives can also result in large opcodes).

- If .LIST_WRAPDATA is in effect, the opcode value is wrapped so that it fits in the opcode column (resulting in multiple listing lines).
- If .NOLIST_WRAPDATA is in effect, the printout is what fits in the opcode column.

Nested source files inherit the current setting of this directive pair, but a change to the setting made in a nested source file does not affect the parent source file.

The .LIST_WRAPDATA/.NOLIST_WRAPDATA directives do not take any qualifiers or arguments.

Syntax:

```
.LIST_WRAPDATA;
```

```
.NOLIST_WRAPDATA;
```

These directives can appear multiple times anywhere in a source file, and their effect depends on their location in the source file.

.LIST/.NOLIST, Listing Source Lines and Opcodes

The .LIST/.NOLIST directives (on by default) turn on and off the listing of source lines and opcodes.

If .NOLIST is in effect, no lines in the current source (or any nested source) are listed until a .LIST directive is encountered in the same source, at the same nesting level. The .NOLIST directive operates on the next source line, so that the line containing a .NOLIST appears in the listing and accounts for the missing lines.

The .LIST/.NOLIST directives do not take any qualifiers or arguments.

Syntax:

```
.LIST
```

```
.NOLIST;
```

These directives can appear multiple times anywhere in a source file, and their effect depends on their location in the source file.

.LONG, Define and Initialize 4-Byte Data Objects

NOTE: Used with the Blackfin processors only.

The .LONG directive declares and optionally initializes four-byte data objects. It is effectively equivalent to .BYTE4 *initExpression1, initExpression2*. For more information, see .BYTE, Declare a Byte Data Variable or Buffer.

Syntax:

When declaring and/or initializing memory variables or buffer elements, use the following format. Note that the terminating semicolon is optional.

.LONG *initExpression1, initExpression2, ...*[;]

.LONG *constExpression1, constExpression2, ...*[;]

where:

initExpressions parameters contain one or more comma-separated "symbol=value" expressions.

constExpressions parameters contain a comma-separated list of constant values.

The following lines of code demonstrate .LONG directives:

```
       // Define an initialized variable
.LONG buf1=0x1234;
       // Define two initialized variables
.LONG 0x1234, 0x5678;
       // Declare three 4-byte areas of memory, initialized to
       // 3, 4, and 5 respectively
.LONG 0x0003, 0x0004, 0x0005;
```

.MESSAGE, Alter the Severity of an Assembler Message

The .MESSAGE directive can be used to alter the severity of an error, warning, or informational message generated by the assembler for all or part of an assembly source.

Syntax:

.MESSAGE/*qualifier warnid1*[*, warnid2, ...*] ;

.MESSAGE/*qualifier warnid1*[*, warnid2, ...*] UNTIL *sym* ;

.MESSAGE/*qualifier warnid1*[*, warnid2, ...*] FOR *n* LINES;

.MESSAGE/DEFAULT/*qualifier warnid1*[*, warnid2, ...*] ;

where:

warnid1[*, warnid2, ...*] is a list of one or more message identification numbers.

A qualifier can be:

- ERROR - change messages to errors
- WARN - change messages to warnings

- INFO - change messages to informational messages
- SUPPRESS - do not output the messages
- RESTORE_CL - change the severity of the messages back to the default values they had at the beginning of the source file, after the command line arguments were processed, but before any DEFAULT directives have been processed.
- RESTORE - change the severity of the messages back to the default values they had at the beginning of the source file, after the command line arguments were processed, and after any DEFAULT directives have been processed.
- POP - change the severity of the messages back to what they were prior to the previous .MESSAGE directive.

The RESTORE, RESTORE_CL, and POP qualifiers cannot be used with the UNTIL, FOR, or DEFAULT forms of the .MESSAGE directive.

The DEFAULT qualifier cannot be used with the UNTIL or FOR forms of the .MESSAGE directive.

The simple form of the .MESSAGE directive changes the severity of messages until another .MESSAGE directive is seen. It can be placed anywhere in a source file. Messages that cannot be associated with a source line can be reported with line number 0. These cannot be altered in severity by a .MESSAGE directive; use the -Werror number[, number], -Wwarn number[, number], -Winfo number[, number], or -Wsuppress number[, number] assembler switch.

Example:

```
.MESSAGE/ERROR 1049;
.SECTION program;
.VAR two[2]=1;                  // generates an error
.MESSAGE/SUPPRESS 1049;
.VAR three[3]=1,2;              // generates no message
.MESSAGE/WARN 1049;
.VAR four[4]=1,2,3;             // generates a warning
```

The temporary forms of the .MESSAGE directive (UNTIL and FOR) changes the severity of messages until the specified label (or for the specified number of source lines). The temporary forms of the .MESSAGE directive must start and end within a single .SECTION directive.

Include files inherit any severity changes from the files which #include them. .MESSAGE directives in include files do not control the severity of messages generated after returning to the source file which included them.

A .MESSAGE/DEFAULT directive in an include file controls the severity of messages generated after returning to the source file which included them.

.NEWPAGE, Insert a Page Break in a Listing File

The .NEWPAGE directive inserts a page break in the printed listing file (.lst), which the assembler produces when you use the -l filename switch. The assembler inserts a page break at the location of the .NEWPAGE directive.

The .NEWPAGE directive does not take any qualifiers or arguments.

Syntax:

```
.NEWPAGE;
```

This directive can appear anywhere in your source file. In the absence of the `.NEWPAGE` directive, the assembler generates no page breaks in the file.

.NOCOMPRESS, Terminate Compression

NOTE: Used with the ADSP-214xx SHARC processors only.

The `.NOCOMPRESS` directive indicates that all of the following instructions in the source file should not be compressed.

Syntax:

```
.NOCOMPRESS;
```

The directive's effect is canceled by a `.COMPRESS, Start Compression` directive later in the source file. The directive's effect also is canceled by a `.FORCECOMPRESS, Compress the Next Instruction` directive for one instruction only.

.PAGELENGTH, Set the Page Length of a Listing File

The `.PAGELENGTH` directive controls the page length of the listing file produced by the assembler when you use the `-l filename` switch.

Syntax:

`.PAGELENGTH` *`expression`*;

where:

`expression` - evaluates to an integer 0 or greater. It specifies the number of text lines per printed page. The default page length is 0, which means the listing has no page breaks.

To format the entire listing, place the `.PAGELENGTH` directive at the beginning of your assembly source file. If a page length value greater than 0 is too small to allow a properly formatted listing page, the assembler issues a warning and uses its internal minimum page length (approximately 10 lines).

Example:

```
.PAGELENGTH 50; // starts a new page after printing 50 lines
```

NOTE: You can set the page length only once per source file. If the assembler encounters multiple occurrences of the directive, it ignores all except the last directive.

.PAGEWIDTH, Set the Page Width of a Listing File

The `.PAGEWIDTH` directive sets the page width of the listing file produced by the assembler when you use the `-l` switch.

Syntax:

`.PAGEWIDTH` *`expression`* `;`

where:

`expression` evaluates to an integer

Depending on setting of the `.LEFTMARGIN` directive, this integer should be at least equal to:

- `LEFTMARGIN` value plus 46 (for Blackfin processors)
- `LEFTMARGIN` value plus about 66 (for SHARC processors)

You cannot set this integer to less than 46, 49, or 66, respectively. There is no upper limit. If `LEFTMARGIN` = 0 and the `.PAGEWIDTH` value is not specified, the actual page width is set to any number over 46, 49, or 66, respectively.

To change the number of characters per line in the entire listing, place the `.PAGEWIDTH` directive at the beginning of the assembly source file.

Example:

```
.PAGEWIDTH 72;   // starts a new line after 72 characters
                 // are printed on one line, assuming
                 // the .LEFTMARGIN setting is 0.
```

NOTE: You can set the page width only once per source file. If the assembler encounters multiple occurrences of the directive, it ignores all of them except the last directive.

.PORT, Legacy Directive

NOTE: Used with the SHARC processors only.

The `.PORT` legacy directive assigns port name symbols to I/O ports. Port name symbols are global symbols that correspond to memory-mapped I/O ports defined in the `.ldf` file.

The `.PORT` directive uses the following syntax:

`.PORT` *`portName;`*

where:

`portName` is a globally available port symbol.

Example:

```
.PORT p1; // declares I/O port P1

.PORT p2; // declares I/O port P2
```

To declare a port using the SHARC assembler, use the `.VAR` directive (for port-identifying symbols) and the linker description file (for corresponding I/O sections). The linker resolves port symbols in the `.ldf` file.

For more information on the linker description file, see the *Linker and Utilities Manual*.

.PRECISION, Select Floating-Point Precision

NOTE: Used with the SHARC processors only.

The `.PRECISION` directive controls how the assembler interprets floating-point numeric values in constant declarations and variable initializations. To configure the floating-point precision of the target processor system, you must set up control registers of the chip using instructions specific to the processor core.

Use one of the following syntax options:

```
.PRECISION [=] 32;
```

```
.PRECISION [=] 40;
```

where:

The precision of 32 or 40 (default) specifies the number of significant bits for floating-point data. The equal sign (=) following the `.PRECISION` keyword is optional.

Note that the `.PRECISION` directive applies only to floating-point data. Precision of fixed-point data is determined by the number of digits specified. The `.PRECISION` directive applies to all following floating-point expressions in the file up to the next `.PRECISION` directive.

Example:

```
.PRECISION=32; /* Selects standard IEEE 32-bit
                  single-precision format. */

.PRECISION 40; /* Selects 40-bit format with
                  extended mantissa. This is the default
                  setting. */
```

NOTE: The `.ROUND_` directives (see .ROUND_, Select Floating-Point Rounding) specify how the assembler converts a value of many significant bits to fit into the selected precision.

.PREVIOUS, Revert to the Previously Defined Section

The `.PREVIOUS` directive instructs the assembler to set the current section in memory to the section described immediately before the current one. The `.PREVIOUS` directive operates on a stack.

Syntax:

```
.PREVIOUS;
```

The following examples provide valid and invalid cases of the use of the consecutive `.PREVIOUS` directives.

Example of Invalid Directive Use:

```
.SECTION data1;       // data
.SECTION code;        // instructions
```

```
.PREVIOUS;            // previous section ends, back to data1
.PREVIOUS;            // no previous section to set to
```

Example of Valid Directive Use:

```
#define MACRO1    \
.SECTION data2;   \
    .VAR vd = 4;  \
.PREVIOUS;
.SECTION data1;       // data
   .VAR va = 1;
.SECTION program;     // instructions
    .VAR vb = 2;
    MACRO1            // invoke macro
.PREVIOUS;
   .VAR vc = 3;
```

evaluates as:

```
.SECTION data1;       // data
   .VAR va = 1;
.SECTION program;     // instructions
    .VAR vb = 2;
                      // Start MACRO1
.SECTION data2;
    .VAR vd = 4;
.PREVIOUS;            // end data2, section program
                      // End MACRO1
.PREVIOUS;            // end program, start data1
    .VAR vc = 3;
```

PRIORITY, Allow Prioritized Symbol Mapping in Linker

The `.PRIORITY` directive allows prioritized symbol mapping in the linker. The directive can be specified for:

- A symbol defined in the same file as the directive
- A globally defined symbol
- A local symbol in a different source file

Syntax:

```
.PRIORITY symbolName, priority;
.PRIORITY symbolName, "sourcefile", priority;
```

where:

In the first case, *`symbolName`* is a global symbol or locally defined symbol. In the second case, *`symbolName`* is a symbol defined in " *`sourcefile`* ".

Example:

```
.PRIORITY _foo, 35;                // Symbol with highest priority
```

```
.PRIORITY _main, 15;                // Symbol with medium priority
.PRIORITY bar, "barFile.asm", -10;  // Symbol with lowest priority
```

Linker Operation

After the absolute placement of symbols specified in the `.ldf` file's `RESOLVE()` command, but before mapping commands are processed, the linker tries to map all symbols appearing in priority directives in decreasing order of their priorities.

The prioritized symbol is placed into memory that contains only the `INPUT_SECTIONS()` command for input sections defining the symbol. Symbols with assigned priority are mapped after absolutely placed symbols, but before symbols without assigned priority.

The symbols are placed into memory segments based on the order that the segments appear in the `.ldf` file. Therefore, an output section targeting a higher-priority memory segment should appear before an output section targeting a lower-priority segment.

Example of Assembler Code:

```
.section program;
     _func1:
     _func2:
.section L1_code;
     _L1_func:
     ...
.PRIORITY _L1_func,10;
.PRIORITY _func1,11;
.PRIORITY _func2,12;
```

Example of LDF Code:

```
L1_A   { INPUT_SECTIONS($OBJECTS(L1_code))} > L1_A;
L1_B   { INPUT_SECTIONS($OBJECTS(L1_code program)) } > L1_B;
L2     { INPUT_SECTIONS($OBJECTS(program))} > L2;
```

The preceding examples result in the linker executing the following steps:

1. Because `_func2` is assigned the highest priority (12) in the assembler code, the linker tries to map it first. It is in section "program" and can only be mapped into memory segment `L1_B` or `L2`. The linker attempts to map in the order the segments appear in the LDF. The linker first tries to map it into the `L1_B` memory segment. If `_func2` does not fit into `L1_B` , the linker tries the `L2` segment.
2. Because `_func1` is assigned the middle priority (11) in the assembler code, the linker maps it next. This symbol is also in the section "program" and so the linker first tries to map it into the `L1_B` memory segment. If `_func1` does not fit into `L1_B` , the linker tries the `L2` segment.
3. Because `_L1_func` is assigned the lowest priority (10) in the assembler code, the linker maps it last. Because this symbol is in section `L1_code`, the linker first tries to map it into the `L1_A` memory segment. If `_L1_func` does not fit into `L1_A` , the linker tries the `L1_B` segment.

.REFERENCE, Provide Better Info in an X-REF File

NOTE: Used with the Blackfin processors only.

