VISUALDSP++™ 3.1
Assembler and Preprocessor
Manual for Blackfin Processors

Revision 2.1, April 2003

Part Number
82-000410-04

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

Copyright Information

Disclaimer

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

Trademark and Service Mark Notice

The Analog Devices logo, VisualDSP, the VisualDSP logo, SHARC, the SHARC logo, TigerSHARC, the TigerSHARC logo, Blackfin, the Blackfin logo are registered trademarks of Analog Devices, Inc.

VisualDSP++, the VisualDSP++ logo, CROSSCORE, the CROSSCORE logo, and EZ-KIT Lite are trademarks of Analog Devices, Inc.

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

CONTENTS

PREFACE

ASSEMBLER

PREPROCESSOR

INDEX

PREFACE

Thank you for purchasing Analog Devices development software for Blackfin® embedded media processors.

Purpose

The *VisualDSP++ 3.1 Assembler and Preprocessor Manual for Blackfin Processors*contains information about the assembler program for Blackfin embedded processors that support a Media Instruction Set Computing (MISC) architecture. This architecture is the natural merging of RISC, media functions, and signal processing characteristics that delivers signal processing performance in a microprocessor-like environment.

The manual provides information on how to write assembly programs for Blackfin embedded processors and 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 programmers who are familiar with Analog Devices Blackfin processors. This manual assumes that the audience has a working knowledge of the appropriate Blackfin processor architecture and instruction set. Programmers who are unfamiliar with

Analog Devices processors can use this manual but should supplement it with other texts (such as Hardware Reference and Instruction Set Reference manuals that describe your target architecture).

Manual Contents

The manual consists of:

- Chapter 1, "Assembler"

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

- Chapter 2, "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 edition of the *VisualDSP++ 3.1 Assembler and Preprocessor Manual for Blackfin Processors* supports all current Blackfin processors (see "Supported Processors".

VisualDSP++ 3.1 for Blackfin processors is an upgrade of VisualDSP++ 3.0 development environmant that introduces several new features, such as:

- Updated feature macros that support all Blackfin processors
- Support for 1.31 fract via `.BYTE4/R32` and `.VAR/R32`
- Support for a new `BITPOS()` operator
- Preprocessor's stringization operator

Technical or Customer Support

You can reach DSP Tools Support in the following ways:

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

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

Supported Processors

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

ADSP-BF531	ADSP-BF532 (formerly ADSP-21532)
ADSP-BF533	ADSP-BF535 (formerly ADSP-21535)
ADSP-DM102	AD6532

Product Information

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

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

MyAnalog.com

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

Registration:

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

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

DSP Product Information

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

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

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

Related Documents

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

VisualDSP++ 3.1 Getting Started Guide for Blackfin Processors

VisualDSP++ 3.1 User's Guide for Blackfin Processors

VisualDSP++ 3.1 C/C++ Compiler and Library Manual for Blackfin Processors

VisualDSP++ 3.1 C/C++ Assembler and Preprocessor Manual for Blackfin Processors

VisualDSP++ 3.1 Linker and Utilities Manual for Blackfin Processors

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

Online Technical Documentation

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

A description of each documentation file type is as follows.

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

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

Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or Analog Devices website.

From VisualDSP++

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

From Windows

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

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

Using Windows Explorer

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

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **VisualDSP**, and **VisualDSP++ Documentation**.
- Access the `.PDF` files by clicking the **Start** button and choosing **Programs**, **VisualDSP**, **Documentation for Printing**, and the name of the book.

From the Web

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

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

Printed Manuals

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

VisualDSP++ Documentation Set

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

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

Hardware Manuals

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

Data Sheets

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

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

Contacting DSP Publications

Please send your comments and recommendation on how to improve our manuals and online Help. You can contact us by:

- Emailing `dsp.techpubs@analog.com`
- Filling in and returning the attached Reader's Comments Card found in our manuals

Notation Conventions

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

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

Example	**Description**
Close command (**File** menu)	Text in **bold** style indicates the location of an item within the VisualDSP++ environment's menu system. For example, the **Close** command appears on the **File** menu.
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.

Notation Conventions

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 italic style format.
ⓘ	A note, providing information of special interest or identifying a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
■	A caution, providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word **Caution** appears instead of this symbol.

1 ASSEMBLER

This chapter provides information on how to use the assembler for developing and assembling programs for Blackfin® embedded processors.

The chapter contains:

- "Assembler Guide" on page 1-2
  Describes the process of developing new programs in the Blackfin processors assembly language.
- "Assembler Syntax Reference" on page 1-19
  Provides the assembler rules and conventions of syntax which is used to define symbols (identifiers), expressions, and to describe different numeric and comment formats.
- "Assembler Command-Line Reference" on page 1-76
  Provides reference information on the assembler's switches and conventions.

Assembler Guide

The `easmblkfn.exe` assembler driver runs from the VisualDSP++ Integrated Debugging and Development Environment (IDDE) or from an operating system command line. The assembler processes assembly source, data, header files, and produces an object file. Assembler operations depend on two types of controls: assembler directives and assembler switches.

This section describes the process of developing new programs in the Blackfin 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:

- "Writing Assembly Programs" on page 1-3
- "Using Assembler Support for C Structs" on page 1-12
- "Preprocessing a Program" on page 1-14
- "Using Assembler Feature Macros" on page 1-15
- "Reading a Listing File" on page 1-17
- "Make Dependencies" on page 1-16
- "Specifying Assembler Options in VisualDSP++" on page 1-89

For information about the processor architecture, including the processor's instruction set used when writing the assembly programs, see the *Hardware Reference Manual* and *Instruction Set Manual* for an appropriate processor.

Assembler Overview

The assembler processes data from assembly source (`.ASM`), data (`.DAT`), and include header (`.H`) files to generate object files in Executable and Linkable Format (ELF), an industry-standard format for binary object files. The object file name has a `.DOJ` extension.

In addition to the object file, the assembler can produce a listing file, which shows the correspondence between the binary code and the source.

Assembler switches are specified from the VisualDSP++ IDDE or in the command 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.
See "Assembler Command-Line Reference" on page 1-76.

You can also set assembler options via the **Assemble** tab of the VisualDSP++ **Project Options** dialog box (see "Specifying Assembler Options in VisualDSP++" on page 1-89).

Writing Assembly Programs

Assembler directives are coded in your assembly source file. The directives allow you to define variables, set up some hardware features, and identify program's 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 VisualDSP++ 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.

Figure 1-1 shows a graphical overview of the assembly process. It shows the preprocessor processing the assembly source (`.ASM`) and header (`.H`) files.

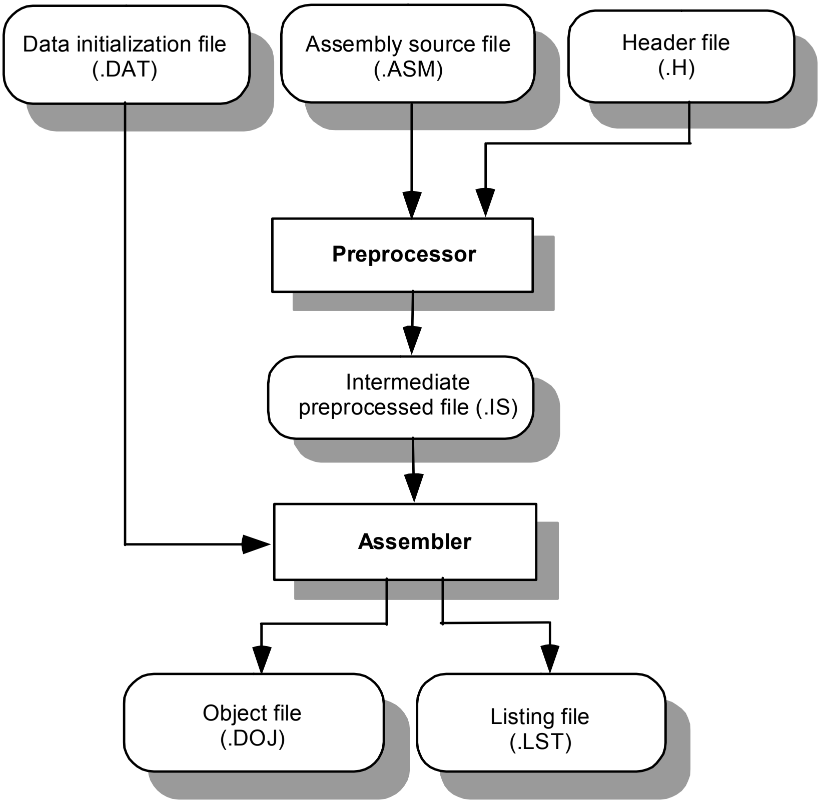


Figure 1-1. Assembler Input and Output Files

Assemble your source files from the VisualDSP++ environment or using any mechanism, such as a batch file or makefile, that will support invoking the assembler driver `easmblkfn.exe` with a specified command-line command. By default, the assembler processes an input file to produce a binary object file (`.DOJ`) and an optional listing file (`.LST`).

Object files produced by the Blackfin 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 on the linker and archiver, see the *VisualDSP++ 3.1 Linker and Utilities Manual for Blackfin DSPs*.

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

The assembler listing files are text files read for information on the results of the assembly process. The listing file also provides information about the imported C data structures. It tells which imports were used within the program, followed by a more detailed section (see `.IMPORT` directive on page 1-50). It shows the name, total size and layout with offset for the members. The information appears at the end of the listing. You must request a listing file (see the `-l listname` switch on page 1-85) to get the information.

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 will be 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 Blackfin processor instruction set syntax documented in the corresponding Instruction Set manual. Each instruction line must be terminated by a semicolon (;). Figure 1-2 on page 1-9 shows an example 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 then refer to this memory location using the label instead of an absolute 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,

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

Assembler Directives

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

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

For example,

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

For a complete description of the Blackfin assembler’s directive set, see “Assembler Directives” on page 1-37.

Preprocessor Commands

Preprocessor commands begin with a pound sign (#) and end with a carriage return. 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 onto the next line.

Do not put any characters between the backslash and the carriage return. 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" on page 2-3. For a list of the preprocessor commands, see "Preprocessor Command Reference" on page 2-10.

Program Structure

An assembly source file defines code (instructions) and data, and organizes the instructions and data to allow 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 will occupy contiguous memory addresses in the processor. The name given in a section directive corresponds to an input section name in the Linker Description File.

Suggested input section names that you could use in your assembly source appear in Table 1-1. Using these predefined names in your sources makes it easier to take advantage of the default `.LDF` file included in your DSP system. For more information on LDFs, see the *VisualDSP ++ 3.1 Linker and Utilities Manual for Blackfin Processors*.

You can use sections in a program to group elements to meet hardware constraints. For example, the ADSP-BF535 processor has a separate program and data memory in the Level 1 memory only. Level 2 memory and external memory are not separated into instruction and data memory.

Table 1-1. Suggested Input Section Names

`.SECTION` **Name**	**Description**
`data1`	A section that holds data.
`program`	A section that holds code.

To group the code that reside in off-chip memory, declare a section for that code and place that section in the selected memory with the linker. Figure 1-2 on page 1-9 shows how a program divides into sections that match the memory segmentation of Blackfin processors.

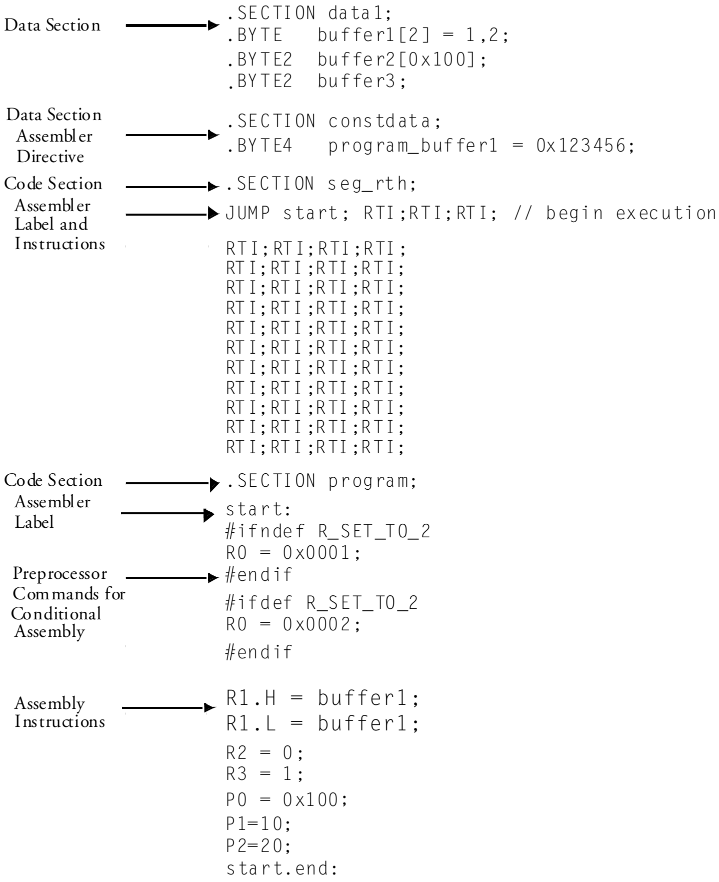
The example assembly program defines four sections; each section begins with a `.SECTION` directive and ends with the occurrence of the next `.SECTION` directive or end-of-file. The source program contains two data and two program sections:

- Data Sections—`data1` and `constdata`. Variables and buffers are declared and can be initialized.
- Program Sections—`seg_rth` and `program`. Data, instructions, and statements for conditional assembly are coded.

Looking at Figure 1-2, notice that an assembly source may contain preprocessor commands, such as `#include` to include other files in your source code, `#ifdef` for conditional assembly, or `#define` to define macros.

Assembler directives, such as `.BYTE`, appear within sections to declare and initialize variables.

Listing 1-1 on page 1-10 shows a sample user-defined Linker Description File. Looking at the LDF's `SECTIONS{}` command, notice that the `INPUT_SECTION` commands map to sections `program`, `data1`, `constdata`, `ctor`, and `seg_rth`. The LDF's `SECTIONS{}` command defines the `.SECTION` placements in the system's physical memory as defined by the linker's `Memory{}` command.


Data Section
Data Section Assembler Directive
Code Section
Assembler Label and Instructions
Code Section
Assembler Label
Preprocessor Commands for Conditional Assembly
Assembly Instructions

```
.SECTION data1;
.BYTE   buffer1[2] = 1,2;
.BYTE2  buffer2[0x100];
.BYTE2  buffer3;

.SECTION constdata;
.BYTE4   program_buffer1 = 0x123456;
.SECTION seg_rth;
JUMP start; RTI;RTI;RTI; // begin execution
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;
.SECTION program;
start:
#ifndef R_SET_TO_2
R0 = 0x0001;
#endif
#ifdef R_SET_TO_2
R0 = 0x0002;
#endif

R1.H = buffer1;
R1.L = buffer1;
R2 = 0;
R3 = 1;
P0 = 0x100;
P1=10;
P2=20;
start.end:
```

Figure 1-2. Assembly Source File Structure

Listing 1-1. Example Linker Description File

```
ARCHITECTURE(ADSP-BF535)
SEARCH_DIR($ADI_DSP\Blackfin\lib)
$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY      /* Define/label system memory
{           /* List of global Memory Segments */
MEM_PROGRAM { TYPE(RAM) START(0xF0000000) END(0xF002FFFF) WIDTH(8) }
MEM_HEAP    { TYPE(RAM) START(0xF0030000) END(0xF0037FFF) WIDTH(8) }
MEM_STACK   { TYPE(RAM) START(0xF0038000) END(0xF003DFFF) WIDTH(8) }
MEM_SYSSTACK { TYPE(RAM) START(0xF003E000) END(0xF003FDFF) WIDTH(8) }
MEM_ARGV    { TYPE(RAM) START(0xF003FE00) END(0xF003FFFF) WIDTH(8) }
}

PROCESSOR p0 /* The processor in the system */
{
   OUTPUT($COMMAND_LINE_OUTPUT_FILE)

SECTIONS
{                /* List of sections for processor P0 */
   program
   {            // Align all code sections on 2 byte boundary
       INPUT_SECTION_ALIGN(2)
           INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
       INPUT_SECTION_ALIGN(1)
           INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
       INPUT_SECTION_ALIGN(1)
           INPUT_SECTIONS( $OBJECTS(constdata) $LIBRARIES(constdata))
       INPUT_SECTION_ALIGN(1)
           INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor))
       INPUT_SECTION_ALIGN(2)
           INPUT_SECTIONS( $OBJECTS(seg_rth))
       } >MEM_PROGRAM


    stack
      {
        ldf_stack_space = .;
        ldf_stack_end = ldf_stack_space +
                  MEMORY_SIZEOF(MEM_STACK) - 4;
      } >MEM_STACK
    sysstack
      {
        ldf_sysstack_space = .;
        ldf_sysstack_end = ldf_sysstack_space +
                  MEMORY_SIZEOF(MEM_SYSSTACK) - 4;
      } >MEM_SYSSTACK

    heap
      {         // Allocate a heap for the application
        ldf_heap_space = .;
         ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(MEM_HEAP) - 1;
       ldf_heap_length = ldf_heap_end - ldf_heap_space;
      } >MEM_HEAP

    argv
      {        // Allocate argv space for the application
        ldf_argv_space = .;
        ldf_argv_end = ldf_argv_space + MEMORY_SIZEOF(MEM_ARGV) - 1;
       ldf_argv_length = ldf_argv_end - ldf_argv_space;
      } >MEM_ARGV
  }
}
```

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 *VisualDSP ++ 3.1 Linker and Utilities Manual for Blackfin Processors*.

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 DSP. 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 Blackfin's C/C++ compiler is provided in the *VisualDSP++ 3.1 C/C++ Compiler and Library Manual for Blackfin Processors*, 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 are the assembler data directives and built-ins that provide high-level programming features with C structs in the assembler:

- **Data Directives:**

`.IMPORT`	(see on page 1-50)
`.EXTERN STRUCT`	(see on page 1-46)
`.STRUCT`	(see on page 1-66)

- **C Struct in Assembly Built-ins:**

`offsetof(struct/typedef,field)`	(see on page 1-33)
`sizeof(struct/typedef)`	(see on page 1-33)

- **Struct References:**

`struct->field` (nesting supported)	(see on page 1-34)

For more information on C struct support, refer to the "-flags-compiler" command-line switch on page 1-81 and to "Reading a Listing File" on page 1-17.

C structs in assembly features accept the full set of legal C symbol names, including those that are otherwise reserved in Blackfin processor assembler. For example, 'X' and 'Z' are reserved keywords in the Blackfin processor assembler, but it is legal to reference them in the context of the C struct in assembly features.

The example below shows how to access the parts of the struct defined in the header file, but it is not a complete program on its own. Refer to your DSP project files for complete code examples.

```
.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);
```

Preprocessing a Program

The assembler includes 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. Table 2-3 on page 2-11 lists preprocessor commands and provides a brief description of each command.

You can see the command line the assembler uses to invoke the preprocessor by adding the "-v[erbose]" switch (on page 1-88) to the assembler command line or by selecting **Verbose output** on the **Assemble** tab (property page) of the **Project Options** dialog box, accessible from the **Project** menu (see "Specifying Assembler Options in VisualDSP++" on page 1-89).

Preprocessor commands are useful for modifying assembly code. For example, you can use the `#include` command to fill memory, load configuration registers, and 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 macro `MAXIMUM` in the example on page 1-7 is replaced with the number `100` during preprocessing. For more information on the preprocessor command set, see "Preprocessor Command Reference" on page 2-10. For more information on preprocessor usage, see "-flags-pp -opt1 [,-opt2...]" on page 1-83.

Note that 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 will not match a preprocessor macro "`extern`".

This behavior can affect how macro expansion is done for some Blackfin 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 includes the command to invoke preprocessor macros to define the context, such as the source language, the architecture, and the specific processor. These "feature macros" allow the programmer to use preprocessor conditional commands to configure the source for assembly based on the context.

The set of feature macros include:

`-D_LANGUAGE_ASM=1`	Always present
`-D__ADSPBLACKFIN__ =1`	Always present
`-D__ADSPBF531__=1`	Present when running `easmblkfn -proc ADSP-BF531` with ADSP-BF531 processor.

`-D__ADSPBF532__=1`
`-D__ADSP21532__=1`
	Present when running `easmblkfn -proc ADSP-BF532` with ADSP-BF532 processor.

`-D__ADSPBF533__=1`
`-D__ADSP21533__=1`
	Present when running `easmblkfn -proc ADSP-BF533` with ADSP-BF533 processor.

`-D__ADSPBF535__=1`
`-D__ADSP21535__=1`
	Present when running `easmblkfn -proc ADSP-BF535` with ADSP-BF535 processor.

`-D__ADSPDM102__=1`	Present when running `easmblkfn -proc ADSP-DM102` with ADSP-DM102 processor.
`-D__AD6532__=1`	Present when running `easmblkfn -proc AD6532` with AD6532 processor.