The `.REFERENCE` directive is used by the compiler to provide better information in an X-REF file generated by the linker. This directive is used when there are indirect symbol references that would otherwise not appear in an X-REF file.

The `.REFERENCE` directive uses the following syntax:

`.REFERENCE` *`symbol;`*

where:

`symbol` is a symbol.

Example:

```
.REFERENCE
P1;

.REFERENCE
P2;
```

.RETAIN_NAME, Stop Linker from Eliminating Symbol

The `.RETAIN_NAME` directive stops the linker from eliminating the symbol when linking the generated object file. This directive has the same effect as the `KEEP()` LDF command.

Syntax:

The `.RETAIN_NAME` directive uses the following syntax:

`.RETAIN_NAME` *`symbol;`*

where:

`symbol` is a user-defined symbol.

For information on `KEEP()`, refer to the *Linker and Utilities Manual*.

.ROUND_, Select Floating-Point Rounding

NOTE: Used with the SHARC processors only.

The `.ROUND_` directives control how the assembler interprets literal floating-point numeric data after `.PRECISION` is defined. The `.PRECISION` directive determines the number of bits to be truncated to match the number of significant bits; see .PRECISION, Select Floating-Point Precision.

The .ROUND_ directives determine how the assembler handles the floating-point values in constant declarations and variable initializations. They do not affect the floating-point rounding modes of the target processor system which are determined by processor-specific control registers.

The .ROUND_ directives use the following syntax:

.ROUND_*mode* ;

where:

The *mode* string specifies the rounding scheme used to fit a value in the destination format. Use one of the following IEEE standard modes:

.ROUND_NEAREST; (default: round to nearest)

.ROUND_PLUS; (round towards positive infinity)

.ROUND_MINUS; (round towards negative infinity)

.ROUND_ZERO; (round towards zero)

In the following examples, the numbers with four decimal places are reduced to three decimal places and rounded accordingly.

```
.ROUND_NEAREST;
  /* Selects Round-to-Nearest scheme; the default setting.
     A 5 is added to the digit that follows the third
     decimal digit (the least significant bit - LSB). The
     result is truncated after the third decimal digit (LSB).

     1.2581 rounds to 1.258
     8.5996 rounds to 8.600
     -5.3298 rounds to -5.329
     -6.4974 rounds to -6.496
   */
.ROUND_ZERO;
  /* Selects Round-to-Zero. The closer to zero value is taken.
     The number is truncated after the third decimal digit (LSB).

     1.2581 rounds to 1.258
     8.5996 rounds to 8.599
     -5.3298 rounds to -5.329
     -6.4974 rounds to -6.497
   */
.ROUND_PLUS;
  /* Selects Round-to-Positive Infinity. The number rounds
     to the next larger.
     For positive numbers, a 1 is added to the third decimal
     digit (the least significant bit). Then the result is
     truncated after the LSB.
     For negative numbers, the mantissa is truncated after
     the third decimal digit (LSB).
```

```
     1.2581 rounds to 1.259
     8.5996 rounds to 8.600
     -5.3298 rounds to -5.329
     -6.4974 rounds to -6.497
   */
.ROUND_MINUS;
  /* Selects Round-to-Negative Infinity. The value
     rounds to the next smaller.
     For negative numbers, a 1 is subtracted from the
     third decimal digit (the least significant bit).
     Then the result is truncated after the LSB.
     For positive numbers, the mantissa is truncated
     after the third decimal digit (LSB).

     1.2581 rounds to 1.258
     8.5996 rounds to 8.599
     -5.3298 rounds to -5.330
     -6.4974 rounds to -6.498
   */
```

.SECTION, Declare a Memory Section

The `.SECTION` directive marks the beginning of a logical section corresponding to an array of contiguous locations in your processor memory. Statements between one `.SECTION` directive and the following `.SECTION` directive (or the end-of-file instruction), comprise the content of the section.

Blackfin Syntax:

`.SECTION`*`/qualifier [/qualifier] sectionName [sectionType];`*

SHARC Syntax:

`.SECTION`*`/TYPE/qualifier sectionName [sectionType];`*

NOTE: All qualifiers are optional, and more than one qualifier can be used.

Common .SECTION Attributes

The following are common syntax attributes used by the assembler:

- *`sectionName`* - section name symbol which is not limited in length and is case sensitive. Section names must match the corresponding input section names used by the `.ldf` file to place the section. Use the default `.ldf` file included in the *`<install_path>/ processor_family`* `/ldf` subdirectory of the installation directory, or write your own `.ldf` file.

  NOTE: Some sections starting with "`.`" names have certain meaning within the linker. Do not use the dot (`.`) as the initial character for *`sectionName`*.

The assembler generates relocatable sections for the linker to fill in the addresses of symbols at link-time. The assembler implicitly prefixes the name of the section with the ".rela." string to form a relocatable section. To avoid ambiguity, ensure that your section names do not begin with ".rela.".

- *sectionType* - an optional ELF section type identifier. The assembler uses the default SHT_PROGBITS when this identifier is absent.

Blackfin Example:

```
/* Declared below memory sections correspond to the default
   LDF's input sections. */
.SECTION/DOUBLE32 data1;     /* memory section to store data */
.SECTION/DOUBLE32 program;   /* memory section to store code */
```

DOUBLE* Qualifiers

The DOUBLE* qualifier can be one of the qualifiers in the ***DOUBLE Qualifiers*** table.

Table 2-13: DOUBLE Qualifiers

Qualifier	*Description*
DOUBLE32	DOUBLEs are represented as 32-bit types
DOUBLE64	DOUBLEs are represented as 64-bit types
DOUBLEANY	Section does not include code that depends on the size of DOUBLE

The DOUBLE size qualifiers are used to ensure that object files are consistent when linked together and with run-time libraries. A memory section can have one DOUBLE size qualifier - it cannot have two DOUBLE size qualifiers. Sections in the same file do not have to have the same type size qualifiers.

NOTE: Use of DOUBLEANY in a section implies that DOUBLE's are not used in this section in any way that would require consistency checking with any other section.

SHARC-Specific Qualifiers

For the SHARC assembler, the .SECTION directive supports qualifiers that specify the size of data words in the section. A qualifier can specify restricted placement for the section. Each section that defines data or code must bear an appropriate size qualifier; the placement qualifier is optional. The ***SHARC-Specific Qualifiers*** table lists qualifiers that are specific to SHARC processors.

Table 2-14: SHARC-Specific Qualifiers

Memory/Section Type	*Description*
PM or Code	Section contains instructions and/or data in the processor's default code word size; for example, 16 bits for ADSP-214xx processors (without the -nwc switch), and 48 bits for everything else.
DM or Data	Section contains data in 40-bit words

Table 2-14: SHARC-Specific Qualifiers (Continued)

Memory/Section Type	*Description*
DATA64	Section defines data in 64-bit words
DMAONLY	Placement qualifier for a section to be placed in memory and accessed through DMA only. The qualifier passes to the linker a request to place the section in a memory segment with the DMAONLY qualifier, which applies to memory accessed through the external parallel port of the ADSP-2126x and some ADSP-2136x processors.
NW	Placement qualifier for a normal-word section. Instructions will be assembled as normal 48-bit instructions and loaded into a 48-bit memory segment. Unlike PM sections, NW sections are always 48 bits and unaffected by the -short-word-code or -swc switch. See -short-word-code or -swc. **NOTE**: Applicable to the ADSP-214xx processors only.
SW	Placement qualifier for a 16-bit short-word section. Instructions will be assembled and loaded into a 16-bit short-word memory segment. Instructions will be assembled as compressed 16- or 32-bit instructions, if possible. See -short-word-code or -swc for more information. **NOTE**: Applicable to the ADSP-214xx processors only.

Example:

```
.SECTION/DM/DMAONLY seg_extm;

.VAR _external_var[100];
```

Initialization Section Qualifiers

The .SECTION directive may identify "how/when/if" a section is initialized. The initialization qualifiers, common for all supported assemblers, are listed in the ***SHARC-Specific Initialization Qualifiers*** table.

Table 2-15: SHARC-Specific Initialization Qualifiers

Qualifier	*Description*
NO_INIT	The section is "sized" to have enough space to contain all data elements placed in this section. No data initialization is used for this memory section.
ZERO_INIT	Similar to /NO_INIT, except that the memory space for this section is initialized to zero at "load time" or "runtime", if invoked with the linker's -meminit switch. If the -meminit switch is not used, the memory is initialized at "load" time when the .DXE file is loaded via the IDE, or boot-loaded by the boot kernel. If the memory initializer is invoked, the C/C++ run-time library (CRTL) processes embedded information to initialize the memory space during the CRTL initialization process.
RUNTIME_INIT	If the memory initializer is not run, this qualifier has no effect. If the memory initializer is invoked, the data for this section is set during the CRTL initialization process.

Example:

```
.SECTION/NO_INIT seg_bss;
.VAR big[0x100000];
```

```
.SECTION/ZERO_INIT seg_bsz;
.VAR big[0x100000];
```

Initialized data in a /NO_INIT or /ZERO_INIT section is ignored. For example, the assembler generates a warning for the .VAR zzinitialization below.

```
.SECTION/DM/NO_INIT seg_bss;
.VAR xx[1000];
.VAR zz = 25;  /* [Warning ea1141] "example.asm":3 'zz':
                  Data directive with assembly-time initializers found
                  in .SECTION 'seg_bss' with qualifier /NO_INIT. */
```

Likewise, the assembler generates a warning for an explicit initialization to 0 in a ZERO_INIT section.

```
.SECTION/DM/ZERO_INIT seg_bsz;
.VAR xx[1000];
.VAR zz = 0;
```

The assembler calculates the size of NO_INIT and ZERO_INIT sections exactly as for the standard SHT_PROGBITS sections. These sections, like the sections with initialized data, have the SHF_ALLOC flag set. Alignment sections are produced for NO_INIT and ZERO_INIT sections.

.SET, Set a Symbolic Alias

The .SET directive is used to alias one symbol to another.

Syntax:

```
.SET symbol1, symbol2
```

where:

symbol1 becomes an alias to *symbol2*.

Example:

```
.SET symbol1_alias, symbol1
```

.SHORT, Defines and Initializes 2-Byte Data Objects

NOTE: Used with the Blackfin processors only.

The .SHORT directive declares and optionally initializes two-byte data objects. It is effectively equivalent to .BYTE2 *initExpression1, initExpression2, ...*. For more information, see .BYTE, Declare a Byte Data Variable or Buffer.

Syntax:

When declaring and/or initializing memory variables or buffer elements, use this format. Note that the terminating semicolon is optional.

```
.SHORT initExpression1, initExpression2, ...[;]
```

```
.SHORT constExpression1, constExpression2, ...[;]
```

where:

initExpressions parameters - contain one or more comma-separated "symbol=value" expressions

constExpressions parameters - contain a comma-separated list of constant values

The following lines of code demonstrate the .SHORT directives:

```
       // Declare three 2-byte variables, zero-initialized
.SHORT Ins, Outs, Remains;
       // Declare a 2-byte variable and initialize it to 100
.SHORT taps=100;
       // Declare  three 2-byte areas of memory, initialized to
       // 3, 4 and 5 respectively
.SHORT 0x3, 0x4, 0x5;
```

.STRUCT, Create a Struct Variable

The .STRUCT directive allows you to define and initialize high-level data objects within the assembly code. The .STRUCT directive creates a struct variable using a C-style *typedef* as its guide from .IMPORT C header files.

Syntax:

```
.STRUCT typedef structName;
.STRUCT typedef structName = {};
.STRUCT typedef structName = {struct-member-initializers
          [, struct-member-initializers ...] };
.STRUCT typedef ArrayOfStructs[] = {struct-member-initializers
          [, struct-member-initializers ...] };
```

where:

typedef - name of a struct type definition

structName - name of a new struct variable

struct-member-initializers - per struct member initializers

The { } curly braces are used for consistency with the C initializer syntax. Initialization can be in long form or short form where data member names are not included. The short form corresponds to the syntax in C compiler struct initialization with these changes:

- C compiler keyword struct is changed to .struct by adding the period (.)
- C compiler constant string syntax "MyString" is changed to 'MyString' by changing the double quotes (" ") into single quotes (' ').

The long form is assembler-specific and provides the following benefits:

- Provides better error checking
- Supports self-documenting code

- Protects from possible future changes to the layout of the `struct`. If an additional member is added before the member is initialized, the assembler will continue to offset to the correct location for the specified initialization and zero-initialize the new member.

Any members that are not present in a long-form initialization are initialized to zero. For example, if struct `StructThree` has three members (`member1`, `member2`, and `member3`), and

```
.STRUCT StructThree myThree = {
   member1 = 0xaa,
   member3 = 0xff
};
```

`member2` is initialized to 0 because no initializer was present for it. If no initializers are present, the entire `struct` is zero-initialized.

If data member names are present, the assembler validates that the assembler and compiler are in agreement about these names. The initialization of data struct members declared via the assembly `.STRUCT` directive is processor-specific.

Example 1. Long Form .STRUCT Directive

```
#define NTSC 1
            // contains layouts for playback and capture_hdr
.IMPORT "comdat.h";
.STRUCT capture_hdr myLastCapture = {
   captureInt = 0,
   captureString = `InitialState'
};
.STRUCT myPlayback playback = {
   theSize = 0,
   ready = 1,
   stat_debug = 0,
   last_capture = myLastCapture,
   watchdog = 0,
   vidtype = NTSC
};
```

Example 2. Short Form .STRUCT Directive

```
#define NTSC 1
            // contains layouts for playback and capture_hdr
.IMPORT "comdat.h";
.STRUCT capture_hdr myLastCapture = { 0, `InitialState' };
.STRUCT playback myPlayback = {0, 1, 0, myLastCapture, 0, NTSC};
```

Example 3. Long Form .STRUCT Directive to Initialize an Array

```
.STRUCT structWithArrays XXX = {
   scalar = 5,
   array1 = { 1,2,3,4,5 },
   array2 = { "file1.dat" },
   array3 = "WithBraces.dat"   // must have { } within dat
```

```
};
```

In the short form, nested braces can be used to perform partial initializations as in C. In Example 4 below, if the second member of the struct is an array with more than four elements, the remaining elements are initialized to zero.