For the `.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 Blackfin processors,

`easmblkfn -ADSP-BF535` assembly -> `ccblkfn -BF535`

`easmblkfn -ADSP-BF532` assembly -> `ccblkfn -BF532`

Use the `-verbose` option to verify what macro is default-defined. Refer to Chapter 1 in the *VisualDSP++ 3.1 C/C++ Compiler and Library Manual for Blackfin Processors* for more information.

Make Dependencies

The assembler can generate "make dependencies" for a file to allow VisualDSP++ and other makefile-based build environments to determine when to rebuild an object file due to changes in the input files. The assembler source file and any files mentioned in `#include` commands, `.IMPORT` directives, or buffer initializations (in `.VAR`, `.STRUCT`, and `.BYTEn` directives) constitute the "make dependencies" for an object file.

When VisualDSP++ requests make dependencies for the assembly, the assembler produces the dependencies from buffer initializations and invokes

- The preprocessor to determine the make dependency from `#include` commands, and
- The compiler to determine the make dependencies from the `.IMPORT` headers.

The following example shows make dependencies for `VCSE_IBase.h` which includes `vcse.h`. Note that the same header `VCSE_IBase.h` when called from the assembler (with assembler `#defines`) also includes `VCSE_asm.h`, but this was not the case when called for compiling `.IMPORT`.

```
easmblkfn -proc ADSP-BF532 -M -l main.lst main.asm

// dependency from the assembler
"main.doj": "main.asm"

// dependencies from the assembler preprocessor pp for the
// #include headers
"main.doj": "ACME_Impulse_factory.h"
"main.doj": "ACME_Impulse_types.h"
"main.doj": "VCSE_IBase.h"
"main.doj": "VCSE_asm.h"
"main.doj": "vcse.h"

// dependencies from the compiler for the .IMPORT headers
main.doj: .\ACME_IFir.h
main.doj: .\ADI_IAlg.h
main.doj: .\VCSE_IBase.h
main.doj: .\vcse.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 listname.lst` option (see on page 1-84) to get a listing file.

Assembler Syntax Reference

When you develop 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 conventions of syntax to define symbols (identifiers), expressions, and use different numeric and comment formats.

Software developers who write assembly programs should be familiar with:

- "Assembler Keywords and Symbols" on page 1-20
- "Assembler Expressions" on page 1-24
- "Assembler Operators" on page 1-25
- "Numeric Formats" on page 1-27
- "Comment Conventions" on page 1-30
- "Conditional Assembly Directives" on page 1-30
- "C Struct Support in Assembly Built-In Functions" on page 1-33
- "-> Struct References" on page 1-34
- "Assembler Directives" on page 1-37

Assembler Keywords and Symbols

The assembler supports predefined keywords that include register and bitfield names, assembly instructions, and assembler directives. Table 1-2 lists the assembler keywords for Blackfin processors. Although the keywords in the listing appear in uppercase, the keywords are case insensitive in the assembler's syntax. For example, the assembler does not differentiate between "`DATA`" and "`data`".

Table 1-2. Blackfin Processor Assembler Keywords

.ALIGN	.ASCII	.ASM_ASSERT	.ASSERT	.BYTE
.BYTE2	.BYTE4	.ELSE	.ELIF	.ENDIF
.EXTERN	.FILE	.GLOBAL	.IF	.IMPORT
.LEFTMARGIN	.LIST	.LIST_DATA	.LIST_DATFILE	.LIST_DEFTAB
.LIST_LOCTAB	.LIST_WRAPDATA	.NEWPAGE	.NOLIST	.NOLIST_DATA
.NOLIST_DATFILE	.NOLIST_WRAPDATA	.ORG	.PAGELENGTH	.PAGEWIDTH
.PREVIOUS	.SECTION	.STRUCT	.TYPE	.VAR
.WEAK				
A0	A1	ABORT	ABS	AC
ALIGN8	ALIGN16	ALIGN24	AMNOP	AN
AND	ASHIFT	ASL	ASR	ASSIGN
ASTAT	AV0	AV1	AZ	
B	B0	B1	B2	B3
BANG	BAR	BITCLR	BITMUX	BITPOS
BITSET	BITTGL	BITTST	BIT_XOR_AC	BP
BREV	BRF	BRT	BY	BYTEOP1P
BYTEOP16M	BYTEOP1NS	BYTEOP16P	BYTEOP2M	BYTEOP2P
BYTEOP3P	BYTEPACK	BYTEUNPACK	BXOR	BXORSHIFT

Table 1-2. Blackfin Processor Assembler Keywords (Cont'd)

CALL	CARET	CC	CLI	CLIP
CO	CODE	COLON	COMMA	CSYNC
DATA	DBG	DBGA	DBGAH	DBGAL
DBGCMPLX	DBGHALT	DEPOSIT	DISALGNEXCPT	DIVSDEPOSIT
DOZE	DIVQ	DIVS	DOT	EMUCAUSE
EMUEXCPT	EXCAUSE	EXCPT	EXPADJ	EXTRACT
FEXT	FEXTSX	FLUSH	FLUSHINV	FP
FU	GE	GF	GT	
H	HI	HLT	HWERRCAUSE	
I0	I1	I2	I3	IDLE
IDLE_REQ	IFLUSH	IH	INTRP	IS
ISS2	IU	JUMP	JUMP.L	JUMP.S
L	LB0	LB1	LC0	LC1
LE	LENGTH	LINK	LJUMP	LMAX
LMIN	LO	LOOP	LOOP_BEGIN	LOOP_END
LPAREN	LSETUP	LSHIFT	LT	LT0
LT1	LZ			
M	M0	M1	M2	M3
MAX	MIN	MINUS	MNOP	MUNOP
NEG	NOP	NOT	NS	
ONES	OR	OUTC		
P0	P1	P2	P3	P4
P5	PACK	PC	PRNT	PERCENT
PLUS	PREFETCH			
R	R0	R1	R2	R3

Table 1-2. Blackfin Processor Assembler Keywords (Cont'd)

R32	R4	R5	R6	R7
RAISE	RBRACE	RBRACK	RETI	RETN
RETS	RETX	RND	RND12	RND20
RNDH	RNDL	ROL	ROR	ROT
ROT_L_AC	ROT_R_AC	RPAREN	RSDL	RTE
RTI	RTN	RTS	RTX	R1_COLON0
S	S2RND	SAA	SAA1H	SAA1L
SAA2H	SAA2L	SAA3H	SAA3L	SAT
SCO	SEARCH	SHT_TYPE	SIGN	SIGNBITS
SLASH	SLEEP	SKPF	SKPT	SP
SS	SSF	SSF_RND_HI	SSF_TRUNC	SSF_TRUNC_HI
SSF_RND	SSF_TRUNC	SSYN	STI	STRUCT
STT_TYPE	SU	SYSCFG		
T	TESTSET	TFU	TH	TL
TST	UNLINK	UNLNK	UNRAISE	UU
V	VIT_MAX	W	W32	WEAK
X	XB	XH	XOR	Z

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.

- Begin symbols with alphabetic characters.

  Symbols can use alphabetic characters (`A–Z` and `a–z`), digits (`0–9`), and 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 VisualDSP++ environment. Such symbols will not appear in the symbol table accessible in the debugger. A symbol name in which the first two characters are points will not appear even in the symbol table of the object.

  The compiler and runtimes prepend '_' to avoid using symbols in the user name space 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. The linker uses `.SECTION` name symbols to place code and data in Blackfin processor's memory. For more details, see the *VisualDSP++ 3.1 Linker and Utilities Manual for Blackfin Processors*.

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

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

- The reserved word list for Blackfin 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 stand alone 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 and symbolic.

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

7.6r — .8r
```

For information about fractional type support, refer to "Fractional Type Support" on page 1-28.

Symbolic expressions
Symbolic expressions contain symbols, whose values may not be known until link time:

```
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 combines that meaning of a symbol with a use of the length operator (see Table 1-4 on page 1-26).

Assembler Operators

Table 1-3 lists the assembler's numeric and bitwise operators used in constant expressions and address expressions. These operators are listed in groups order from highest to lowest precedence. Operators with highest precedence are evaluated first. When two operators have the same precedence, the assembler evaluates the leftmost operator first. Relational operators are only supported in relational expressions in conditional assembly, as described in "Conditional Assembly Directives" on page 1-30.

Table 1-3. Operator Precedence

Operator	Usage Description	Designation
`(expression)`	`expression` in parentheses evaluates first	
~ -	Ones complement Unary minus	Tilde Minus
* / %	Multiply Divide Modulus	Asterisk Slash Percentage
+ -	Addition Subtraction	Plus Minus
<< >>	Shift left Shift right	
&	Bitwise AND (preprocessor only)	
|	Bitwise inclusive OR	
^	Bitwise exclusive OR (preprocessor only)	

The assembler also supports special operators. Table 1-4 lists and describes these operators used in constant and address expressions.

Table 1-4. Special Assembler Operators

Operator	Usage Description
`ADDRESS`(symbol)	Address of `symbol`
`BITPOS`(constant)	Bit position
`symbol`	Address pointer to `symbol`
`LENGTH(symbol)`	Length of `symbol` in number of elements (in a buffer/array)

The "address of" and "length of" operators can be used with external symbols—apply it to symbols that are defined in other sections as `.GLOBAL` symbols.

The following example demonstrates how the 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 sample

.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;
```

This code fragment initializes `p0` and `p1` to the base address and length, respectively, of the buffer `real_data`. The loop will be 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));
```

Numeric Formats

The assembler supports binary, decimal, hexadecimal, and fractional numeric formats (bases) within expressions and assembly instructions. Table 1-5 describes the conventions of notation the assembler uses to distinguish between numeric formats.

Note that due to the support for `b#` and `B#` binary notation, the preprocessor stringization functionality has been turned off by default to avoid possible undesired stringization. For more information, refer to "# (Argument)" on page 2-27 and the preprocessor's "-stringize" command-line switch (on page 2-37), and to the assembler's "-flags-pp -opt1 [,-opt2...]" command-line switch (on page 1-83).

Table 1-5. Numeric Formats

Convention	Description
`0x`*`number`*	"`0x`" prefix indicates a hexadecimal number

`B#`*`number`*
`b#`*`number`*
	"`B#`" or "`b#`" prefix indicates a binary number

`number`	No prefix indicates a decimal number
`number``r`	"`r`" suffix indicates a fractional number

Fractional Type Support

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

The legal range is [– `1…1`). which means the values must be greater than or equal – `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 E37] …Fract constant '1.5r' is out of range.
   Fract constants must be greater than or equal to -1 and
   less than 1. */
```

Fract 1.15 is a 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

The Blackfin processors support fracts that use 1.31 format, meaning 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 `2**31`.