Example 4. Short Form .STRUCT Directive to Initialize an Array

```
.STRUCT structWithArrays XXX = { 5, { 1,2,3,4 }, 1, 2 };
```

Example 5. Initializing a Pointer

A struct may contain a pointer. Initialize pointers with symbolic references.

```
.EXTERN outThere;
.VAR myString[] = 'abcde',0;
.STRUCT structWithPointer PPP = {
   scalar = 5,
   myPtr1 = myString,
   myPtr2 = outThere
};
```

Example 6. Initializing a Nested Structure

A struct may contain a struct. Use fully qualified references to initialize nested struct members.

```
.STRUCT NestedStruct nestedOne = {
   scalar = 10,
   nested->scalar1 = 5,
   nested->array = { 0x1000, 0x1010, 0x1020 }
};
```

.TYPE, Change Symbol Type

The `.TYPE` directive directs the assembler to change the symbol type of an object. This directive may appear in the compiler-generated assembly source file (`.s`).

Syntax:

`.TYPE` *`symbolName, symbolType`*;

where:

`symbolName` is the name of the object to which the *`symbolType`* is applied.

`symbolType` is an ELF symbol type `STT_*`. Valid ELF symbol types are listed in the `ELF.h` header file. By default, a label has an `STT_FUNC` symbol type, and a variable or buffer name defined in a storage directive has an `STT_OBJECT` symbol type.

.VAR, Declare a Data Variable or Buffer

The `.VAR` directive declares and optionally initializes variables and data buffers. A variable uses a single memory location, and a data buffer uses an array of memory locations.

When declaring or initializing variables:

- A `.VAR` directive can appear only within a section. The assembler associates the variable with the memory type of the section in which the `.VAR` appears.
- A single `.VAR` directive can declare any number of variables or buffers, separated by commas, on one line.

  Unless the absolute placement for a variable is specified with a `RESOLVE()` command (from an `.ldf` file), the linker places variables in consecutive memory locations. For example, `.VAR d,f,k[50];` sequentially places symbols `x`, `y`, and 50 elements of the buffer `z` in the processor memory.
- The number of initializer values may not exceed the number of variables or buffer locations that you declare.
- The `.VAR` directive may declare an implicit-size buffer by using empty brackets `[ ]`. The number of initialization elements defines the *`length`* of the implicit-size buffer. At runtime, the length operator can be used to determine the buffer size. For example,

```
.SECTION data1;
   .VAR buffer [] = 1,2,3,4;
.SECTION program;
   LO = LENGTH( buffer );   // Returns 4
```

Syntax:

The `.VAR` directive takes one of the following forms:

```
.VAR varName1[, varName2, ...];
.VAR = initExpression1, initExpression2, ...;
.VAR bufferName[] = {initExpression1, initExpression2, ...};
.VAR bufferName[] = {"fileName"};
.VAR bufferName[length] = "fileName";
.VAR bufferName[length] = initExpression1, initExpression2, ...;
```

where:

`varName` - user-defined symbols that identify variables

`bufferName` - user-defined symbols that identify buffers

`fileName` parameter - indicates that the elements of a buffer get their initial values from the *`fileName`* data file. The *`fileName`* can consist of the actual name and path specification for the data file. If the initialization file is in the current directory of your operating system, only the *`fileName`* is quoted. Note that when reading in a data file, the assembler reads in whitespace-separated lists of decimal digits or hex strings.

Initialization from files is useful for loading buffers with data, such as filter coefficients or FFT phase rotation factors that are generated by other programs. The assembler determines how the values are stored in memory when it reads the data files.

Ellipsis (`...`) - a comma-delimited list of parameters.

`[` *`length`* `]` - optional parameter that defines the length (in words) of the associated buffer. When length is not provided, the buffer size is determined by the number of initializers.

Brackets ([ ]) - enclosing the optional [*length*] is required. For more information, see the following .VAR examples.

initExpressions parameters - set initial values for variables and buffer elements.

NOTE: With Blackfin processors, the assembler uses a /R32 qualifier (.VAR/R32) to support 32-bit initialization for use with 1.31 fracts. (See 1.31 Fracts.)

The following code demonstrates some .VAR directives.

```
.VAR buf1=0x1234;
     /* Define one initialized variable */
.VAR=0x1234, 0x5678;
     /* Define two initialized words */
.VAR samples[] = {10, 11, 12, 13, 14};
     /* Declare and initialize an implicit-length buffer
        since there are five values; this has the same effect
        as samples[5]. */
     /* Initialization values for implicit-size buffer must
        be in curly brackets. */
.VAR Ins, Outs, Remains;
     /* Declare three uninitialized variables */
.VAR samples[100] = "inits.dat";
     /* Declare a 100-location buffer and initialize it
        with the contents of the inits.dat file; */
.VAR taps=100;
     /* Declare a variable and initialize the variable to 100 */
.VAR twiddles[10] = "phase.dat";
     /* Declare a 10-location buffer and load the buffer
        with the contents of the phase.dat file */
.VAR Fract_Var_R32[] = "fr32FormatFract.dat";
     /* Declare a buffer that gets its initial values from the
        file fr32FormatFract.dat */
```

NOTE: All Blackfin processor memory accesses require proper alignment. Therefore, when loading or storing an N-byte value into the processor, ensure that this value is aligned in memory by N boundary; otherwise, a hardware exception is generated.

Blackfin Code Example:

In the following example, the 4-byte variables y0, y1, and y2 would be misaligned unless the .ALIGN 4; directive is placed before the .VAR y0; and .VAR y2; statements.

```
.SECTION data1;
.ALIGN 4;
.VAR X0;
.VAR X1;
.BYTE B0;
.ALIGN 4;  /* aligns the following data item "Y0" on a word
              boundary; advances other data items consequently */
```

```
.VAR Y0;
.VAR Y1;
.BYTE B1;
.ALIGN 4;  /* aligns the following data item "Y2" on a word
              boundary */
.VAR Y2;
```

VAR and ASCII String Initialization Support

The assemblers support ASCII string initialization. This allows the full use of the ASCII character set, including digits and special characters.

On SHARC processors, the characters are stored in the lower byte of 32-bit words. The remaining bits are cleared.

On Blackfin processors, use the `.BYTE` directive (see .BYTE, Declare a Byte Data Variable or Buffer for more information).

String initialization takes one of the following forms:

```
.VAR symbolString[length] = `initString', 0;
.VARsymbolString[] =  `initString', 0;
```

Note that the number of initialization characters defines the length of a string.

For example,

```
.VAR x[13] = `Hello world!', 0;
.VAR x[] = {`Hello world!', 0};
```

The trailing zero character is optional. It simulates ANSI-C string representation.

Strings can be mixed with numeric constants, which allows special characters to be included. For example,

```
.VAR s1[] = {'1st line',13,10,'2nd line',13,10,0};
                                        /* carriage return */
.VAR s2[] = {'say:"hello"',13,10,0}; /* quotation mark */
.VAR s3[] = {'say:',39,'hello',39,13,10,0};
                                        /* simple quotation marks */
```

.WEAK, Weak Symbol Definition and Reference

The `.WEAK` directive supports weak binding for a symbol. Use this directive where the symbol is defined (replacing the `.GLOBAL` directive to make a weak definition) and the `.EXTERN` directive (to make a weak reference).

Syntax:

`.WEAK` *`symbol`* `;`

where:

`symbol` is the user-defined symbol.

Although the linker generates an error if two objects define global symbols with identical names, it allows any number of instances of weak definitions of a name. All will resolve to the first, or to a single, global definition of a symbol.

One difference between `.EXTERN` and `.WEAK` references is that the linker does not extract objects from archives to satisfy weak references. Such references, left unresolved, have the value 0.

NOTE: The `.WEAK` (or `.GLOBAL` scope) directive is required to make symbols available for placement through `RESOLVE` commands in the `.ldf` file.

Assembler Command-Line Reference

This section describes the assembler command-line interface and switch set. It describes the assembler's switches, which are accessible from the operating system's command line or from the IDE.

This section contains:

- Running the Assembler
- Assembler Command-Line Switch Descriptions

Command-line switches control certain aspects of the assembly process, including debugging information, listing, and preprocessing. Because the assembler automatically runs the preprocessor as your program assembles (unless you use the `-sp` switch), the assembler's command line can receive input for the preprocessor program and direct its operation. For more information on the preprocessor, see the Preprocessor chapter.

NOTE: When developing a DSP project, you may find it useful to modify the assembler's default options settings. The way you set assembler options depends on the environment used to run the DSP development software.

See Specifying Assembler Options for more information.

Running the Assembler

To run the assembler from the command line, type the name of the appropriate assembler program followed by arguments (in any order), and the name of the assembly source file.

```
easm21K [-switch1 [-switch2 ]] sourceFile
easmblkfn [-switch1 [-switch2 ]] sourceFile
```

The ***Assembler Command Line Arguments*** table explains these arguments.

Table 2-16: Assembler Command Line Arguments

Argument	*Description*
`easm21K`	Name of the assembler program for SHARC and Blackfin processors, respectively.

Table 2-16: Assembler Command Line Arguments (Continued)

Argument	*Description*
`easmblkfn`	
`-switch`	Switch (or switches) to process. The command-line interface offers many optional switches that select operations and modes for the assembler and preprocessor. Some assembler switches take a file name as a required parameter.
`sourceFile`	Name of the source file to assemble.

The name of the source file to assemble can be provided as:

- *`short_file_name`* - a file name without quotes (no special characters)
- *`long_file_name`* - a quoted file name (may include spaces and other special path name characters)

The assembler outputs a list of command-line options when run without arguments (same as `-h[elp]`).

The assembler supports relative path names and absolute path names. When you specify an input or output file name as a parameter, follow these guidelines for naming files:

- Include the drive letter and path string if the file is not in the current project directory.
- Enclose long file names in double quotation marks; for example, "long file name".
- Append the appropriate file name extension to each file.

The ***File Name Extension Conventions*** table summarizes file extension conventions accepted by the IDE.

Table 2-17: File Name Extension Conventions

File Extension	*Description*
`.asm`	Assembly source file **NOTE:** The assembler treats files with unrecognized (or not existing) extensions as assembly source files.
`.is`	Preprocessed assembly source file
`.h`	Header file
`.lst`	Listing file
`.doj`	Assembled object file in ELF/DWARF-2 format
`.dat`	Data initialization file

Assembler command-line switches are case sensitive.

For example, the following command line

```
easmblkfn -proc ADSP-BF533 -l pList.lst -Dmax=100 -v -o bin/p1.doj p1.asm
```

runs the assembler with:

- `-proc ADSP-BF533` - specifies ADSP-BF533 as target processor
- `-l pList.lst` - directs the assembler to output a listing file
- `-Dmax=100` - defines the preprocessor macro `max` to be 100
- `-v` - displays verbose information on each phase of the assembly
- `-o bin/p1.doj` - specifies the name and directory for the assembled object file
- `p1.asm` - identifies the assembly source file to assemble

Assembler Command-Line Switch Descriptions

This section describes the assembler command-line switches in alphabetic order. A summary of the assembler switches appears in the ***Assembler Command-Line Switch Summary*** table. A detailed description of each assembler switch follows the table.

Table 2-18: Assembler Command-Line Switch Summary

Switch Name	*Purpose*
`-anomaly-detect {id1[, id2]|all|none}`	Enables detection of specified anomalies. See -anomaly-detect {id1[, id2...]|all|none} -anomaly-warn {id1[,id2]|all|none}.
`-anomaly-warn {id1[, id2]|all|none}`	Same as `-anomaly-detect`. See -anomaly-detect {id1[, id2...]|all|none} -anomaly-warn {id1[,id2]|all|none}. **NOTE:** Blackfin processors only.
`-anomaly-workaround {id1[, id2]|all|none}`	Enables workarounds for specified anomalies. See -anomaly-workaround {id1[, id2...]|all|none}.
`-Dmacro[=definition]`	Passes macro definition to the preprocessor. See -Dmacro[=definition].
`-dependency-add-target`	Adds a target to dependency output. See -dependency-add-target.
`-double-size-32`	Adds `/DOUBLE32` to the `.SECTION`s in the source file. See -double-size-32.
`-double-size-64`	Adds `/DOUBLE64` to the `.SECTION`s in the source file. See -double-size-64.
`-double-size-any`	Adds `/DOUBLEANY` to the `.SECTION`s in the source file. See -double-size-any.
`-expand-symbolic-links`	Enables support for Cygwin style paths. See -expand-symbolic-links.
`-expand-windows-shortcuts`	Enables support for Windows shortcuts. See -expand-windows-shortcuts.
`-file-attr attr [=value]`	Creates an attribute in the generated object file. See -file-attr attr[=val].
`-flags-compiler -opt1`	Passes each comma-separated option to the compiler. Used when compiling `.IMPORT` C header files. See -flags-compiler.
`-flags-pp -opt1`	Passes each comma-separated option to the preprocessor. See -flags-pp -opt1[, -opt2...].
`-g`	Generates debug information (DWARF-2 format). See -g.

Table 2-18: Assembler Command-Line Switch Summary (Continued)

Switch Name	*Purpose*
`-gnu-style-dependencies`	Produces dependency information compatible with the GNU Make tool. See -gnu-style-dependencies.
`-h[elp]`	Outputs a list of assembler switches. See -h[elp].
`-i|-I` *`directory_pathname`*	Searches a directory for included files. See -i.
`-l` *`filename`*	Outputs the named listing file. See -l filename.
`-li` *`filename`*	Outputs the named listing file with `#include` files expanded. See -li filename.
`-M`	Generates make dependencies for `#include` and data files only, but does not assemble. No object file is created. See -M.
`-MM`	Generates make dependencies for `#include` and data files. Use `-MM` for make dependencies with assembly. See -MM.
`-Mo` *`filename`*	Writes make dependencies to the *`filename`* specified. The `-Mo` option is for use with either the `-M` or `-MM` option. If `-Mo` is not present, the default is `<stdout>` display. See -Mo filename.
`-Mt` *`filename`*	Specifies the make dependencies target name. The `-Mt` option is for use with either the `-M` or `-MM` option. If `-Mt` is not present, the default is base name plus '`DOJ`'. See -Mt filename.
`-micaswarn`	Treats multi-issue conflicts as warnings. See -micaswarn. **NOTE:** Blackfin processors only.
`-no-anomaly-detect {id1[, id2...]|all|none}`	Does not issue a warning or an error for an anomaly ID. See -no-anomaly-detect {id1[, id2...]|all|none}.
`-no-anomaly-workaround {id1[, id2...]|all|none}`	Does not implement a workaround for an anomaly id. See -no-anomaly-workaround {id1[, id2...]|all|none}.
`-no-expand-symbolic-links`	Disables support for Cygwin style paths. See -no-expand-symbolic-links.
`-no-expand-windows-shortcuts`	Disables support for Windows shortcuts. See -no-expand-windows-shortcuts.
`-no-source-dependency`	Suppresses output of the source filename in the dependency output produced when `-M` or `-MM` is specified. See -no-source-dependency.
`-no-temp-data-file`	Suppresses writing temporary data to a disk file. See -no-temp-data-file. **NOTE:** Blackfin processors only.
`-normal-word-code` `-nwc`	Encodes input sections bearing the `/PM` qualifier (for execution from normal word memory. See SHARC-Specific Qualifiers and -normal-word-code or -nwc. **NOTE:** ADSP-214xx processors only.
`-o` *`filename`*	Outputs the named object [binary] file. See -o filename.
`-path-compiler` *`pathname`*	Specifies which compiler to invoke when processing `.IMPORT` directives. See -path-compiler.

Table 2-18: Assembler Command-Line Switch Summary (Continued)

Switch Name	*Purpose*
`-pp`	Runs the preprocessor only; does not assemble. See -pp.
`-proc` *processor*	Specifies a target processor for which the assembler should produce suitable code. See -proc processor.
`-save-temps`	Saves intermediate files. See -save-temps.
`-short-word-code` `-swc`	Encodes input sections bearing the `/PM` qualifier for execution from short word memory. See SHARC-Specific Qualifiers and -normal-word-code or -nwc. **NOTE:** ADSP-214xx processors only.
`-si-revision` *version*	Specifies silicon revision of the specified processor. See -si-revision version.
`-sp`	Assembles without preprocessing. See -sp.
`-stallcheck={none|cond|all}`	Displays stall information: • `none` - no messages • `cond` - conditional stalls only (default) • `all` - all stall information See -stallcheck. **NOTE:** Blackfin processors only.
`-swc-exclude` *name1[, name2]*	Excludes the named section(s) from the effect of the `-short-word-code` (`-swc`) switch. See -swc-exclude name1[, name2]. **NOTE:** ADSP-214xx processors only.
`-v` or `-verbose`	Displays information on each assembly phase. See -v[erbose].
`-version`	Displays version information for the assembler and preprocessor programs. See -version.
`-w`	Disables all assembler-generated warnings. See -w.
`-Werror` *number[, number]*	Selectively turns assembler messages into errors. See -Werror number[, number].
`-Winfo` *number[, number]*	Selectively turns assembler messages into informationals. See -Winfo number[, number].
`-Wno-info`	Does not display informational assembler messages. See -Wno-info.
`-Wnumber` *[, number]*	Selectively disables warnings by one or more message numbers. For example, `-W1092` disables warning message `ea1092`. See -Wnumber[, number].
`-Wsuppress` *number[, number]*	Selectively turns off assembler messages. See -Wsuppress number[, number].
`-Wwarn` *number[, number]*	Selectively turns assembler messages into warnings. See -Wwarn number[, number].
`-Wwarn-error`	Displays all assembler warning messages as errors. See -Wwarn-error.

-anomaly-detect {id1[, id2...] | all | none} -anomaly-warn {id1[,id2] | all | none}

The `-anomaly-detect` and `-anomaly-warn` switches direct the assembler to check for assembly code that may be affected by any of the hardware anomalies listed in the `id` switches.

NOTE: The `-anomaly-warn` switch is used with Blackfin processors only.

The `id` switch parameter is:

`id` - Anomaly identifier. The anomaly identifier syntax can use or omit dashes. For example, either `05-00-0245` or `05000245` are accepted. Additionally, ids specified override the `all` and `none` options. For example, `-anomaly-detect none,050000245` detects only anomaly 05000245.

`all` - use all identifiers applicable to the given processor

`none` - ignore all identifiers not specifically provided (same as `-no-anomaly-detect all`)

If the check detects assembly code that will be affected by the specified anomaly (or anomalies), the assembler issues a warning. Using this option helps you detect and avoid code combinations affected by anomalies.

NOTE: A warning can also be issued if the assembler always implements a code workaround for an anomaly, instead of just performing a check.

The most recent list of anomalies can be obtained from Software and Tools Anomalies search page on the Analog Devices website at:

http://www.analog.com/blackfin-anomalies.

-anomaly-workaround {id1[, id2...] | all | none}

The `-anomaly-workaround` switch directs the assembler to switch on workarounds for specific anomalies. Switch parameters are:

`id` - anomaly identifier (for example, `05-00-0245` or `05000245`)

`all` - uses all identifiers applicable to the given processor (same as `-no-anomaly-workaround none`)

`none` - ignore all identifiers not specifically provided (same as `-no-anomaly-workaround all`)

The anomaly identifier syntax may use or omit dashes. For example, either `05-00-0245` or `05000245` are accepted. Additionally, specified ids override the `all` and `none` options. For example, `-anomaly-workaround none, 050000245` works around only anomaly 050000245.

The workaround may result in the assembler altering the user assembly code so that it cannot encounter the anomaly. The assembler may issue a message to indicate that it has altered the user assembly code. This option overrules any default behavior for the anomaly.

-Dmacro[=definition]

The -D (define macro) switch directs the assembler to define a macro and pass it to the preprocessor. See Using Assembler Feature Macros for the list of predefined macros.

For example,

```
-Dinput         // defines input as 1
-Dsamples=10    // defines samples as 10
-Dpoint='Start' // defines point as the string `Start'
```

-dependency-add-target

The -dependency-add-target switch directs the assembler to add the file name that follows the switch to the list of targets when generating a list of dependencies. The switch is used when dependencies are being generated, by invoking the -M or -MM switch.

-double-size-32

The -double-size-32 switch directs the assembler to add /DOUBLE32 to .SECTIONs in the source file that do not have double size qualifiers. For .SECTIONs in the source file that already have a double size qualifier, this option is ignored and a warning is produced. For more information, see .SECTION, Declare a Memory Section.

-double-size-64

The -double-size-64 switch directs the assembler to add /DOUBLE64 to .SECTIONs in the source file that do not have double size qualifiers. For .SECTIONs in the source file that already have a double size qualifier, this option is ignored and a warning is produced. During linking, the linker ensures that all of the sections have a consistent interpretation of the size of the C type double. If the sections are not consistent, the linker generates a warning.

Linker Warning Example:

```
[Warning li2151] Input sections have inconsistent qualifiers as follows.
```

For more information, see .SECTION, Declare a Memory Section.

-double-size-any

The -double-size-any switch directs the assembler to add /DOUBLEANY to .SECTIONs in the source file that do not have double size qualifiers, making SECTION contents independent of size of double type.
For .SECTIONs in the source file that already have a double size qualifier, this option is ignored and a warning is produced. For more information, see .SECTION, Declare a Memory Section.

-expand-symbolic-links

The -expand-symbolic-links switch directs the assembler to correctly access directories and files whose name or path contain Cygwin path components.

-expand-windows-shortcuts

The `-expand-windows-shortcuts` switch directs the assembler to correctly access directories and files whose name or path contain Windows shortcuts.

-file-attr attr[=val]

The `-file-attr` (file attribute) switch directs the assembler to add an attribute (`attr`) to the object file. The attribute will be given the value (`val`) or "1" if the value is omitted. `Attr` should follow the rules for naming symbols. `Val` should be double-quoted unless it follows the rules for naming symbols. See Assembler Keywords and Symbols for more information on naming conventions.

-flags-compiler

The `-flags-compiler -opt1 [,-opt2 ]` switch passes each comma-separated option to the C compiler when compiling `.IMPORT` headers.

For example,

```
// file.asm has .IMPORT "myHeader.h"
easmbkln -proc ADSP-BF533 -flags-compiler -I/Path -I. file.asm
```

The rest of the assembly program, including its `#include` files, are processed by the assembler preprocessor. The `-flags-compiler` switch processes a list of one or more valid C compiler options, including the `-D` and `-I` options.

User-Specified Defines Options

`-D` (defines) options in an assembler command line are passed to the assembler preprocessor, but they are not passed to the compiler for `.IMPORT` header processing. If `#defines` are used for `.IMPORT` header compilation, they must be explicitly specified with the `-flags-compiler` switch.

For example,

```
       // file.asm has .IMPORT "myHeader.h"
easmblkfn -proc ADSP-BF533 -DaDef -flags-compiler -DbDef, -DbDefTwo=2 file.asm
       // -DaDef is not passed to the compiler
ccblkfn -proc ADSP-BF533 -c -debug-types -DbDef -DbDefTwo=2 myHeader.h
```

NOTE: See Using Assembler Feature Macros for the list of predefined macros, including default macros.

Include Options

The `-I` (include search path) options and `-flags-compiler` arguments are passed to the C compiler for each `.IMPORT` header compilation. The compiler `include` path is always present automatically.

Use the `-flags-compiler` switch to control the order that the `include` directories are searched. The `-flags-compiler` switch attributes take precedence from the assembler's `-I` options.

For example,

```
easmblkfn -proc ADSP-BF533 -I/aPath -DaDef -flags-compiler -I/cPath,-I. file.asm
ccblkfn -proc ADSP-BF533 -c -debug-types -I/cPath -I. myHeader.h
```

The `.IMPORT` C header files are preprocessed by the C compiler preprocessor. The `struct` headers are standard C headers, and the standard C compiler preprocessor is needed. The rest of the assembly program (including its `#include` files) are processed by the assembler preprocessor.

Assembly programs are preprocessed using the `pp` preprocessor (the assembler/linker preprocessor) as well as `-I` and `-D` options from the assembler command line. However, the `pp` call does not receive the `-flags-compiler` switch options.

-flags-pp -opt1[, -opt2...]

The `-flags-pp` switch passes each comma-separated option to the preprocessor.

NOTE: Use `-flags-pp` with caution. For example, if `pp` legacy comment syntax is enabled, the comment characters become unavailable for non-comment syntax.

-g

The `-g` (generate debug information) switch directs the assembler to generate data type information for arrays, functions, and C structs. This switch also generates DWARF2 function information with starting and ending ranges based on the `myFunc: myFunc.end:` label boundaries, as well as line number and symbol information in DWARF2 binary format, allowing you to debug the assembly source files.

When the assembler's `-g` switch is in effect, the assembler produces a warning when it is unable to match a `*.end` label to a matching beginning label. This feature can be disabled using the `-Wnumber[, number]` switch.

WARNING ea1121: Missing End Labels

Warning `ea1121` occurs on assembly file debug builds (using the `-g` switch) when a globally-defined function or label for a data object is missing its corresponding ending label, with the naming convention label + "`.end`".

For example,

```
[Warning ea1121] "./gfxeng_thickarc.asm":42 _gfxeng_thickarc:
          -g assembly with global function without ending label. Use
          '_gfxeng_thickarc.end' or '_gfxeng_thickarc.END' to mark the
          ending boundary of the function for debugging information for
          automated statistical profiling of assembly functions.
```

The ending label marks the boundary of the end of a function. Compiled code automatically provides ending labels. Hand-written assembly code needs to have the ending labels explicitly added to tell the tool chain where the ending boundary is. This information is used to automate statistical profiling of assembly functions. It is also needed by the linker to eliminate unused functions and other features.

To suppress a specific assembler warning by unique warning number, the assembler provides the `-Wsuppress 1121` switch.

NOTE: It is highly recommended that warning `ea1121` ***not*** be suppressed and the code be updated to have ending labels.

Functions (Code):

```
_gfxeng_vertspan:
   [--sp] = fp;
   ...
   rts;
```

Add an ending label after `rts;`. Use the suffix "`.end`" and begin the label with "`.`" to have it treated as an internal label that is not displayed in the debugger.

```
.global _gfxeng_vertspan;
_gfxeng_vertspan:
   [--sp] = fp;
   ...
   rts;
._gfxeng_vertspan.end:
```

-gnu-style-dependencies

The `-gnu-style-dependencies` switch directs the assembler to produce dependency information that is compatible with the GNU Make tool. This switch is used when dependencies are being generated, by invoking the `-M` or `-MM` switch. This switch ensures that all backslashes in dependency paths are converted to forward slashes, that the names of target and dependencies have their case preserved, whitespace in file names is escaped using backslashes, and all path names are normalized to be relative to the current working directory of the tool.

-h[elp]

The `-h` (or `-help`) switch directs the assembler to print a list of command-line switches with a syntax summary to standard output.

-i

The `-i` *`directory`* (or `-I`) switch (include directory path) directs the assembler to append the specified directory (or a list of directories separated by semicolons "`;`") to the search path for included files.

NOTE: No space is allowed between `-i` and the path name.

These files are:

- Header files (`.h`) included with the `#include` preprocessor command
- Data initialization files (`.dat`) specified with the `.VAR` assembly directive

The assembler passes this information to the preprocessor; the preprocessor searches for included files in the following order:

1. Directory for assembly program

2. <*processor_family*>/include subdirectory of the installation directory
3. Specified directory (or list of directories). The order of the list defines the order of multiple searches.