The conversion formula used by a Blackfin processor to convert from the floating-point to fixed-point 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-03;
   // 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` for the closest approximation of `1.0r` within the 1.15 representation.

Fractional Arithmetic

The assembler provides supports for arithmetic expressions using operations on fractional constants, consistent with the support for other numeric types in constant expressions, as described in "Assembler Expressions" on page 1-24.

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,

```
.VAR myFract = 1 - 0.5r;
[Error ea5004] Syntax Error: .VAR myFract = 1 - 0.5r.
```

Comment Conventions

The assembler supports C and C++ style formats for inserting comments in assembly sources. The `easmblkfn` assembler does not support nested comments. Table 1-6 lists and describes assembler comment conventions.

Table 1-6. 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. In addition to integer arithmetic, the operands may be the C struct in assembly built-ins `SIZEOF()`and `OFFSETOF()` that return integers.

The conditional assembly directives are:

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

All conditional assembly blocks begin with an `.IF` directive and end with an `.ENDIF` directive. Table 1-7 on page 1-32 shows examples of conditional directives.

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

The `".IF"`, `".ELSE"`, `".ELIF` “, and `".ENDIF"` directives (in any case) are reserved keywords.

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

In addition, the `SIZEOF()` and `OFFSETOF()` built-ins (see “C Struct Support in Assembly Built-In Functions” on page 1-33) can be used in relational expressions. Different code sequences can be included based on the result of the expression. For example, a `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:

- The # directives are evaluated during preprocessing by the `PP` preprocessor. Therefore, preprocessor `#IF` directives cannot use the 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 would be able to write a relational or logical expression whose value will depend on the value of a `#define`:. For example,

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

  If you have "`#define tryit 2`", then the code `<some code>` will be assembled, `<some more code>` will not be.

- There are no parallel assembler directives for C-style directives `#define`, `#include`, `#ifdef`, `#if defined(name)`, `#ifndef`, etc..

Table 1-7. Relational Operators for Conditional Assembly

Relational Operators	**Conditional Directive Examples**
`not !`	`.if !0;`
`greater than >`	`.if ( sizeof(myStruct) > 16 );`
`greater than equal >=`	`.if ( sizeof(myStruct) >= 16 );`
`less than <`	`.if ( sizeof(myStruct) < 16 );`
`less than equal <=`	`.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);`

C Struct Support in Assembly Built-In Functions

The assembler supports built-in functions that enable you to pass information obtained from the imported C struct layouts. The supported built-in functions are `OFFSETOF()` and `SIZEOF()`.

OFFSETOF() Built-In

The `OFFSETOF()` built-in is used to calculate the offset of a specified member from the beginning of its parent data structure. For Blackfin processors, `OFFSETOF()` units are in bytes.

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

where:

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

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

SIZEOF() Built-In

The `SIZEOF()` built-in 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:

`SIZEOF()` built-in takes a symbolic reference as its single argument. A symbolic reference is a name followed by zero or more qualifiers to members.

The `SIZEOF()` built-in gives the amount of storage associated with:

- An aggregate type (structure)
- A C base type (`int`, `char`, etc.)
- A member of a structure (any type)

For example,

```
.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
```

The `SIZEOF()` built-in returns the size in the units appropriate for its processor. For Blackfin DSPs, 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 “->”. For example,

```
myStruct->Member5
```

references the address of `Member5` located within `myStruct`. If the struct layout changes, there is no need to change the reference. The assembler re-calculates the offset when the source is re-assembled with the updated header.

Nested `struct` references are supported. For example,

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

Unlike `struct` members in C, `struct` members in the assembler are always referenced with "->" (and not ".") because "." is a legal character in identifiers in assembly and not available as a `struct` reference.

References within nested structures are permitted. A nested struct definition can be provided in a single reference in assembly code while a nested `struct` via a pointer type requires more than one instruction. Make use of the `OFFSETOF()` built-in to avoid hard-coded offsets that could 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

```
.EXTERN STRUCT myStructTag _myStruct;
P3.l = _myStruct->NestedOne->State;
P3.h = _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`, use 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

```
// in assembly file
.EXTERN STRUCT myStructTagWithPtr _myStructWithPtr;
P1.l = _myStructWithPtr->pNestedOne;
P1.h = _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 Blackfin processor assembler supports the directives shown in Table 1-8. A description of each directive appears in the following sections.

Table 1-8. Assembler Directive Summary

Directive	Description
`.ALIGN` (see on page 1-40)	Specifies an alignment requirement for data or code
`.BYTE| .BYTE2| .BYTE4` (see on page 1-42)	Defines and initializes one-, two-, and four-byte data objects, respectively
`.EXTERN` (see on page 1-45)	Allows reference to a global symbol
`.EXTERN STRUCT` (see on page 1-46)	Allows reference to a global symbol (`struct`) that was defined in another file
`.FILE` (see on page 1-48)	Overrides `filename` given on the command line. Used by C compiler
`.GLOBAL` (see on page 1-49)	Changes a symbol's scope from local to global

Table 1-8. Assembler Directive Summary (Cont'd)

Directive	Description
`.IMPORT` (see on page 1-49)	Provides the assembler with the structure layout (C struct) information
`.LEFTMARGIN` (see on page 1-52)	Defines the width of the left margin of a listing
`.LIST` (see on page 1-53)	Starts listing of source lines
`.LIST_DATA` (see on page 1-54)	Starts listing of data opcodes
`.LIST_DATFILE` (see on page 1-55)	Starts listing of data initialization files
`.LIST_DEFTAB` (see on page 1-56)	Sets the default tab width for listings
`.LIST_LOCTAB` (see on page 1-57)	Sets the local tab width for listings
`.LIST_WRAPDATA` (see on page 1-58)	Starts wrapping opcodes that do not fit listing column
`.NEWPAGE` (see on page 1-59)	Inserts a page break in a listing
`.NOLIST` (see on page 1-53)	Stops listing of source lines
`.NOLIST_DATA` (see on page 1-54)	Stops listing of data opcodes
`.NOLIST_DATFILE` (see on page 1-55)	Stops listing of data initialization files
`.NOLIST_WRAPDATA` (see on page 1-58)	Stops wrapping opcodes that do not fit listing column
`.PAGELENGTH` (see on page 1-60)	Defines the length of a listing page
`.PAGEWIDTH` (see on page 1-61)	Defines the width of a listing page

Table 1-8. Assembler Directive Summary (Cont'd)

Directive	Description
`.PREVIOUS` (see on page 1-62)	Reverts to a previously described `.SECTION`
`.SECTION` (see on page 1-64)	Marks the beginning of a section
`.STRUCT` (see on page 1-66)	Defines and initializes data objects based on C `typedefs` from `.IMPORT` C header files
`.TYPE` (see on page 1-70)	Changes the default data type of a symbol. Used by C compiler
`.VAR` (see on page 1-71)	Defines and initializes 32-bit data objects
`.WEAK` (see on page 1-75)	Creates a weak definition or reference

.ALIGN, Specify an Address Alignment

The `.ALIGN` directive forces the address alignment of an instruction or data item. Use it to ensure section alignments in the `.LDF` file. You may use `.ALIGN` to ensure the alignment of the first element of a section, therefore providing the alignment of the object section ("`INPUT SECTION`" to the linker). You may also use the `INPUT_SECTION_ALIGN(number)` linker command in the `.LDF` file to force all the following input sections to the specified alignment.

Refer to Chapter 1 "*Linker*" in the *VisualDSP++ 3.1 Linker and Utilities Manual for Blackfin Processors* for more information on section alignment.

Syntax:

```
.ALIGN expression;
```

where

- *`expression`* evaluates to an integer. It specifies the byte 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 evenly divided by the value of *`expression`*, with no remainder. The expression set to 0 or 1 signifies no address alignment requirement.

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

Example:

```
.ALIGN 0;       /* no alignment requirement */
…
.ALIGN 1;       /* no alignment requirement */
…
.SECTION data1;
.ALIGN 2;
.BYTE single;
                /* aligns the data item on a half-word boundary,
            at the location with the address value that can
            be evenly divided by 2 */
.ALIGN 4;
.BYTE samples1[100]=”data1.dat”;
            /* aligns the first data item on a word boundary,
            at the location with the address value that can be
            evenly divided by 4; advances other data items
            consequently */
```

.BYTE, Declare a Byte Data Variable or Buffer

The `.BYTE`, `.BYTE2`, and `.BYTE4` directives declare and optionally initialize one-, two-, or four-byte data objects. 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,varName2,... = initExpression1, initExpression2,…;
.BYTE bufferName[] = initExpression1, initExpression2,…;
.BYTE bufferName[] = “fileName";
.BYTE bufferName[length ] = ” fileName“;
.BYTE bufferName1[length] [,bufferName2[length],…];
.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 current directory of your operating system, only the *filename* need be given inside double quotes.

  If the file name is not found in the current directory, rhe aassembler will look in the directories in the processor `include` path. You may use the `-I` switch (see on page 1-84) to add an 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—set initial values for variables and buffer elements.

ⓘ The optional `[`*`length`*`]` parameter defines the length of the associated buffer in words. The number of initialization elements defines *`length`* of an implicit-size buffer. The brackets `[ ]` that enclose the optional [*`length`*] are required. For more information, see the following `.BYTE` examples.

ⓘ Use a `/R32` qualifier (`.BYTE4/R32`) to support 32-bit initialization for use with 1.31 fracts (see on page 1-28).