Relative paths are based on the path of the assembly source, not the directory of the project. Usage of full path names for the -I switch on the command line is recommended.

For example,

```
easm21K -proc ADSP-21161 -I "/bin/SHARC/include" file.asm
```

-l filename

The -l *filename* (listing) switch directs the assembler to generate the named listing file. Each listing file (.lst) shows the relationship between your source code and instruction opcodes that the assembler produces.

For example,

```
easmblkfn -proc ADSP-BF533 -I <file_path> -I. -l file.lst file.asm
```

The file name is a required argument to the -l switch. For more information, see Reading a Listing File.

-li filename

The -li (listing) switch directs the assembler to generate the named listing file with #include files. The file name is a required argument to the -li switch. For more information, see Reading a Listing File.

-M

The -M (generate make rule only) assembler switch directs the assembler to generate make dependency rules, suitable for the make utility, describing the dependencies of the source file. No object file is generated for -M assemblies. For make dependencies with assembly, use the -MM switch.

The output, an assembly make dependencies list, is written to stdout in the standard command-line format:

"target_file": "dependency_file.ext"

dependency_file.ext may be an assembly source file, a header file included with the #include preprocessor command, a data file, or a header file imported via the .IMPORT directive.

The -Mo *filename* switch writes make dependencies to the *filename* specified instead of <stdout>. For consistency with the compilers, when -o *filename* is used with -M, the assembler outputs the make dependencies list to the named file. The -Mo *filename* takes precedence if both -o *filename* and -Mo *filename* are present with -M.

-MM

The -MM (generate make rule and assemble) assembler switch directs the assembler to output a rule, suitable for the make utility, describing the dependencies of the source file. The assembly of the source into an object file proceeds normally. The output, an assembly make dependencies list, is written to stdout. The only difference between -MM and -M actions is that the assembling continues with -MM. See -M for more information.

-Mo filename

The -Mo (output make rule) assembler switch specifies the name of the make dependencies file that the assembler generates when you use the -M or -MM switch. If -Mo is not present, the default is <stdout> display. If the named file is not in the current directory, you must provide the path name in double quotation marks (" ").

NOTE: The -Mo filename switch takes precedence over the -o filename switch.

-Mt filename

The -Mt *filename* (output make rule for named object) assembler switch specifies the name of the object file for which the assembler generates the make rule when you use the -M or -MM switch. If the named file is not in the current directory, you must provide the path name. If -Mt is not present, the default is the base name plus the .doj extension. See -M for more information.

-micaswarn

The -micaswarn switch treats multi-issue conflicts as warnings.

NOTE: This switch is used with Blackfin processors only.

-no-anomaly-detect {id1[, id2...] | all | none}

The -no-anomaly-detect switch directs the assembler to switch off any check for a specific anomaly ID in the assembler. No assembler warning or error will be issued when the assembler encounters assembly code that the anomaly will have an impact upon. This option overrules any default behavior for the anomaly.

The switch parameter is:

id - anomaly identifier (for example, 05-00-0245 or 05000245)

all - ignore all identifiers not specifically provided (same as -anomaly-detect none))

none - use all identifiers applicable to the given processor (same as -anomaly-detect all)

The anomaly identifier syntax may use or omit dashes. For example, either 05-00-0245 or 05000245 are accepted. Additionally, specified ids override the all and none options. For example, -no-anomaly-detect all, 050000245 detects only anomaly 050000245.

A warning may be issued if the assembler always implements a workaround for the anomaly instead of a check.

-no-anomaly-workaround {id1[, id2...] | all | none}

The -no-anomaly-workaround switch directs the assembler to switch off any workaround for a specific anomaly id in the assembler. The assembler will not alter the user assembly code so that it cannot encounter the anomaly. This option overrules any default behavior for the anomaly.

The switch parameter is:

id - anomaly identifier (for example, 05-00-0245 or 05000245)

`all` - ignore all identifiers not specifically provided (same as `-anomaly-workaround none`)

`none` - use all identifiers applicable to the given processor (same as `-anomaly-workaround all`)

The anomaly identifier syntax may use or omit dashes. For example, either `05-00-0245` or `05000245` are accepted. Additionally, specified ids override the `all` and `none` options. For example, `-no-anomaly-workaround all,050000245` works around only anomaly 050000245.

A warning may be issued if the assembler always checks for the anomaly and has no workaround.

-no-expand-symbolic-links

The `no-expand-symbolic-links` switch directs the assembler not to expand directories or files whose name or path contain Cygwin path components.

-no-expand-windows-shortcuts

The `-no-expand-windows-shortcuts` switch directs the assembler not to expand directories or files whose name or path contain Windows shortcuts.

-no-source-dependency

The `-no-source-dependency` switch directs the assembler not to print anything about dependency between the `.asm` source file and the `.doj` object file when outputting dependency information. This switch can only be used in conjunction with the `-M` or `-MM` switch.

-no-temp-data-file

The `-no-temp-data-file` switch directs the assembler not to write temporary data to a disk.

As part of a space saving measure, the assembler stores all data declarations into a file. This is to allow large sources to assemble more quickly by freeing valuable memory resources. By default, the temporary data files are stored into the system temporary folder (for example, `C:/Documents and Settings/User/Local Settings/Temp`) and is given the prefix "`EasmblkfnNode`"). These files are removed by the assembler but, if for any reason the assembler does not complete, these files will not be deleted and persist in the temporary folder. These files can always be safely deleted in such circumstances after the assembler has stopped.

This command-line option allows the user to turn off this default feature. When turned off, all data is stored into internal memory and not written to the disk.

-normal-word-code or -nwc

The `-normal-word-code` or `-nwc` switch directs the assembler to encode input sections bearing the `/PM` qualifier (see SHARC-Specific Qualifiers) for execution from normal word memory.

NOTE: The switch is used with the ADSP-214xx SHARC processors only.

-o filename

The `-o` *`filename`* (output file) switch directs the assembler to use the specified *`filename`* argument as the output file. This switch names the output, whether for conventional production of an object, a preprocessed, assemble-produced file (`.is`), or make dependency (`-M`). By default, the assembler uses the root input file name for the output and appends a `.doj` extension.

Some examples of this switch syntax are:

```
easmblkfn -proc ADSP-BF533 -pp -o test1.is test.asm
          // preprocessed output goes into test1.is

easmblkfn -proc ADSP-BF533 -o -debug/prog3.doj prog3.asm
          // specify directory and filename for the object file
```

-path-compiler

The `-path-compiler` *`pathname`* switch allows you to specify the compiler to be invoked when processing headers in `.IMPORT` directives. By default, the assembler will invoke the compiler for the appropriate architecture in its CrossCore Embedded Studio installation.

For example:

```
easmbkln -proc ADSP-BF533 -path-compiler C:\CCES_2.0.0\ccblkfn.exe file.asm
```

-pp

The `-pp` (proceed with preprocessing only) switch directs the assembler to run the preprocessor, but stop without assembling the source into an object file. When assembling with the `-pp` switch, the `.is` file is the final result of the assembly. By default, the output file name uses the same root name as the source, with the `.is` extension.

-proc processor

The `-proc` *`processor`* (target processor) switch instructs the assembler to produce code suitable for the specified processor.

The *`processor`* identifiers directly supported by CCES are listed in the online help.

For example,

```
easm21K   -proc ADSP-21161 -o bin/p1.doj p1.asm
easmblkfn -proc ADSP-BF533 -o bin/p1.doj p1.asm
```

NOTE: See also the description of the -si-revision version switch for specifying a particular silicon revision of a processor

-save-temps

The `-save-temps` (save intermediate files) switch directs the assembler to retain intermediate files generated and normally removed as part of the assembly process.

-short-word-code or -swc

The `-short-word-code` or `-swc` switch directs the assembler to encode input sections bearing the `/PM` qualifier (see SHARC-Specific Qualifiers) for execution from short word memory.

NOTE: The switch is used with the ADSP-214xx SHARC processors only.

-si-revision version

The `-si-revision` *`version`* (silicon revision) switch directs the assembler to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The *`version`* parameter represents a silicon revision for the processor specified by the `-proc processor` switch.

For example,

```
easmblkfn -proc ADSP-BF533 -si-revision 0.6
```

If silicon version "`none`" is used, no errata workarounds are enabled, whereas specifying silicon version "`any`" enables all errata workarounds for the target processor.

If the `-si-revision` switch is not used, the assembler will build for the target processor's latest known silicon revision and will enable any errata workarounds appropriate for the latest silicon revision.

The `__SILICON_REVISION__` macro is set by the assembler to two hexadecimal digits representing the major and minor numbers in the silicon revision. For example, `1.0` becomes `0x100` and `10.21` becomes `0xa15`.

If the silicon revision is set to "`any`", the `__SILICON_REVISION__` macro is set to `0xffff`. If the `-si-revision` switch is set to "`none`", the assembler will not set the `__SILICON_REVISION__` macro.

-sp

The `-sp` (skip preprocessing) switch directs the assembler to assemble the source file into an object file without running the preprocessor. When the assembler skips preprocessing, no preprocessed assembly file (`.is`) is created.

-stallcheck

The `-stallcheck=`*`option`* switch provides the following choices for displaying stall information. See the ***-stallcheck Options*** table.

Table 2-19: -stallcheck Options

-stallcheck Option	*Description*
`-stallcheck=none`	Displays no messages for stall information. Default setting for SHARC processors.
`-stallcheck=cond` (Blackfin processors only)	Displays information about conditional stalls only. Default setting for Blackfin processors.
`-stallcheck=all`	Displays information for all detected stalls.

-swc-exclude name1[, name2]

The `-swc-exclude` *`name`* switch directs the assembler to exclude the named input section(s) from the effect of the `-short-word-code`/`-swc` switch.

NOTE: The switch is used with the ADSP-214xx SHARC processors only.

-v[erbose]

The `-v` (or `-verbose`) switch directs the assembler to display version and command-line information for each phase of assembly.

-version

The `-version` (display version) switch directs the assembler to display version information for the assembler and preprocessor programs.

-w

The `-w` (disable all warnings) switch directs the assembler not to display warning messages generated during assembly.

-Werror number[, number]

The `-Werror` *`number`* switch turns the specified assembler messages into errors. For example, "`-Werror 1177`" turns warning message `ea1177` into an error. This switch optionally accepts a list, such as `[,number]`.

NOTE: Some error messages cannot be altered in severity.

-Winfo number[, number]

The `-Winfo` *`number`* switch turns the specified assembler messages into informational messages. For example, "`-Winfo 1177`" turns warning message `ea1177` into an informational message. This switch optionally accepts a list, such as `[,number]`.

NOTE: Some error messages cannot be altered in severity.

-Wno-info

The `-Wno-info` switch turns off all assembler informational messages.

-Wnumber[, number]

The `-W` (warning suppression) switch selectively disables warnings specified by one or more message numbers. For example, `-W1092` disables warning message `ea1092`. Optionally, this switch accepts a list, such as `[,number]`.

-Wsuppress number[, number]

The `-Wsuppress number` switch selectively turns off assembler messages. For example, "`-Wsuppress 1177`" turns off warning message `ea1177`. Optionally, this switch accepts a list, such as `[,number]`.

NOTE: Some error messages cannot be altered in severity.

-Wwarn number[, number]

The `-Wwarn number` switch turns the specified assembler messages into warnings. For example, "`-Wwarn 1154`" turns error message `ea1154` into a warning. Optionally, this switch accepts a list, such as `[,number]`.

NOTE: Some error messages cannot be altered in severity.

-Wwarn-error

The `-Wwarn-error` switch turns all assembler warnings into errors.

Specifying Assembler Options

Within the CCES IDE, specify settings for projects. Select a project in any project navigation view and choose ***Properties*** > ***C/C++ Build*** > ***Settings***. Use the ***Settings*** property pages to select the target processor, configure tool-chain, build steps, and other settings for your project.

Use the assembler property pages of ***Tool Settings*** to specify assembler options used while building your project.

Most dialog box options have corresponding assembler command-line switches described in Assembler Command-Line Switch Descriptions.

Use the ***Additional Options*** page to enter appropriate command-line switches, file names, and options that do not have corresponding controls on the assembler property pages but are available via command-line invocation.

Assembler options direct calls to an assembler when assembling `.asm` files. Changing assembler options in the IDE does not affect the assembler calls made by the compiler during the compilation of `.c`/`.cpp` files.

For more information, refer to the CCES help.

3 Preprocessor

The preprocessor program (`pp.exe`) evaluates and processes preprocessor commands in source files on all supported processors. The preprocessor commands direct the preprocessor to define macros and symbolic constants, include header files, test for errors, and control conditional assembly and compilation. The preprocessor supports ANSI C standard preprocessing with extensions, such as `?` and `....`

The preprocessor is run by other build tools (assembler and linker) from the operating system's command line or from within the IDE. The `pp` preprocessor can also operate from the command line with its own command-line switches.

This chapter contains:

- Preprocessor Guide

  Contains the information on building programs.

- Preprocessor Command Reference

  Describes the preprocessor's commands, with syntax and usage examples.

- Preprocessor Command-Line Reference

  Describes the preprocessor's command-line switches, with syntax and usage examples.

Preprocessor Guide

This section describes `pp` preprocessor information used when building programs from a command line or from within the IDE. Software developers who use the preprocessor should be familiar with:

- Writing Preprocessor Commands
- Header Files and #include Command
- Writing Macros
- Using Predefined Preprocessor Macros
- Specifying Preprocessor Options

Compiler Preprocessor

The compiler has it own preprocessor that enables the use of preprocessor commands within C/C++ source. The compiler preprocessor automatically runs before the compiler. This preprocessor is separate from the assembler preprocessor and has some features that may not be used within your assembly source files. For more information, refer to the *C/C++ Compiler and Library Manual* for the target processor.