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

```
.BYTE = 5, 6, 7;
     // initialize three 8-bit memory locations
.BYTE samples[] = 123, 124, 125, 126, 127;
     // declare an implicit-length buffer and initialize it
     // with five 1-byte constants
.BYTE4 points[] = 1.01r, 1.02r, 1.03r;
     // 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`, take under consideration constraints applied to the `.VAR` directive. The `.VAR` directive allocates and optionally initializes 32-bit data objects. For information about the `.VAR` directive, refer to information on page 1-71.

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. The characters are stored in the upper byte of 32-bit words. The LSBs are cleared. The assembler also accepts ASCII characters within comments

In Blackfin processors, ASCII initialization can be provided in `.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 where each `char` needs four bytes.

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.

.EXTERN, Refer to a Globally Available Symbol

The `.EXTERN` directive allows a code module to reference global data structures, symbols, etc. that are declared as `.GLOBAL` in other files. For additional information, see the `.GLOBAL` directive on page 1-49.

Syntax:

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

where

`symbolName` — the name of a global symbol to import. A single `.EXTERN` directive can reference any number of symbols 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 that was 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 that was declared in another file. The naming conventions are the same for structs 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 that was 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 does reference 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:

```
.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 = _StNick->isSeniorCitizen;
P3.h = _StNick->isSeniorCitizen;
```

.FILE, Override the Name of a Source File

The `.FILE` directive overrides the name of the source file. This directive may 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 had 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.

Example:

```
.FILE “vect.c”;      // the argument may be a *.c file
.SECTION data1;
    …
    …
```

.GLOBAL, Make a Symbol Globally Available

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 it only from the local file, that is, the same file in which it is defined. It 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.

You 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 the `.EXTERN` directive on page 1-45. Note that `.GLOBAL` (or `.WEAK`) scope is required for symbols that appear in the `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 separated by commas.

Example:

```
.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 the `sizeof()` C base types).

Syntax:

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

where

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

The `.IMPORT` directive does not allocate space for a variable of this type—that requires the `.STRUCT` directive (on page 1-66).

The assembler takes advantage of knowing the struct layouts. The assembly programmer may reference struct data members by name in assembler source, as one would do 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 need not change. It just needs to get the new layout from the header file, via the compiler. The make dependencies track the `.IMPORT` header files and know when a re-build is needed. Use the `-flags-compiler` assembler switch option (see on page 1-81) to pass options to the C compiler for the `.IMPORT` header file compilations.

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

The `.IMPORT` directive with one or more `.EXTERN` directives allows code in the module to refer to a struct variable that was declared and initialized elsewhere. The C struct can either 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 the `.EXTERN` directive on page 1-45 and the `.STRUCT` directive on page 1-45.

Example:

```
.IMPORT "CHeaderFile.h";
.IMPORT "ACME_IIir.h","ACME_IFir.h";
```

.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 checks 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 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. */
```

ⓘ 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/.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, will be 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` will appear in the listing (and thus account 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.

.LIST_DATA/.NOLIST_DATA, Listing Data Opcodes

The `.LIST_DATA/.NOLIST_DATA` directives (off by default) turn the listing of data opcodes on or off. If `.NOLIST_DATA` is in effect, opcodes corresponding to variable declarations will not be shown 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 will not affect the parent source file.

The `.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_DATFILE/.NOLIST_DATFILE, Listing Data Initialization Files

The `.LIST_DATFILE/.NOLIST_DATFILE` directives (off by default) turn the listing of data initialization files on or 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_DEFTAB, Set the Default Tab Width 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 control, 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 while the `.LIST_LOCTAB` directive sets the local tab width (see on page 1-57).

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;
```

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. A value of 0 sets 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_LOCTAB, Set the Local Tab Width 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 control, but it can be overridden if the local tab width is explicitly set with a directive.

The `.LIST_LOCTAB` directive sets the local tab width, and the `.LIST_DEFTAB` directive sets the default tab width (see on page 1-56).

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_LOCTAB expression;
```

where

`expression`—evaluates to an integer greater than or equal to 0. A value of 0 sets the local tab width to the current setting of the default tab width.

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

Example: See the `.LIST_DEFTAB` example on page 1-56.

.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 would not fit, but in practice, such a value will almost always be 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 will 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.

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

.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` switch (see on page 1-84).

Syntax:

```
.PAGELENGTH expression;
```

where

`expression`—evaluates to an integer of equal or greater than 10, or 0. It specifies the number of text lines per printed page. The default page length is now 0, which means the listing will have 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 will issue a warning and use its internal minimum page length (approximately 10 lines).

Example:

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

You can set the page length only once per source file. If the assembler encounters multiple occurrences of the directive, it ignores all of them 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 (see on page 1-84).

Syntax:

```
.PAGEWIDTH expression;
```

where

> *`expression`*—evaluates to an integer. Depending on setting of the `.LEFTMARGIN` directive, this integer should be at least equal to the `LEFTMARGIN` value plus 46.You can speify the *`expression`* value as any integer over 46. You cannot set this integer to be less than 46. There is no upper limit. For example, if `LEFTMARGIN = 0` and the `.PAGEWIDTH` is not specified, the actual page width is set to 46.

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

Example:

```
.PAGEWIDTH 26;     // starts a new line after 72 characters
                   // are printed on one line, assuming
                   // the .LEFTMARGIN setting is 0.
```

> ⓘ 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.

.PREVIOUS, Revert to the Previously Defined Section

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

Syntax:

```
.PREVIOUS;
```

The following examples provide illegal and legal cases of the use of the consecutive `.PREVIOUS` directives.

Example of Illegal Directive Use

```
.SECTION data1;      // data
.SECTION program;    // instructions
.PREVIOUS;           // previous section ends, back to data1
.PREVIOUS;           // no previous section to set to
```

Example of Legal Directive Use

```
#define MACRO1 \
.SECTION data2; \
   .VAR vd = 4; \
.PREVIOUS;

.SECTION data1;      /* data */
   .VAR va = 1;
.SECTION program;    /* instructions */
.VAR vb = 2;

/* MACRO1 */
MACRO1
.PREVIOUS;
   .VAR vc = 3;
```

evaluates as:

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

.SECTION, Declare a Memory Section

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

Syntax:

```
.SECTION sectionName [sectionType];
```

where

- *`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 `...\Blackfin\ldf` subdirectory of the VisualDSP++ installation directory, or write your own `.LDF` file.

  Note: Some sections starting with "." names have certain meaning within the linker. The dot (.) should not be used as the initial character in *`sectionName`*.

  > The assembler generates relocatable sections for the linker to fill in the addresses of symbols at link time. The assembler implicitly pre-fix the name of the section with the "`.rela.`" string to form a relocatable section. For example, for the sections named `data1` and `program`, the relocation section are `.rela.data1` and `..rela.program,` respectively. To avoid such an ambiguity, ensure that your sections' 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. Valid `sectionTypes` are described in the `ELF.h` header file, which is

available from third-party software development kits.
For more information on the ELF file format, see the *VisualDSP++ 3.1 Linker and Utilities Manual for Blackfin Processors*.

Example:

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

.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`—the type definition for a struct VAR

`structName`—a `struct` name.

`struct-member-initializers`—per struct member initializers

The `{ }` curly braces are used for consistency with the C initializer syntax. Initialization can be in “long” 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:

- Change C compiler keyword “`struct`” to “`.struct`”
- Change C compiler constant string syntax “`MyString`” to `'MyString'`

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
};
```

then `member2` will be 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 will be 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. The struct name is implied.

For example, the reference "`scalar`" ("`nestedOne->scalar`" implied) and "`nested->scalar1`" ("`nestedOne->nested->scalar1`" implied).

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

.TYPE, Change Default Symbol Type

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

Syntax:

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

where

> *`symbolName`*—the name of the object to which the *`symbolType`* should be applied.
>
> *`symbolType`*—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 32-Bit Data Variable or Buffer

The `.VAR` directive declares and optionally initializes 32-bit 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 may 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.

  Unless the absolute placement for a variable is specified with the `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 `k` in the processor's memory. Therefore, code example may look as:

  ```
  .VAR d;
  .VAR f;
  .VAR k[50];
  ```

- 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. For implicit-size buffer initialization, the elements may appear within curly brackets { }. 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 varName1 = initexpression1 [,varName2 = initexpression2,…];
.VAR bufferName[] = initExpression1, initExpression2,…;
.VAR bufferName[] = "fileName";
.VAR bufferName[length] = "fileName";
.VAR bufferName1[length] [,bufferName2[length],…];
.VAR bufferName[length] = initExpression1,initExpression2,…;
```

where:

- *`varName`* —represents user-defined symbols that identify variables.
- *`bufferName`* —represents 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 current directory of your operating system, only the *`fileName`* need be given quotes.

  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.
- `[`*`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.

Use a `/R32` qualifier (`.VAR/R32`) to support 32-bit initialization for use with 1.31 fracts (see on page 1-28).

The following lines of code demonstrate some `.VAR` directives:

```
.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]
.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/R32 Fract_Var_R32[] = "fr32FormatFract.dat";
```

All Blackfin processor's memory accesses should have proper alignment. This means that when loading or storing a `N`-byte value into the processor, ensure that this value is aligned in memory by `N` boundary, or a hardware exception would be generated.

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 assembler supports ASCII string initialization. The `easmblkfn` assembler supports ASCII string initialization. Refer to the `.BYTE` directive description on page 1-44 for more information.

.WEAK, Support a 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—the user-defined symbol

While the linker will generate an error if two objects define global symbols with identical names, it will allow 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 will not extract objects from archives to satisfy weak references. Such references, left unresolved, have the value of 0.

ⓘ Note that `.WEAK` (or `.GLOBAL` scope) is required for symbols that appear in the `RESOLVE` commands in the `.LDF` file.

Assembler Command-Line Reference

This section describes the `easmblkfn` 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 VisualDSP++ environment.

This section contains:

- "Running the Assembler" on page 1-77
- "Assembler Command-Line Switch Summary" on page 1-79
- "Assembler Command-Line Switch Descriptions" on page 1-81

Command-line switches control certain aspects of the assembly process, including library searching, listing, and preprocessing. Because the assembler automatically runs the preprocessor as your program is assembled (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 Chapter 2, "Preprocessor".

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

See "Specifying Assembler Options in VisualDSP++" on page 1-89 for more information.

Running the Assembler

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

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

runs the assembler with

`easmblkfn` — name of the assembler program for Blackfin processors

`-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:

- *`ShortFileName`*—a file name without quotes (no special characters)
- *`LongFileName`*—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 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.

Table 1-9 summarizes file extension conventions accepted by the VisualDSP++ environment.

Table 1-9. File Name Extension Conventions

Extension	File Description
`.asm`	Assembly source file **Note:** Files with unrecognized extensions are treated 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-BF535 -l p1.lst -Dmaximum=100 -v -o bin\p1.doj p1.asm
```

runs the assembler with

`-proc ADSP-BF535` — specifies assembles instructions unique to ADSP-BF535 processor.

`-l p1.lst` — directs the assembler to output the listing file.

`-Dmaximum=100` — defines the preprocessor macro `maximum` 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 Summary

This section describes the `easmblkfn` command-line switches in ASCII collation order. A summary of the assembler switches appears in Table 1-10. Refer to "Assembler Command-Line Switch Descriptions" starting on page 1-81 for a detailed description of each assembler switch.

Table 1-10. Assembler Command-Line Switch Summary

Switch Name	Purpose
`-D`*`macro`*`[=`*`definition`*`]` (on page 1-81)	Passes macro definition to the preprocessor.
`-flags-compiler -opt1 [,-opt2...](` (on page 1-81)	Passes each comma-separated option to the compiler. (Used when compiling `.IMPORT` C header files.)
`-flags-pp -opt1 [,-opt2...]` (on page 1-83)	Passes each comma-separated option to the preprocessor.
`-g(` (on page 1-83)	Generates debug information (DWARF-2 format).
`-h[elp]` (on page 1-83)	Outputs a list of assembler switches.
`-i|I` *`directory pathname`* (on page 1-84)	Searches a directory for included files.
`-l` *`filename`* (on page 1-84)	Outputs the named listing file.
`-li` *`filename`* (on page 1-85)	Outputs the named listing file with `#include` files expanded.
`-M` (on page 1-85)	Generates make dependencies for `#include` and data files only; does not assemble. For example, `-M` suppresses the creation of an object file.
`-MM` (on page 1-85)	Generates make dependencies for `#include` and data files. Use `-MM` for make dependencies with assembly.
`-Mo` *`filename`* (on page 1-86)	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.

Table 1-10. Assembler Command-Line Switch Summary (Cont'd)

Switch Name	Purpose
`-Mt` *`filename`* (on page 1-86)	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`'.
`-micaswarn` (on page 1-86)	Treats multi-issue conflicts as warnings.
`-o` *`filename`* (on page 1-86	Outputs the named object [binary] file.
`-pp(` (on page 1-87)	Runs the preprocessor only; does not assemble.
`-proc` *`processor`* (on page 1-87)	Specifies a processor for which the assembler should produce suitable code.
`-sp` (on page 1-88)	Assembles without preprocessing.
`-stallcheck=(none|cond|all)` (on page 1-88)	Displays stall information: • `none` - no messages • `cond` - conditional stalls only (default) • `all` - all stall information
`-v[erbose]` (on page 1-88)	Displays information on each assembly phase.
`-version` (on page 1-88)	Displays version information for the assembler and preprocessor programs.
`-w` (on page 1-88)	Removes all assembler-generated warnings.
`-Wnumber` (on page 1-88)	Suppresses any report of the specified warning.

Assembler Command-Line Switch Descriptions

A description of each command-line switch includes information about case-sensitivity, equivalent switches, switches overridden/contradicted by the one described, and naming and spacing constraints on parameters.

-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" on page 1-15 for the list of predefined macros.

Examples:

```
-Dinput              // defines input as 1
-Dsamples=10         // defines samples as 10
-Dpoint="Start"      // defines point as “Start”
                     // (You can use “.” to delimit the macro
                     // defintion.)
```

-flags-compiler

The `-flags-compiler -opt1 [-opt2...]` switch passes each comma-separated option to the C compiler. The switch takes a list of one or more comma-separated compiler options that are passed on the compiler command-line for compiling `.IMPORT` headers. The assembler calls the compiler to process each header file in an `.IMPORT` directive. It calls the compiler with the `-debug-types` option along with any `-flags-compiler` options given on the assembler command line.

For example,

```
// file.asm has .IMPORT "myHeader.h";
easmblkfn -proc ADSP-BF535 -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 legal C compiler options, including `-D` and `-I` options.

User-Specified Defines Options

The `-D` (defines) options on the assembler command line are passed to the assembler preprocessor, but they are not passed to the compiler for `.IMPORT` header processing. If you have `#defines` for the `.IMPORT` header compilation, they must be explicitly specified with the `-flags-compiler` switch. For example,

```
// file.asm has .IMPORT "myHeader.h";
easmblkfn -proc ADSP-BF535 -DaDef -flags-compiler -DbDef,-DbDefTwo=2. file.asm
// -DaDef is not passed to the compiler
ccblkfn -proc ADSP-BF535 -debug-types -flags-compiler -DbDef,-DbDefTwo=2 myHeader.h
```

See "Using Assembler Feature Macros" on page 1-15 for the list of predefined macros including default macros.

Include Options

The `-I` (include search path) options and `-flags-compiler` options are passed to the C compiler for each `.IMPORT` header compilation. The compiler `include` path is always present automatically. Using the `-flags-compiler` option, you can control the order the `include` directories are searched. The `-flags-compiler` switch attributes always take precedence from the assembler's `-I` options.

For example,

```
easmblkfn -proc ADSP-BF535 -I\aPath -DaDef -flags-compiler -I\cPath,-I. file.asm
ccblkfn -proc ADSP-BF535 -I\aPath -DaDef -flags-compiler -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.

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 line number and symbol information in DWARF-2 binary format, allowing you to debug the assembly source files.

-h[elp]

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

-i|I directory

The `-i directory` or `-I directory` (include directory) switch directs the assembler to append the specified directory or a list of directories separated by semicolons (;) to the search path for included files. 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. Current project directory (`.DPJ`)
2. …`\Blackfin\include` subdirectory of the VisualDSP++ installation directory
3. Specified directory (or list of directories). The order of the list defines the order of multiple searches.

Current directory is your `*.dpj` project directory, not the directory of the assembler program. Usage of full path names for the `-I` switch on the command line is recommended. For example,

```
easmblkfn -proc ADSP-BF535 -I \bin\include
```

-l filename

The `-l` (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-BF535 -flags-compiler -I\path,-I. -l file.lst file.asm
```

The file name is a required argument to the `-l` option. For more information, see "Reading a Listing File" on page 1-17.

-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 `-l` option. For more information, see "Reading a Listing File" on page 1-17.

-M

The`M` (generate make rule only) assembler switch directs the assembler to generate make dependency rules, which is 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 `-MM`.

The output, an assembly 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, 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 (on page 1-86) writes make dependencies to the *`filename`* specified instead of `<stdout>`. For consistency with the compilers, when the `-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, which is 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 which 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 quotaton marks (" ").

The `-Mo` *`filename`* option takes precedence over the `-o` *`filename`* option if both are used with `-M`.

-Mt filename

The `-Mt` *`filename`* (output make rule for the 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.

-o [filename]

The `-o` (output) switch directs the assembler to use the specified *`filename`* argument for 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`). If no `-o` setting is specified, 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-BF535 -pp -o test1.is test.asm
        // preprocessed output goes into test1.is
```

```
easmblkfn -proc ADSP-BF535-o “C:\bin\prog3.doj” prog3.asm
          // specify directory for the object file
```

-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 extension `.is`.

-proc processor

The `-proc` *`processor`* (target processor) switch specifies that the assembler should produce code suitable for the specified processor.

The *`processor`* identifiers directly supported in VisualDSP++ 3.1 are:

`ADSP-BF531`, `ADSP-BF532`, `ADSP-BF533`,
`ADSP-BF535`, `ADSP-DM102`, and `AD6532`

For example,

```
easm21K -proc ADSP-BF535 -o bin\p1.doj p1.asm
```

If the processor identifier is unknown to the assembler, it attempts to read required switches for code generation from the file `<processor>.ini`. The assembler searches for the `.ini` file in the VisualDSP ++ System folder. For custom processors, the assembler searches the section “`proc`” in the `<processor>.ini` for key '`architecture`'. The custom processor must be based on an architecture key that is one of the known processors.

For example, `-proc Custom-xxx` searches the `Custom-xxx.ini` file.

-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:

`-stallcheck=none`	Display no messages for stall information
`-stallcheck=cond`	Display information about conditional stalls only. (Default)
`-stallcheck=all`	Display all stall information

-v[erbose]

The `-v` or `-verbose` (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.

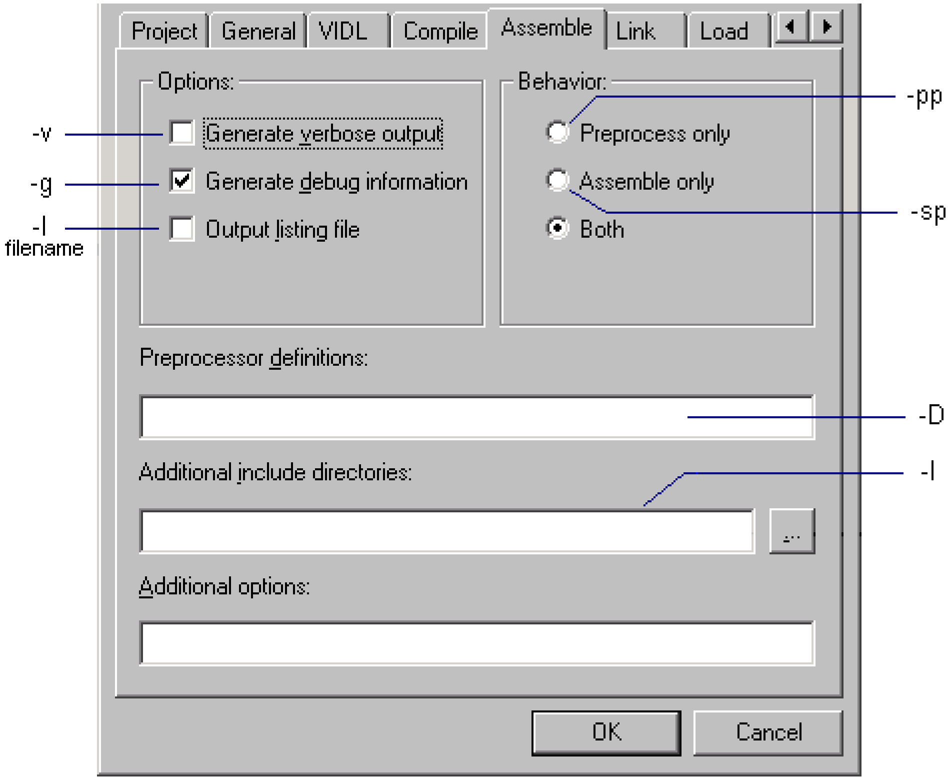
-Wnumber

The `-Wnumber` (warning suppression) switch selectively disables warnings specified by one or more message numbers. For example, `-W1092` disables warning message `ea1092`.

Specifying Assembler Options in VisualDSP++

When using the VisualDSP++ IDDE, use the **Assemble** property page from the **Project Options** dialog box to set assembler functional options (as shown in Figure 1-3).

For more information on assembler configuration, use the VisualDSP++ online Help.



Options:

☐ Generate verbose output

☑ Generate debug information

☐ Output listing file

Behavior:

○ Preprocess only

○ Assemble only

● Both

Preprocessor definitions:

Additional include directories:

Additional options:

Figure 1-3. Project Options – Assemble Property Page

Callouts in Figure 1-3 refer to the corresponding assembler command-line switches described in "Assembler Command-Line Switch Descriptions" starting on page 1-81. The **Additional options** field is used to enter the appropriate file names and options that do not have corresponding controls on the **Assemble** property page but are available as assembler switches.