Assembler Preprocessor

The assembler preprocessor differs from the ANSI C standard preprocessor in several ways. First, the assembler preprocessor supports a "?" operator (see ? (Generate a unique label)) that directs the preprocessor to generate a unique label for each macro expansion. Second, the assembler preprocessor does not treat "." as a separate token. Instead, "." is treated as part of an identifier. This behavior matches the assembler's behavior, which uses "`.`" to start directives and accepts "`.`" in symbol names. For example, the following command sequence:

```
#define VAR my_var
.VAR x;
```

does not cause any change to the variable declaration. The text "`.VAR`" is treated as a single identifier which does not match the macro name `VAR`.

The standard C preprocessor treats `.VAR` as two tokens ("`.`" and "`VAR`") and makes the following substitution:

```
.my_var x;
```

The assembler preprocessor also produces assembly-style strings (single-quote delimiters) instead of C-style strings.

Finally, under command-line switch control, the assembler preprocessor supports legacy assembler commenting formats ("`!`" and "`{ }`").

Writing Preprocessor Commands

Preprocessor commands begin with a pound sign (`#`) and extend to the end of the line. The pound sign must be the first non-white space character on the line containing the command. If the command is longer than one line, use a backslash (`\`) and a carriage return to continue the command on the next line. Do not place any characters between the backslash and the line ending. Unlike assembly directives, preprocessor commands are case sensitive and must be lowercase.

For more information on preprocessor commands, see Preprocessor Command-Line Reference.

For example,

```
#include "string.h"
#define MAXIMUM 100
```

When the preprocessor runs, it modifies the source code by:

- Including system header files and user-defined header files
- Defining macros and symbolic constants

- Providing conditional assembly

Specify preprocessing options with preprocessor commands-lines that start with a `#` character. In the absence of commands, the preprocessor performs these three global substitutions:

- Replaces comments with single spaces
- Deletes line continuation characters (`\`)
- Replaces macro references with corresponding expansions

The following cases are notable exceptions to the described substitutions:

- The preprocessor does not recognize comments or macros within the file name delimiters of an `#include` command.
- The preprocessor does not recognize comments or predefined macros within a character or string constant.

Header Files and #include Command

Header (`.h`) files contain lines of source code to be included (textually inserted) into another source file. Typically, header files contain declarations and macro definitions.

The `#include` preprocessor command includes a copy of the header file at the location of the command. There are three forms for the `#include` command, as described in:

- Include Path Search
- System Header Files
- User Header Files
- Sequence of Tokens

System Header Files

Syntax: `#include <`*`filename`*`>`

The file name is placed between a pair of angle bracket characters. The file name in this form is interpreted as a system header file. These files are used to declare global definitions, especially memory-mapped registers, system architecture, and processors.

Example:

```
#include <device.h>
```

System header files are installed in the <*processor*>`/include` folder for the processor family in the CCES installation.

User Header Files

Syntax: `#include "`*`filename`*`"`

The file name is placed within a pair of double quote characters. The file name in this form is interpreted as a user header file. These files contain declarations for interfaces between the source files of the program.

Example:

```
#include "def.h"
#include "fft_ovly.h"
```

Sequence of Tokens

Syntax: `#include` *`text`*

In this case, *`text`* is a sequence of tokens subject to macro expansion by the preprocessor.

It is an error if after macro expansion the text does not match one of the two header file forms. If the text on the line after the `#include` is not placed between double quotes (as a user header file) or between angle brackets (as a system header file), the preprocessor performs macro expansion on the text. After that expansion, the line requires either of the two header file forms.

NOTE: Unlike most preprocessor commands, the text after the `#include` is available for macro expansion.

Examples:

```
    /* define preprocessor macro with name for include file */
#define includefilename "header.h"
    /* use the preprocessor macro in an #include command */
#include includefilename
    /* the code above evaluates to #include "header.h" */

    /* define preprocessor macro to build system include file */
#define syshdr(name) <name ## .h>
    /* use the preprocessor macro in a #include command */
#include syshdr(adi)
    /* the code above evaluates to #include <adi.h> */
```

Include Path Search

It is good programming practice to distinguish between system header files and user header files. The only technical difference between the two different notations is the directory search order that the assembler follows to locate the specified header file.

For example, when using Blackfin processors, the `#include <`*`file`*`>` search order is:

1. The include path specified by the `-I` switch
2. <*`install_path`*>`/Blackfin/include` folders

The `#include "file"` search order is:

1. The local directory - the directory in which the source file resides
2. The include path specified by the `-I` switch

3. <*install_path*>/Blackfin/include folders

If you use the -I and the -I- switches on the command line, the system search path (#include < >) is modified in such a manner that search directories specified with the -I switch that appear before the directory specified with the -I- switch are ignored. For syntax information and usage examples on the #include preprocessor command, see #include.

Writing Macros

The assembler/linker preprocessor processes macros in assembly source files and linker description files (.ldf). Macros provide for text substitution.

The term *macro* defines a macro-identifying symbol and its corresponding definition that the preprocessor uses to substitute the macro reference(s).

For example, use macros to define symbolic constants or to manipulate register bit masks in an assembly program based on a macro argument, as follows:

```
/* Define a symbolic constant */
#define MAX_INPUT 256

/* Mask peripheral #x interrupt */
#define SIC_MASK(x) (1 << ((x)&0x1F))
```

Macros can be defined to repeat code sequences in assembly source code. When you pass parameters to a code macro, the macro serves as a general-purpose routine that is usable in many different programs. The block of instructions that the preprocessor substitutes can vary with each new set of arguments.

A macro differs from a subroutine call. During assembly, each instance of a macro inserts a copy of the same block of instructions, so multiple copies of that code appear in different locations in the object code. By comparison, a subroutine appears only once in the object code, and the block of instructions at that location is executed for every call.

For more information, see:

- #define
- Macro Definition and Usage Guidelines
- Examples of Multi-Line Code Macros With Arguments
- Debugging Macros

Macro Definition and Usage Guidelines

A macro definition can be any text that may occur legally in the source file that references the macro. In assembly files, the macro may expand to include instructions, directives, register names, constants, and so on. In LDFs, a macro may expand to include LDF commands, memory descriptions and other items that are legal in an LDF. The macro definition may also have other macro names that are replaced with their own definitions.

The following guidelines are provided to help you construct macros and use them appropriately.

- A ***macro definition*** must begin with `#define` and must end with a carriage return.
- ***Macro termination***. If a macro definition ends with a terminator on the instruction (one semicolon (`;`) for SHARC and Blackfin processors), do not place a terminator at the end of the macro (usage) in an assembly statement. However, if a macro definition does not end with a terminator, each instance of the macro usage must be followed by the terminator in the assembly statement.

  Be consistent with regard to how you use terminators in macro definitions.

NOTE: Examples shown in this section omit the terminator in the macro definition and use the terminator in the assembly text. Note that the mac; statement in the following Blackfin example has a ";".

```
#define mac mrf = mrf+R2*R5(ssfr) // macro definition

R2 = R1-R0;                       // macro usage; set parameters
R5 = DM(I1,M0);
mac;
```

- ***Line continuation***. A macro definition can be split across multiple lines for readability. When a macro definition is longer than one line, place a backslash (`\`) character at the end of ***each*** line (except the last line) for line continuation.

  Incorrect:

  ```
  #define MultiLineMacro
     instruction1; \
     instruction2; \
     instruction3
  ```

  Notice that the backslash in the `#define` line is missing.

  Correct:

  ```
  #define MultiLineMacro \
     instruction1; \
     instruction2; \
     instruction3
  ```

  No characters are permitted on a line after a backslash. A warning is generated when there is white space after what might have been intended as a line continuation.

  For example,

  ```
  #define macro1 \
      instruction1; \ (whitespace)
      instruction2; \
      instruction3

      [Warning pp0003] "header.h":3
      The backslash at the end of this line
      is followed by whitespace
      It is not a line continuation
  ```

- ***Comments within #define***. Use C-style comments (`/* comment */`) within multi-line macros. Otherwise, the line-continuation character (`\`) will cause the next line to be concatenated to the comment, thus becoming part of the comment.

  The preprocessor supports C-style comments (`/* comment */`) as well as C++-style comments (`// comment`). The C-style comment has a delimiter at the start and end of the comment; the C++-style comment begins at the "`//`" and terminates at the end of the line.

  The "terminates at the end of the line" aspect of C++-style comments renders "`//`" comments unsuitable within multi-line macro definitions. The line continuation character causes the next line to be concatenated to the comment, thus becoming part of the comment.

  The following code fragment demonstrates the problem.

  ```
  #define macro \
  first line; \
  second line
  ```

  when expanded by writing "`macro`" in your `.asm` file, this code becomes:

  ```
  first line; second line
  ```

  If you use C-style comments, you can write:

  ```
  #define macro \
  /* this macro has two lines */ \
  first line; \
  /* and two comments */ \
  second line
  ```

  which will expand to:

  ```
  first line; second line
  ```

  However, if you use C++ style comments as shown below:

  ```
  #define macro \
  // this comment will devour the rest of the macro \
  first line; \
  second line
  ```

  the macro expands into an "empty" macro.

  In the code above, the first line of the macro definition starts a comment. Since there are line-continuation characters, the logical end of line for that comment is the end of the macro. Thus, the code yields an "empty" macro.

- Macro nesting (macros called within another macro) is limited only by the memory available during preprocessing. Recursive macro expansion, however, is not allowed.

Refer to `#define` for more information on the `#define` command.

Examples of Multi-Line Code Macros With Arguments

The following are examples of multi-line code macros with arguments.

Blackfin Code Example:

```
#define false 0
#define xchg(xv,yv)     \
   P0=xv;               \
   P1=yv;               \
   R0=[P0];             \
   R1=[P1];             \
   [P1]=R0;             \
   [P0]=R1
```

SHARC Code Example:

```
#define ccall(x)        \
   R2=I6; I6=I7;        \
   JUMP (pc, x) (db);   \
   DM(I7,M7)=R2;        \
   DM(I7, M7)=PC
```

Macro Usage in Code Section:

```
        <instruction code here>
        ccall(label1);
        <instruction code here>
label1: NOP;
        <instruction code here>
```

Debugging Macros

If you get an unexpected syntax error from the assembler on a macro expansion, it can be helpful to debug the macro by looking at what the preprocessor produced post preprocessing. The intermediate file produced by the preprocessor is the `.is` output file.

From the IDE, select the ***Save temporary files (-save-temps)*** check box on the ***General*** assembler page of the ***Tool Settings*** dialog box. If you are running the assembler from the command line, add the `-save-temps` switch (see -save-temps in the Assembler chapter).

Tips for Debugging Macros:

Assembly programmers may find it useful to include the processor system header files for predefined macros that are helpful to assembly language programmers for that processor family. These are known as "`def` headers". For example, an ADSP-BF534 programmer uses:

```
//Header is located in <install_path>/Blackfin/include
#include "defBF534.h"
```

A symbol in your program may inadvertently use the same spelling as a `#define` in the `def` header. Typically, this results in a syntax error due to the symbol being replaced with a constant or constant expression, which is not what you intended.

For example, `defBF534.h` contains:

```
  #define  ALARM 0x0002 /* Alarm Interrupt Enable */
```

If an assembly program uses `ALARM` as a symbol name, it will get a textual replacement of "`0x0002`", making the program illegal, as demonstrated by the following code fragment.

```
#include "defBF534.h"
#define FALSE 0
#define TRUE 1

.SECTION data1;

.VAR ALARM = FALSE;

[Error ea5004] "alarm.asm":7 Syntax Error in :
    .var 0x0002 = 1;
    syntax error is at or near text '0x0002'.
    Attempting error recovery by ignoring text until the ';'
```

Using Predefined Preprocessor Macros

In addition to macros you define, the `pp` preprocessor provides a set of predefined macros and feature macros that can be used in assembly code. The preprocessor automatically replaces each occurrence of the macro reference found throughout the program with the specified (predefined) value. The DSP development tools also define feature macros that can be used in your code.

NOTE: The `__DATE__`, `__FILE__`, and `__TIME__` macros return strings within the single quotation marks (‘ ’) suitable for initializing character buffers. For more information, see VAR and ASCII String Initialization Support in the Assembler chapter.

The ***Common Predefined Preprocessor Macros*** table describes the common predefined macros provided by the preprocessor. Tables in Using Assembler Feature Macros list feature macros for Blackfin and SHARC processors. These processor-specific feature macros are defined by the project development tools to specify the architecture and language being processed.

Table 3-1: Common Predefined Preprocessor Macros

Macro	*Definition*
`ADI`	Always set to 1
`__LastSuffix__`	Specifies the last value of suffix that was used to build preprocessor generated labels
`__LINE__`	Expands to the line number in the source file that the macro appears on

Table 3-1: Common Predefined Preprocessor Macros (Continued)

Macro	*Definition*
`__FILE__`	Expands to the name and extension of the file in which the macro is defined, for example, `` `macro.asm ``
`__TIME__`	Expands to the current time in 24-hour format `` `hh:mm:ss' ``, for example, `` `06:54:35' ``
`__DATE__`	Expands to the current date in the format `` `mm dd yyyy' ``, for example, `` `Oct 02 2000' ``

Specifying Preprocessor Options

When developing a DSP project, it may be useful to modify the preprocessor's default options. Because the assembler and linker automatically run the preprocessor as your program is built (unless you skip preprocessing entirely), these project development tools can receive input for the preprocessor program and direct its operation. The way the preprocessor options are set depends on the environment used to run the project development software.

You can specify preprocessor options from the preprocessor's command line or within the IDE:

- From the operating system command line, select the preprocessor's command-line switches. For more information on these switches, see Preprocessor Command-Line Switches.
- In the IDE, select the preprocessor's options in the assembler or linker pages of the ***Tools Settings*** dialog box, accessible by selecting a project in any project navigation view and choose ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings***.

  Refer to Specifying Assembler Options in the Assembler chapter.

  For more information, see the CCES online help.

Preprocessor Command Reference

This section provides reference information about the preprocessor's commands and operators used in source code, including their syntax and usage examples. It provides the summary and descriptions of all preprocessor commands and operators.

The preprocessor reads code from a source file (`.asm` or `.ldf`), modifies it according to preprocessor commands, and generates an altered preprocessed source file. The preprocessed source file is an input file for the assembler or linker.

Preprocessor command syntax must conform to these rules:

- Must be the first non-whitespace character on its line
- Cannot be more than one line in length unless the backslash character (`\`) is inserted
- Cannot come from a macro expansion

The preprocessor operators are defined as special operators when used in a `#define` command.

Preprocessor Commands and Operators

The *Preprocessor Command Summary* table lists preprocessor commands. The *Preprocessor Operator Summary* table lists preprocessor operators. The following sections describe each preprocessor command and operator.

Table 3-2: Preprocessor Command Summary

Command/Operator	*Description*
`#define`	Defines a macro. See #define.
`#elif`	Subdivides an `#if #endif` pair. See #elif.
`#else`	Identifies alternative instructions within an `#if #endif` pair. See #else.
`#endif`	Ends an `#if #endif` pair. See #endif.
`#error`	Reports an error message. See #error.
`#if`	Begins an `#if #endif` pair. See #if.
`#ifdef`	Begins an `#ifdef #endif` pair and tests if macro is defined. See #ifdef.
`#ifndef`	Begins an `#ifndef #endif` pair and tests if macro is not defined. See #ifndef.
`#include`	Includes contents of a file. See #include.
`#line`	Sets a line number during preprocessing. See #line.
`#pragma`	Takes any sequence of tokens. See #pragma.
`#undef`	Removes macro definition. See #undef.
`#warning`	Reports a warning message. See #warning.

Table 3-3: Preprocessor Operator Summary

Command/Operator	*Description*
`#`	Converts a macro argument into a string constant. See # (Argument). By default, this operator is disabled. Use the command-line switch -stringize to enable it.
`##`	Concatenates two tokens. See ## (Concatenate).
`?`	Generates unique labels for repeated macro expansions. See ? (Generate a unique label).
...	Specifies a variable-length argument list. See Variable-Length Argument Definitions.

#define

The `#define` command defines macros.

When defining macros in your source code, the preprocessor substitutes each occurrence of the macro with the defined text. Defining this type of macro has the same effect as using the ***Find/Replace*** feature of a text editor, although it does not replace literals in double quotation marks (`" "`) and does not replace a match within a larger token.

For macro definitions longer than one line, place a backslash character (\) at the end of each line (except the last line) for readability; refer to the macro definition rules in Writing Macros.

You can add arguments to the macro definition. The arguments are symbols separated by commas that appear within parentheses.

Syntax:

```
#define macroSymbol replacementText
#define macroSymbol[(arg1,arg2,)] replacementText
```

where:

macroSymbol - macro identifying symbol

replacementText - text to substitute each occurrence of *macroSymbol* in your source code

Examples:

```
#define BUFFER_SIZE 1020
        /* Defines a macro named BUFFER_SIZE and sets its
           value to 1020.
 */

#define copy(src,dest) \
   R0 = DM(src);       \
   DM(dest) = R0
        /* Define a macro named copy with two arguments.
           The definition includes two instructions that copy
           a word from memory to memory.
           For example,
              copy (0x3F,0xC0);
           calls the macro, passing parameters to it.
           The preprocessor replaces the macro with the code:
              R0 = DM(0x3F);
              DM(0xC0) = R0; */
```

#elif

The `#elif` command (else if) is used within an `#if` `#endif` pair. The `#elif` includes an alternative condition to test when the initial `#if` condition evaluates as `FALSE`. The preprocessor tests each `#elif` condition inside the pair and processes instructions that follow the first true `#elif`. There can be an unlimited number of `#elif` commands inside one `#if` `#end` pair.

Syntax:

`#elif` *condition*

where:

condition - expression to evaluate as `TRUE` (nonzero) or `FALSE` (zero)

Example:

```
#if X == 1
   ...
#elif X == 2
   ...
      /* The preprocessor includes text within the section
      and excludes all other text before #else when X=2. */
#else
#endif
```

#else

The `#else` command is used within an `#if` `#endif` pair. It adds an alternative instruction to the `#if` `#endif` pair. Only one `#else` command can be used inside the pair. The preprocessor executes instructions that follow `#else` after all the preceding conditions are evaluated as `FALSE` (zero). If no `#else` text is specified, and all preceding `#if` and `#elif` conditions are `FALSE`, the preprocessor does not include any text inside the `#if` `#endif` pair.

Syntax:

```
#else
```

Example:

```
#if X == 1
   ...
#elif X == 2
   ...
#else
   ...
      /* The preprocessor includes text within the section
         and excludes all other text before #else when
         x!=1 and x!=2. */
#endif
```

#endif

The `#endif` command is required to terminate `#if` `#endif`, `#ifdef` `#endif`, and `#ifndef` `#endif` pairs. Ensure that the number of `#if` commands matches the number of `#endif` commands.

Syntax:

```
#endif
```

Example:

```
#if condition
   ...
   ...
#endif
      /* The preprocessor includes text within the section only
         if the test is true. */
```

#error

The `#error` command causes the preprocessor to raise an error. The preprocessor uses the text following the `#error` command as the error message.

Syntax:

`#error` *`messageText`*

where:

`messageText` - user-defined text

To break a long *`messageText`* without changing its meaning, place a backslash character (`\`) at the end of each line (except the last line).

Example:

```
#ifndef __ADSPBF533__
#error \
         MyError: \
         Expecting a ADSP-BF533.\
         Check the Linker Description File!
#endif
```

#if

The `#if` command begins an `#if` `#endif` pair. Statements inside an `#if` `#endif` pair can include other preprocessor commands and conditional expressions. The preprocessor processes instructions inside the `#if` `#endif` pair only when the *`condition`* that follows the `#if` evaluates as `TRUE`. Every `#if` command must be terminated with an `#endif` command.

Syntax:

`#if` *`condition`*

where:

`condition` - expression to evaluate as `TRUE` (nonzero) or `FALSE` (zero)

Example:

```
#if x!=100  /* test for TRUE condition */

            /* The preprocessor includes text within the section
               if the test is true. */
#endif
```

More Examples:

```
#if (x!=100) && (y==20)

#if defined(__ADSPBF533__)
```

#ifdef

The `#ifdef` (if defined) command begins an `#ifdef` `#endif` pair and instructs the preprocessor to test whether the macro is defined. Each `#ifdef` command must have a matching `#endif` command.

Syntax:

`#ifdef` *`macroSymbol`*

where:

`macroSymbol` - macro identifying symbol

Example:

```
#ifdef __ADSPBF533__
      /* Includes text after #ifdef only when __ADSPBF533__ has
         been defined. */
#endif
```

#ifndef

The `#ifndef` (if not defined) command begins an `#ifndef` `#endif` pair and directs the preprocessor to test for an undefined macro. The preprocessor considers a macro undefined if it has no defined value. Each `#ifndef` command must have a matching `#endif` command.

Syntax:

`#ifndef` *`macroSymbol`*

where:

`macroSymbol` - macro identifying symbol

Example:

```
#ifndef __ADSPBF533__
      /* Includes text after #ifndef only when __ADSPBF533__
         is not defined. */
#endif
```

#include

The `#include` command directs the preprocessor to insert the text from a header file at the command location. There are two types of header files: system and user. However, the `#include` command may be presented in three forms:

- `#include <`*`filename`*`>` - used with system header files
- `#include "`*`filename`*`"` - used with user header files
- `#include` *`text`* - used with a sequence of tokens The sequence of tokens is subject to macro expansion by the preprocessor. After macro expansion, the text must match one of the header file forms.

The only difference to the preprocessor between the two types of header files is the way the preprocessor searches for them.

- System header file *<fileName>* - The preprocessor searches for a system header file in this order: (1) the directories you specify, and (2) the standard list of system directories.
- User header file "*fileName*" - The preprocessor searches for a user header file in this order:
  1. Current directory - the directory where the source file that has the `#include` command(s) lives
  2. Directories you specify
  3. Standard list of system directories

Refer to Header Files and #include Command for more information.

Syntax:

```
#include <fileName>    // include a system header file
#include "fileName"    // include a user header file
#include macroFileNameExpansion
   /* Include a file named through macro expansion.
      This command directs the preprocessor to expand the
      macro. The preprocessor processes the expanded text,
      which must match either <fileName> or "fileName". */
```

Example:

```
#ifdef __ADSPBF533__
         /* Tests that __ADSPBF533__ has been defined. */
#include <stdlib.h>

#endif
```

#line

The `#line` command directs the preprocessor to set the internal line counter to the specified value. Use this command for error tracking purposes.

Syntax:

`#line` *`lineNumber "sourceFile"`*

where:

`lineNumber` - line number of the source line

`sourceFile` - name of the source file included in double quotation marks. The *`sourceFile`* entry can include the drive, directory, and file extension as part of the file name.

Example:

```
#line 7 "myFile.c"
```

NOTE: All assembly programs have `#line` directives after preprocessing. They always have a first line with `#line 1 "filename.asm"` and they will also have `#line` directives to establish correct line numbers for text that came from include files as a result of the processed `#include` directives.

#pragma

The `#pragma` command is the implementation-specific command that modifies the preprocessor behavior. The `#pragma` command can take any sequence of tokens. This command is accepted for compatibility with other CCES tools. The `pp` preprocessor currently does not support any pragmas; therefore, it ignores any information in the `#pragma` command.

Syntax:

`#pragma` *`any_sequence_of_tokens`*

Example:

```
#pragma disable_warning 1024
```

#undef

The `#undef` command directs the preprocessor to undefine a macro.

Syntax:

`#undef` *`macroSymbol`*

where:

`macroSymbol` - macro created with the `#define` command

Example:

```
#undef BUFFER_SIZE   /* undefines a macro named BUFFER_SIZE*/
```

#warning

The `#warning` command causes the preprocessor to issue a warning. The preprocessor uses the text following the `#warning` command as the warning message.

Syntax:

`#warning` *`messageText`*

where:

`messageText` - user-defined text

To break a long *`messageText`* without changing its meaning, place a backslash character (`\`) at the end of each line (except the last line).

Example:

```
#ifndef __ADSPBF533__
#warning \
      MyWarning: \
      Expecting a ADSPBF533. \
      Check the Linker Description File!
#endif
```

(Argument)

The # (argument) "stringization" operator directs the preprocessor to convert a macro argument into a string constant. The preprocessor converts an argument into a string when macro arguments are substituted into the macro definition.

The preprocessor handles white space in string-to-literal conversions by:

- Ignoring leading and trailing white spaces
- Converting white space in the middle of the text to a single space in the resulting string

Syntax:

```
# toString
```

where:

toString - macro formal parameter to convert into a literal string. The # operator must precede a macro parameter. The preprocessor includes a converted string within double quotation marks ("").

NOTE: This feature is off by default. Use the -stringize command-line switch to enable it.

C Code Example:

```
#define WARN_IF(EXP)\
fprintf (stderr,"Warning:"#EXP "/n")
   /* Defines a macro that takes an argument and converts
      the argument to a string. */
WARN_IF(current <minimum);
   /* Invokes the macro passing the condition. */
fprintf (stderr,"Warning:""current <minimum""/n");
   /* Note that the #EXP has been changed to current <minimum
      and is enclosed in " ". */
```

(Concatenate)

The ## (concatenate) operator directs the preprocessor to concatenate two tokens. When you define a macro, you request concatenation with ## in the macro body. The preprocessor concatenates the syntactic tokens on either side of the concatenation operator.

Syntax:

```
token1 ## token2
```

Example:

```
#define varstring(name) .VAR var_##name[] = {`name', 0};
   varstring (error);
   varstring (warning);

/* The above code results in */
   .VAR var_error[] = {`error', 0};
   .VAR var_warning[] = {`warning', 0};
```

? (Generate a unique label)

The "?" operator directs the preprocessor to generate unique labels for iterated macro expansions. Within the definition body of a macro (#define), you can specify one or more identifiers with a trailing question mark (?) to ensure that unique label names are generated for each macro invocation.

The preprocessor affixes "_num" to a label symbol, where num is a uniquely generated number for every macro expansion. For example,

```
abcd? ===> abcd_1
```

If a question mark is a part of the symbol that needs to be preserved, ensure that "?" is delimited from the symbol. For example, "abcd?" is a generated label, and "abcd ?" is not.

Example:

```
#define loop(x,y) mylabel?:x =1+1;/
x = 2+2;/
yourlabel?:y =3*3;/
y = 5*5;/
JUMP mylabel?;/
JUMP yourlabel?;
loop (bz,kjb)
loop (lt,ss)
loop (yc,jl)

             // Generates the following output:
mylabel_1:bz =1+1;bz =2+2;yourlabel_1:kjb =3*3;kjb = 5*5;
JUMP mylabel_1;
JUMP yourlabel_1;
mylabel_2:lt =1+1;lt =2+2;yourlabel_2:ss=3*3;ss =5*5;
JUMP mylabel_2;
JUMP yourlabel_2;
mylabel_3:yc =1+1;yc =2+2;yourlabel_3:Jl=3*3;Jl =5*5;
JUMP mylabel_3;

JUMP yourlabel_3;
```

The last numeric suffix used to generate unique labels is maintained by the preprocessor and is available through a preprocessor predefined macro __LastSuffix__ (see Using Predefined Preprocessor Macros). This value can be used to generate references to labels in the last macro expansion.