The assembler options apply to directing calls to `easmblkfn` when assembling `*.asm` files. Changing assembler options in VisualDSP++ does not affect the assembler calls made by the compiler during the compilation of `*.c`/`*.cpp` files.

2 PREPROCESSOR

The preprocessor program (`pp.exe`) evaluates and processes preprocessor commands in source files. With these commands, you 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 `pp` preprocessor is run by other build tools (assembler and linker) from the operating system’s command line or within the VisualDSP++ 3.0 environment. These tools accept command information for the preprocessor and pass it to the preprocessor. The `pp` preprocessor can also operate from the command line with its own command-line switches.

The chapter contains:

- “Preprocessor Guide” on page 2-2
  Contains the information on building programs.
- “Preprocessor Command Reference” on page 2-10
  Describes the preprocessor’s commands, with syntax and usage examples.
- “Preprocessor Command-Line Reference” on page 2-31
  Describes the preprocessor’s command-line switches, with syntax and usage examples.

Preprocessor Guide

This section contains the `pp` preprocessor information on how to build programs from a command line or from the VisualDSP++ 3.1 environment. Software developers using the preprocessor should be familiar with:

- "Writing Preprocessor Commands"
- "Header Files" on page 2-4
- "Writing Macros" on page 2-5
- "Using Predefined Macros" on page 2-7
- "Specifying Preprocessor Options" on page 2-9

The compiler also has it own preprocessor that allows you to use preprocessor commands within your C/C++ source. The compiler preprocessor automatically runs before the compiler. This preprocessor is separate from the assembler and has some features that may not be used within your assembly source files. For more information, see the *VisualDSP++ 3.1 C/C++ Compiler and Library Manual* for the target processors.

The assembler preprocessor differs from the ANSI C standard preprocessor in several ways. First, the assembler preprocessor supports a "?" operator (see on page 2-29) 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 always treated as part of an identifier. This behavior matches the assembler's which uses '.' to start directives and accepts '.' in symbol names. For example,

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

will 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 would treat '`.VAR`' as two tokens, '.' and '`VAR`', and will make the following substitution:

```
.my-var x;
```

This preprocessor behavior was introduced in VisualDSP++ 3.0.

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

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

Writing Preprocessor Commands

Preprocessor commands begin with a pound sign (#) and end with a carriage return. 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 put any characters between the backslash and the carriage return. Unlike assembly directives, preprocessor commands are case sensitive and must be lowercase.

For more information on preprocessor commands, see "Preprocessor Command Reference" on page 2-10.

For example,

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

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

- Including system and user-defined header files
- Defining macros and symbolic constants
- Providing conditional assembly and compilation

You specify preprocessing options with preprocessor commands—lines starting with `#`. Without any commands, the preprocessor performs these three global substitutions:

- Replaces comments with single spaces
- Deletes line continuation characters (`\`)
- Replaces predefined 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

A header file (`.h`) contains lines of source code to be included (textually inserted) into another source file. Typically, the header file contains declarations and macro definitions. The `#include` preprocessor command includes a copy of the header file at the location of the command. There are two main categories of header files:

System Header Files
These files are used to declare global definitions, especially memory mapped registers, system architecture and processors. Use angle brackets to indicate a system header file.

Example:

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

System header files are installed in the **VisualDSP\...Blackfin\include** folder.

User Header Files
These files contain declarations for interfaces between the source files of your program. Use double quotes to indicate a user header file.

Example:

```
#include "defBlackfin.h”
#include "fft_ovly.h"
```

It is a good programming practice to distinguish between system and user header files. The only technical difference between the two different notations is the directory order the assembler searches the specified header file:

The `#include <file>` search order is:

1. include path specified by the `-I` switch
2. `VisualDSP\…21k\include` and `VisualDSP\…Blackfin\include` folders

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

1. local directory
2. include path specified by the `-I` switch
3. `VisualDSP\…Blackfin\include` folders

For syntax information and usage examples on the `#include` preprocessor command, see “#include” on page 2-22.

Writing Macros

The preprocessor processes macros in your assembly source files and Linker Description Files (LDF). Macros are useful for repeating instruction sequences in your source code or defining symbolic constants.

The term *macro* defines a macro-identifying symbol and corresponding definition that the preprocessor uses to substitute the macro reference(s). Macros allow text replacement, file inclusion, conditional assembly, conditional compilation, and macro definition.

Macro definitions start with `#define` and end with a carriage return. If a macro definition is longer than one line, place the backslash character (`\`) at the end of each line except the last, for line continuation. This character indicates that the macro definition continues on the next line and allows to break a long line for cosmetic purposes without changing its meaning.

The macro definition can be any text that would occur in the source file, instructions, commands, or memory descriptions. The macro definition may also have other macro names that will be replaced with their own definitions.

Macro nesting (macros called within another macro) is limited only by the memory that is available during preprocessing. However, recursive macro expansion is not allowed.

Example:

```
#define false 0
#define min(a,b) ((a) < (b) ? (a):(b))
#define xchg(xv,yv)\
   p0=xv;\
   p1=yv;\
   r0=[p0];\
   r1=[p1];\
   [p1]=r0;\
   [p0]=r1;
```

A macro can have arguments. When you pass parameters to a 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, however, 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 are executed for every call.

If a macro ends with a semicolon (`;`), then when it appears in assembly statement, the semicolon is not needed. However, if a macro does not end with a semicolon character (“;”), then it must be followed by the semicolon when appearing in the assembly statement. Users should be consistent in treatment of the semicolon in macro definitions.

For example,

```
#define mac r0=r2+r5         // macro definition
   r2=r1-r0;                 // set parameters
   r5=[p1];mac;              // macro invocation
```

For more syntax information and usage examples for the `#define` preprocessor command, see “#define” on page 2-12.

Using Predefined Macros

In addition to macros you define, the `pp` preprocessor provides a set of predefined and feature macros that you can use in your 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 you can use in your code.

ⓘ Note that the `__DATE__`, `__FILE__`, and `__TIME__` macros return strings within the single quotation marks (‘ ’) suitable for initialization of character buffers (see “.VAR and ASCII String Initialization Support” on page 1-74).

Table 2-1 describes the predefined macros provided by the `pp` preprocessor. Table 2-2 lists feature macros that are defined by the DSP tools to specify the architecture and language being processed.

Table 2-1. Predefined Preprocessor Macros

Macro	Definition
`ADI`	Defines `ADI` as 1.
`__LINE__`	The `__LINE__` macro is replaced with the line number in the source file that the macro appears on.
`__FILE__`	Defines `__FILE__` as the name and extension of the file in which the macro is defined, for example, ‘`macro.asm`’.
`__STDC__`	Defines `__STDC__` as 1.
`__TIME__`	Defines `__TIME__` as current time in the 24-hour format ‘`hh:mm:ss`’, for example, ‘`06:54:35`’.
`__DATE__`	Defines `__DATE__` as current date in the format ‘`Mm dd yyyy`’, for example, ‘`Oct 02 2000`’.

Table 2-2. Feature Preprocessor Macros

Macro	Definition
`__ADSPBLACKFIN__`	Always 1 for Blackfin DSP tools
`__ADSPBF531__`	Present when running `easmblkfn -proc ADSP-BF531` with ADSP-BF531 processor.

`__ADSPBF532__`
`__ADSP21532_=1`
	Present when running `easmblkfn -proc ADSP-BF532` with ADSP-BF532 processor.

`__ADSPBF533__`
`__ADSP21532__=1`
	Present when running `easmblkfn -proc ADSP-BF533` with ADSP-BF533 processor.

`__ADSPBF535__`
`__ADSP21535__=1`
	Present when running `easmblkfn -proc ADSP-BF535` with ADSP-BF535 processor.

`__ADSPDM102__`	Present when running `easmblkfn -proc ADSP-DM102` with ADSP-DM102 processor.

Table 2-2. Feature Preprocessor Macros (Cont'd)

Macro	Definition
`__AD6532__`	Present when running `easmblkfn -proc AD6532` with AD6532 processor.
`_LANGUAGE_ASM`	Always set to 1
`_LANGUAGE_C`	Equal 1 when used for C compiler calls to specify `.IMPORT` headers. Replaces `_LANGUAGE_ASM`.

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 the processing entirely), these DSP 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 DSP development software.

You can specify preprocessor options either from the preprocessor's command line or via the VisualDSP++ environment:

- From the operating system command line, you select the preprocessor's command-line switches. For more information, see "Preprocessor Command-Line Switches" on page 2-32.
- From the VisualDSP++ environment, you select the preprocessor's options in the **Assemble** and **Link** tabs of the **Project Options** dialog boxes, accessible from the **Project** menu.

  For more information, see the *VisualDSP++ 3.1 User's Guide for Blackfin Processors* and online Help. Refer to "Specifying Assembler Options in VisualDSP++" on page 1-89 for the **Assemble** property page.

Preprocessor Command Reference

This section provides reference information about the DSP's preprocessor commands and operators used in source code, including their syntax and usage examples. It provides the summary and descriptions of all preprocessor command and operators.

The preprocessor reads code from a source file (`.ASM`), modifies it according to preprocessor commands, and generates an altered preprocessed source file. The preprocessed source file is a primary input file for the assembler or linker; it is purged when the a binary object file (`.DOJ`) is created.

Preprocessor command syntax must conform to these rules:

- Must be the first non white space character on its line.
- Cannot be more than one line in length unless the backslash character (`\`) is inserted
- Can contain comments containing the backslash character (\)
- Cannot come from a macro expansion

The preprocessor operators are special operators when used in a `#define` command.

Preprocessor Commands and Operators

Table 2-3 lists the preprocessor command set. Table 2-4 lists the preprocessor operator set. Sections that begin on page 2-12 describe each of the preprocessor commands and operators.