The following example assumes the macro "loop" from the previous example.

```
// Some macros for appending a suffix to a label
#define makelab(a, b) a##b
#define Attach(a, b) makelab(a##_, b)
#define LastLabel(foo) Attach( foo, __LastSuffix__)

// jump back to label in the previous expansion
JUMP LastLabel(mylabel);
```

The above expands to (the last macro expansion had a suffix of 3):

```
JUMP mylabel_3;
```

Variable-Length Argument Definitions

A macro can also be defined with a variable-length argument list (by means of the operator).

```
#define test(a, )<definition>
```

For example, the code above defines a macro named test, which takes two or more arguments. It is invoked like any other macro, although the number of arguments can vary.

For example, in the macro definition, the __VA_ARGS__ identifier is available to take on the value of all of the trailing arguments, including the commas, all of which are merged to form a single item. See the ***Sample Variable-Length Argument List*** table.

Table 3-4: Sample Variable-Length Argument List

Sample Argument List	*Description*
test(1)	Error; the macro must have at least one more argument than formal parameters, not counting " ".
test(1,2)	Valid entry
test(1,2,3,4,5)	Valid entry

For example, the code

```
#define test(a, ) bar(a); testbar(__VA_ARGS__);
```

expands into

```
test (1,2) -> bar(1); testbar(2);
test (1,2,3,4,5) -> bar(1); testbar(2,3,4,5);
```

Preprocessor Command-Line Reference

The pp preprocessor is invoked implicitly by the assembler and linker when processing assembly sources and linker description files. The preprocessor can also be invoked directly from the command line.

This section contains:

Running the Preprocessor

To run the preprocessor from the command line, type the name of the program followed by arguments in any order.

```
pp [-switch1 [-switch2 ]] [sourceFile]
```

The ***Preprocessor Command Line Argument Summary*** summarizes these arguments.

Table 3-5: Preprocessor Command Line Argument Summary

Argument	*Description*
`pp`	Name of the preprocessor program
`-switch`	Switch (or switches) to process. The preprocessor offers several switches that are used to select its operation and modes. Some preprocessor switches take a file name as a required parameter.
`sourceFile`	Name of the source file to process. The preprocessor supports relative path names and absolute path names. The `pp.exe` outputs a list of command-line switches when run without this argument.

For example, the following command line:

```
pp -Dfilter_taps=100 -v -o bin/p1.is p1.asm
```

runs the preprocessor with:

- `-Dfilter_taps=100` - defines the macro `filter_taps` as equal to 100
- `-v` - displays verbose information for each phase of the preprocessing
- `-o bin\p1.is` - specifies the name and directory for the intermediate preprocessed file
- `p1.asm` - specifies the assembly source file to preprocess

NOTE: Most switches without arguments can be negated by prefixing `-no` to the switch. For example, `-nowarn` turns off warning messages, and `-nocs!` turns off omitting "`!`" style comments.

Preprocessor Command-Line Switches

The preprocessor is normally controlled through the switches (or CCES options) of other development tools namely the assembler and linker, although it can also operate independently from the command line with its own command-line switches.

The ***Preprocessor Command-Line Switch Summary*** table lists `pp.exe` switches. A detailed description of each switch follows the table.

Table 3-6: Preprocessor Command-Line Switch Summary

Switch Name	*Description*
`-cpredef`	Enables the "stringization" operator and provides C compiler-style preprocessor behavior. See -cpredef.
`-cs!`	Treats as a comment all text after " `!`" on a single line. See -cs!.
`-cs/*`	Treats as a comment all text within `/* *`. See -cs/*.
`-cs//`	Treats as a comment all text after `//`. See -cs//.
`-cs{`	Treats as a comment all text within `{ }`. See -cs{.
`-csall`	Accepts comments in all formats. See -csall.
`-D`*`macro`*`[=`*`definition`*`]`	Defines *`macro`*. See -Dmacro[=def].
`-dependency-add-target`	Adds the name of the target to the target list. See -dependency-add-target.
`-expand-symbolic-links`	Enables support for Cygwin style paths. See -expand-symbolic-links.
`-expand-windows-shortcuts`	Enables support for Windows shortcuts. See -expand-windows-shortcuts.
`-gnu-style-dependencies`	Produces dependency information compatible with the GNU Make tool. See -gnu-style-dependencies.
`-h[elp]`	Outputs a list of command-line switches. See -h[elp].
`-i`	Outputs only makefile dependencies for `include` files specified in double quotes. See -i.
`-i|I`*`directory`*	Searches *`directory`* for included files. See -Idirectory.
`-I-`	Indicates where to start searching for system include files, which are delimited by < >. See -I-.
`-M`	Makes dependencies only. See -M.
`-MM`	Makes dependencies and produces preprocessor output. See -MM.
`-Mo` *`filename`*	Specifies *`filename`* for the make dependencies output file. See -Mo filename.
`-Mt` *`filename`*	Makes dependencies for the specified source file. See -Mt filename.
`-o` *`filename`*	Writes output to named file. See -o filename.
`-stringize`	Enables stringization (includes a string in double quotes). See -stringize.
`-tokenize-dot`	Treats " `.`" (dot) as an operator when parsing identifiers. See -tokenize-dot.
`-Uname`	Undefines a macro on the command line. See -Uname.
`-v[erbose]`	Displays information about each preprocessing phase. See -v[erbose].
`-version`	Displays version information for the preprocessor. See -version.
`-w`	Suppresses all preprocessor-generated warnings. See -w.
`-Wnumber`	Suppresses any report of the specified warning. See -Wnumber.
`-warn`	Prints warning messages (default). See -warn.
`-Wwarn-error`	Raises all preprocessor-generated warnings to errors. See -Wwarn-error.

The following sections describe preprocessor command-line switches.

-cpredef

The `-cpredef` switch directs the preprocessor to produce C compiler-style strings in all cases. By default, the preprocessor produces assembler-style strings within single quotes (for example, `` `string' ``) unless the `-cpredef` switch is used.

The `-cpredef` switch sets the following C compiler-style behaviors:

- Directs the preprocessor to use double quotation marks rather than the default single quotes as string delimiters for any preprocessor- generated strings. The preprocessor generates strings for predefined macros that are expressed as string constants, and as a result of the stringize operator in macro definitions (see the ***Common Predefined Preprocessor Macros*** table in Using Predefined Preprocessor Macros).
- Enables the stringize operator (`#`) in macro definitions. By default, the stringize operator is disabled to avoid conflicts with constant definitions (see -stringize).
- Parses identifiers using C language rules instead of assembler rules. In C, the character "`.`" is an operator and is not considered part of an identifier. In the assembler, the "`.`" is considered part of a directive or label. With `-cpredef`, the preprocessor treats "`.`" as an operator.

The following example shows the difference in effect of the two styles.

```
#define end last
// what label.end looks like with -cpredef
label.last     // "end" parsed as ident and macro expanded

// what label.end looks like without -cpredef (asm rules)
label.end      // "end" not parsed separately
```

-cs!

The `-cs!` switch directs the preprocessor to treat as a comment all text after "`!`" on a single line.

-cs/*

The `-cs/*` switch directs the preprocessor to treat as a comment all text within `/* */` on multiple lines. This is enabled by default.

-cs//

The `-cs//` switch directs the preprocessor to treat as a comment all text after `//` on a single line. This is enabled by default.

-cs{

The `-cs{` switch directs the preprocessor to treat as a comment all text within `{ }` on multiple lines.

-csall

The `-csall` switch directs the preprocessor to accept comments in all formats.

-Dmacro[=def]

The -Dmacro switch directs the preprocessor to define a macro. If you do not include the optional definition string (= def), the preprocessor defines the macro as value 1. Similarly to the C compiler, you can use the -D switch to define an assembly language constant macro.

Examples:

```
-Dinput                 // defines input as 1
-Dsamples=10            // defines samples as 10
-Dpoint="Start"         // defines point as "Start"
-D_LANGUAGE_ASM=1       // defines _LANGUAGE_ASM as 1
```

-dependency-add-target

The -dependency-add-target switch directs the preprocessor to add the file name that follows the switch to the list of targets when generating a list of dependencies. The switch is used when dependencies are being generated, either by invoking the -M or -MM switch.

-expand-symbolic-links

The expand-symbolic-links switch directs the preprocessor to correctly access directories and files whose name or path contain Cygwin path components.

-expand-windows-shortcuts

The expand-windows-shortcuts switch directs the preprocessor to correctly access directories and files whose name or path contain Windows shortcuts.

-gnu-style-dependencies

The -gnu-style-dependencies switch directs the preprocessor to produce dependency information that is compatible with the GNU Make tool. This switch is used when dependencies are being generated, either by invoking the -M or -MM switch. This switch ensures that all backslashes in dependency paths are converted to forward slashes, that the names of target and dependencies have their case preserved, whitespace in file names is escaped using backslashes, and all path names are normalized to be relative to the current working directory of the tool.

-h[elp]

The -h (or -help) switch directs the preprocessor to send to standard output the list of command-line switches with a syntax summary.

-i

The -i (less includes) switch may be used with the -M or -MM switches to direct the preprocessor to *not* output dependencies on any system files. System files are any files that are brought in using #include < >. Files included using #include " " (double quote characters) are included in the dependency list.

-Idirectory

The `-I`*`directory`* switch directs the preprocessor to append the specified directory (or a list of directories separated by semicolons) to the search path for included header files (see #include).

NOTE: No space is allowed between `-I` and the path name.

The preprocessor searches for included files delimited by double quotation marks (`" "`) in this order:

1. The source directory (that is, the directory in which the original source file resides)
2. The directories in the search path supplied by the `-I` switch. If more than one directory is supplied by the `-I` switch, they are searched in the order that they appear on the command line.
3. The system directory (that is, the `/include` subdirectory of the installation directory)

   NOTE: The *current directory* is the directory where the source file lives, not the directory of the assembler program. Usage of full path names for the `-I` switch on the command line (omitting the disk partition) is recommended.

The preprocessor searches for included files delimited by `< >` in this order:

1. The directories in the search path supplied by the -I- switch (subject to modification by the switch. If more than one directory is supplied by the `-I` switch, the directories are searched in the order that they appear on the command line.
2. The system directory (that is, the `<`*`processor_family`*`>/include` subdirectory of the installation directory.

-I-

The `-I-` switch indicates where to start searching for system include files, which are delimited by `< >`. If there are several directories in the search path, the `-I-` switch indicates where in the path the search for system include files begins.

For example,

```
pp -Idir1 -Idir2 -I- -Idir3 -Idir4 myfile.asm
```

When searching for `#include "inc1.h"` the preprocessor searches in the source directory, then `dir1`, `dir2`, `dir3`, and `dir4` in that order.

When searching for `#include <inc2.h>` the preprocessor searches for the file in `dir3` and then `dir4`. The `-I-` switch marks the point where the system search path starts.

-M

The `-M` switch directs the preprocessor to output a rule (generate make rule only) suitable for the make utility, describing the dependencies of the source file. The output, a make dependencies list, is written to `stdout` in the standard command-line format.

`"target_file": "dependency_file.ext"`

where:

dependency_file.ext may be an assembly source file or a header file included with the `#include` preprocessor command

When `-o filename` is used with `-M`, the `-o` switch is ignored. To specify an alternate target name for the make dependencies, use the -Mt filename switch. To direct the make dependencies to a file, use the -Mo filename switch.

-MM

The `-MM` switch directs the preprocessor to output a rule (generate make rule and preprocess) suitable for the make utility, describing the dependencies of the source file. The output, a make dependencies list, is written to `stdout` in the standard command-line format.

The only difference between `-MM` and `-M` actions is that the preprocessing continues with `-MM`. See -M for more information.

-Mo filename

The `-Mo` switch specifies the name of the make dependencies file (output make rule) that the preprocessor generates when using the `-M` or `-MM` switch. The switch overrides default of make dependencies to `stdout`.

-Mt filename

The `-Mt` switch specifies the name of the target file (output make rule for the named source) for which the preprocessor generates the make rule using the `-M` or `-MM` switch. The `-Mt` *`filename`* switch overrides the default *filename*`.is` file. See -M for more information.

-o filename

The `-o` switch directs the preprocessor to use the specified *`filename`* argument for the preprocessed output file. The preprocessor directs the output to `stdout` when no `-o` switch is specified.

-stringize

The `-stringize` switch enables the preprocessor stringization operator. By default, this switch is off to avoid possible undesired stringization.

For example, there is a conflict between the stringization operator and the assembler's boolean constant format in the following macro definition:

```
#define bool_const b#00000001
```

-tokenize-dot

The `-tokenize-dot` switch parses identifiers using C language rules instead of assembler rules, without the need of other C semantics (see -cpredef for more information).

When the -tokenize-dot switch is used, the preprocessor treats "." as an operator and not as part of an identifier. If the -notokenize-dot switch is used, it returns the preprocessor to the default behavior.

-Uname

The -Uname switch directs the preprocessor to undefine a macro on the command line. The "undefine macro" switch applies only to macros defined on the same command line. The functionality provides a way for users to undefine feature macros specified by the assembler or linker.

-v[erbose]

The -v[erbose] switch directs the preprocessor to output the version of the preprocessor program and information for each phase of the preprocessing.

-version

The -version switch directs the preprocessor to display version information for the preprocessor program.

NOTE: The -version switch on the assembler command line provides version information for both the assembler and preprocessor. The version switch on the preprocessor command line provides preprocessor version information only.

-w

The -w (disable all warnings) switch directs the assembler not to display warning messages generated during assembly. Note that -w has the same effect as the -nowarn switch.

-Wnumber

The -W*number* (warning suppression) switch selectively disables warnings specified by one or more message numbers. For example, -W74 disables warning message pp0074.

-warn

The -warn switch generates (prints) warning messages (this switch is on by default). The -nowarn switch negates this action.

-Wwarn-error

The -Wwarn-error switch turns all preprocessor warnings into errors.

Index

Symbols

A

Symbols

A

B

C

Symbols

C

Symbols

C

Symbols

D

Symbols

D

E

Symbols

E

Symbols

F

Symbols

F

Symbols

G

Symbols

P

Q

R

Symbols

S

Symbols

S

Symbols

S

Symbols

S

Symbols

Symbols

Z