Table 2-3. Preprocessor Command Summary

Command/Operator	Description
`#define` (on page 2-12)	Defines a macro
`#elif` (on page 2-15)	Subdivides an `#if` … `#endif` pair
`#else` (on page 2-16)	Identifies alternative instructions within an `#if` … `#endif` pair
`#endif` (on page 2-17)	Ends an `#if` … `#endif` pair
`#error` (on page 2-18)	Reports an error message
`#if` (on page 2-19)	Begins an `#if` … `#endif` pair
`#ifdef` (on page 2-20)	Begins an `#ifdef` … `#endif` pair and tests if macro is defined
`#ifndef` (on page 2-21)	Begins an `#ifndef` … `#endif` pair and tests if macro is not defined
`#include` (on page 2-22)	Includes contents of a file
`#line` (on page 2-23)	Sets a line number during preprocessing
`#pragma` (on page 2-23)	Takes any sequence of tokens
`#undef` (on page 2-25)	Removes macro definition
`#warning` (on page 2-26)	Reports a warning message

Table 2-4. Preprocessor Operator Summary

Command/Operator	Description
# (on page 2-27)	Converts a macro argument into a string constant. By default, this operator is OFF. Use the command-line switch "-stringize" on page 2-37 to enable it.
## (on page 2-28)	Concatenates two tokens
? (on page 2-29)	Generates unique labels for repeated macro expansions
... (on page 2-13)	Specifies a variable length argument list

#define

The `#define` command defines macros.

When you define a macro 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 that are longer than one line, use the backslash character (`\`) at the end of each line except for the last line. 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.

(`arg1,arg2,…`) — optional list of arguments enclosed in parenthesis and separated by commas. No space is permitted between the macro name and the left parenthesis. If there is a space, the parenthesis and arguments are treated as if the space is part of the definition.

`replacementText` — text to substitute each occurrence of `macroSymbol` in your source code.

Examples:

```
#define BUFFER_SIZE 1020
       /* Defines a constant named BUFFER_SIZE and sets its
```

```
        value to 1020.*/

#define MINIMUM (X, Y) ((X) < (Y)? (X): (Y))
         /* Defines a macro named MINIMUM that selects the
         minimum of two numeric arguments. */
#define copy(src,dest)
p1=src;\
p2=dst;\
r0=[p1];\
[p2]=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(0x1000,0x2000);
         calls the macro, passing parameters to it.

         The preprocessor replaces the macro with the code:
            p1=0x1000; p2=0x2000; r0=[p1]; [p2]=r0;
*/
```

Variable Length Argument Definitions

The definition of a macro can also be defined with a variable length argument list (using the ... operator).

```
#define test(a, ...)  <definition>
```

defines a macro `test` which takes two or more arguments. It is invoked as any other macro, although the number of arguments can vary.

For example,

`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

In the macro definition, the identifier `__VA_ARGS__` is available to take on the value of all of the trailing arguments, including the separating commas, all of which are merged to form a single item. For example,

```
#define test(a, ...) bar(a); testbar(__VA_ARGS__);
```

expands as

```
test (1,2) -> bar(1); testbar(2);

test (1,2,3,4,5) -> bar(1); testbar(2,3,4,5);
```

#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`. You can have an unlimited number of `#elif` commands inside one `#if` … `#end` pair.

Syntax:

```
#elif condition
```

where

condition — expression to evaluate as `TRUE` (non zero) or `FALSE` (zero)

Example:

```
#if X == 1
   …
#elif X == 2
   …
      /* The preprocessor includes text within the section if
      the test is true and excludes all alternatives within
      #if ... elif pair.  */
#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. Make sure 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 the backslash character (\) at the end of each line except for the last line.

Example:

```
#ifndef __ADSPBF535__
#error \
        MyError:\
        Expecting a ADSP-BF535. \
        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 *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` (non zero) or `FALSE` (zero)

Example:

```
#if x!=100/* test for TRUE condition */
…
      /* The preprocessor includes text within the section
      if the test is true and excludes all other text
      after #if only when x!=100 */
#endif
```

More examples:

```
#if (x!=100) && (y==20)

#if defined(__ADSPBF535__)
```

#ifdef

The `#ifdef` (if defined) command begins an `#ifdef … #endif` pair and instructs the preprocessor to test whether macro is defined. The number of `#ifdef` commands must match the number of `#endif` commands.

Syntax:

```
#ifdef macroSymbol
```

where

`macroSymbol` — macro identifying symbol

Example:

```
#ifdef __ADSPBF535__
      /* Includes text after #ifdef only when __ADSPBF535__ has
      been defined */
#endif
```

#ifndef

The `#ifndef` command (if not defined) 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. The number of `#ifndef` commands must equal the number of `#endif` commands.

Syntax:

```
#ifndef macroSymbol
```

where

`macroSymbol` — macro identifying symbol

Example:

```
#ifndef __ADSPBF535__
     /* Includes text after #ifndef only when __ADSPBF535__ has
     not been 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. The only difference to the preprocessor between these two types of files is the way the preprocessor searches for them.

- System Header `<fileName>` — The preprocessor searches for a system header file in the order: (1) the directories you specify and (2) the standard list of system directories.
- User Header "`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

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 __ADSPBF535__   /* Tests that __ADSPBF535__ 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`— number of the source line that you want to output

`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”
```

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 `#include` directives that were processed.

#pragma

The `#pragma` is the implementation-specific command that could modify the preprocessor behavior. The `#pragma` command can take any sequence of tokens. This command is accepted for compatibility with other VisualDSP++ software tools. The `pp` preprocessor currently does not support pragmas; therefore, it will ignore any information in the `#pragma`.

Syntax:

```
#pragma any_sequence_of_tokens
```

Example:

```
#pragma disable_warning 1024
```

#undef

The `#undef` command directs the preprocessor to undefine the 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 is used to cause 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 the backslash character (\) at the end of each line except for the last line.

Example:

```
#ifndef __ADSPBF535__
#warning \
      MyWarning: \
      Expecting an ADSPBF535. \
      Check the Linker Description File!
#endif
```

# (Argument)

The # (argument) “stringanization” 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 any 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 the double quotation marks.

This feature is "off" by default. Use the “-stringize” command-line switch (on page 2-37) to enable it.

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
   an 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, while "`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;
```

Preprocessor Command-Line Reference

The `pp` preprocessor is the first step in the process of building (assembling, compiling, linking) your programs. The `pp` preprocessor is run before the assembler and compiler from the assembler or linker. You can also run it independently from its own command line.

This section contains:

- "Running the Preprocessor"
- "Preprocessor Command-Line Switches" on page 2-32

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]
```

where:

`pp` — Name of the preprocessor program

`-switch1, -switch2` — 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 and absolute path names. The `pp.exe` outputs a list of command-line switches when runs 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

Most switches without arguments can be negated by prepending `-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 controlled through the switches (or VisualDSP++ options) of other DSP development tools, such as the compiler, assembler, and linker. Note that the preprocessor (`pp.exe`) can operate independently from the command line with its own command-line switches.

Table 2-5 lists the `pp.exe` switches. A detailed description of each switch appears beginning on page 2-33.

Table 2-5. Preprocessor Command-Line Switch Summary

Switch Name	**Description**
`-cpredef`	Enables the stringization operator
`-cs!`	Treats as a comment all text after "`!`" on a single line

Table 2-5. Preprocessor Command-Line Switch Summary (Cont'd)

`-cs/*`	Treats as a comment all text within `/*   */`
`-cs//`	Treats as a comment all text after `//`
`-cs{`	Treats as a comment all text within `{  }`
`-csall`	Accepts comments in all formats
`-Dmacro[=definition]`	Defines `macro`
`-h[elp]`	Outputs a list of command-line switches
`-i|Idirectory`	Searches `directory` for included files
`-M`	Makes dependencies only
`-MM`	Makes dependencies and produces preprocessor output
`-Mo filename`	Specifies `filename` for the make dependencies output file
`-Mt filename`	Makes dependencies for the specified source file
`-o filename`	Outputs named object file
`-stringize`	Enables stringization (includes a string in quotes)
`-v[erbose]`	Displays information about each preprocessing phase
`-version`	Displays version information for preprocessor.

The following sections describe each of the preprocessor command-line switches.

-cpredef

Directs the preprocessor to use double quotation marks rather than the default single quotes as string delimiters for the predefined macros that are expressed as string constants. (See Table 2-1 on page 2-8 for the predefined macros).

-cs!

Directs the preprocessor to treat as a comment all text after “!” on a single line.

-cs/*

Directs the preprocessor to treat as a comment all text within /* */ on multiple lines.

-cs//

Directs the preprocessor to treat as a comment all text after `//` on a single line.

-cs{

Directs the preprocessor to treat as a comment all text within { }.

-csall

Directs the preprocessor to accept comments in all formats.

-Dmacro[=def]

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. Similar to the C compiler, you can use the `-D` switch to define an assembly language constant macro.

Some examples of this switch are:

```
-Dinput             // defines input as 1
-Dsamples=10        // defines samples as 10
-Dpoint="Start"     // defines point as “Start”
-D_LANGUAGE_ASM=1   // defines assembly language as 1
```

-h[elp]

Directs the preprocessor to output to standard output the list of command-line switches with a syntax summary.

-i|Idirectory

Directs the preprocessor to append the specified directory (or a list of directories separated by semicolon`)` to the search path for included header files (see on page 2-22).

Note that no space is allowed between `-i` or `-I` and the path name.

The preprocessor searches for included files in these order:

1. Current project (`.DPJ`) directory (where the source file lives)
2. Specified directory (a list of directories). The order of the list defines the order of multiple searches.
3. `...\include` subdirectory of the VisualDSP++ installation directory
4. Connected project directories `(.DPJ`)

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.

-M

Directs the preprocessor to output a rule (generate make rule only), which is 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 the "-o filename" option is used with `-M`, the `-o` option is ignored. To specify an alternate target name for the make dependencies, use the "-Mt filename" option. To direct the make dependencies to a file, use the "-Mo filename" option.

-MM

Directs the preprocessor to output a rule (generate make rule and preprocess), which is 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

Specifies the name of the make dependencies file (output make rule) that the preprocessor generates when using the `-M` or `-MM` switch. If the named file is not in the current directory, you must provide the path name in the double quotation marks (" "). The "-o filename" option overrides default of make dependencies to `stdout`.

-Mt filename

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 `-M` *`fileneme`* switch overrides the default `base.doj`. See "-M" for more information.

-o filename

Directs the preprocessor to use (output) the specified `filename` argument for the preprocessed assembly file. The preprocessor directs the output to `stdout` when no `-o` option is specified.

-stringize

Enables the preprocessor stringization operator. By default, this switch is off. When set, this switch turns on the preprocessor stringization functionality (see "# (Argument)" on page 2-27) which is by default turned 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
```

-v[erbose]

Directs the preprocessor to output the version of the preprocessor program and information for each phase of the preprocessing.

-version

Directs the preprocessor to display the version information for the preprocessor program.

ⓘ The `-version` option on the assembler command line provides version information for both the assembler and preprocessor. The `-version` option on the preprocessor command-line provides preprocessor version information only.

I INDEX