VISUAL**DSP++**® 4.5
Assembler and Preprocessor Manual

Revision 2.0, April 2006

Part Number:
82-000420-04

**ANALOG
DEVICES**

CONTENTS

PREFACE

CONTENTS

CONTENTS

CONTENTS

PREPROCESSOR

CONTENTS

INDEX

VisualDSP++ 4.5 Assembler and Preprocessor Manual

PREFACE

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

Purpose

The *VisualDSP++ 4.5 Assembler and Preprocessor Manual* contains information about the assembler preprocessor utilties for the following Analog Devices, Inc. processor families—SHARC® (ADSP-21xxx) processors, TigerSHARC® (ADSP-TSxxx) processors, and Blackfin® (ADSP-BFxxx) processors.

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

Manual Contents

The manual consists of:

- 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

The *VisualDSP++ 4.5 Assembler and Preprocessor Manual* is a new manual that documents assembler support for all currently available Analog Devices' SHARC, TigerSHARC and Blackfin processors listed in "Supported Processors".

Refer to *VisualDSP++ 4.5 Product Release Bulletin* for information on all new and updated VisualDSP++® 4.5 features and other release information.

Technical or Customer Support

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

- Visit the Embedded Processing and DSP products Web site at
  http://www.analog.com/processors/technicalSupport

- E-mail tools questions to
  processor.tools.support@analog.com

- E-mail processor questions to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)

- Phone questions to **1-800-ANALOGD**

- Contact your Analog Devices, Inc. local sales office or authorized distributor

- Send questions by mail to:

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

Supported Processors

The following is the list of Analog Devices, Inc. processors supported in VisualDSP++ 4.5.

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

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

SHARC (ADSP-21xxx) Processors
The name "SHARC" refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC processors:

ADSP-21020	ADSP-21060	ADSP-21061	ADSP-21062
ADSP-21065L	ADSP-21160	ADSP-21161	ADSP-21261
ADSP-21262	ADSP-21266	ADSP-21267	ADSP-21363
ADSP-21364	ADSP-21365	ADSP-21366	ADSP-21367
ADSP-21368	ADSP-21369	ADSP-21371	ADSP-21375

Blackfin (ADSP-BFxxx) Processors

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

ADSP-BF531	ADSP-BF532	AD6901
ADSP-BF533	ADSP-BF534	AD6902
ADSP-BF535	ADSP-BF536	ADSP-BF541
ADSP-BF537	ADSP-BF538	ADSP-BF542
ADSP-BF539	ADSP-BF561	ADSP-BF544
AD6903	AD6531	ADSP-BF549
AD6901	AD6902	

Product Information

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

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

MyAnalog.com

`MyAnalog.com` is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests. `MyAnalog.com` provides access to books, application notes, data sheets, code examples, and more.

Registration

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

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

Processor Product Information

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

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

- E-mail questions or requests for information to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`

- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49-89-76903-157 (Europe)

- Access the FTP Web site at
  `ftp ftp.analog.com` (or `ftp 137.71.25.69`)
  `ftp://ftp.analog.com`

Related Documents

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

- *VisualDSP++ 4.5 Getting Started Guide*

- *VisualDSP++ 4.5 User's Guide*

- *VisualDSP++ 4.5 C/C++ Compiler and Library Manual for SHARC Processors*

- *VisualDSP++ 4.5 C/C++ Compiler and Library Manual for TigerSHARC Processors*

- *VisualDSP++ 4.5 C/C++ Compiler and Library Manual for Blackfin Processors*

- *VisualDSP++ 4.5 Assembler and Preprocessor Manual*

- *VisualDSP++ 4.5 Linker and Utilities Manual*

- *VisualDSP++ 4.5 Loader Manual*

- *VisualDSP++ 4.5 Product Release Bulletin*

- *VisualDSP++ Kernel (VDK) User's Guide*

- *Device Drivers and System Services Manual for Blackfin Processors*

- *Quick Installation Reference Card*

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

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

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

Online Technical Documentation

Online documentation includes the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest using the Search function of VisualDSP++ Help system. For easy printing, supplementary `.PDF` files of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
`.CHM`	Help system files and manuals in Help format
`.HTM` or `.HTML`	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the `.HTML` files requires a browser, such as Internet Explorer 5.01 (or higher).
`.PDF`	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the `.PDF` files requires a PDF reader, such as Adobe Acrobat Reader (4.5 or higher).

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

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.

- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

Accessing Documentation From Windows

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

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

Using Windows Explorer

- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.

- Open your VisualDSP++ installation CD-ROM and double-click any file that is part of the VisualDSP++ documentation set.

Using the Windows Start Button

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

Accessing Documentation From the Web

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

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

Printed Manuals

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

VisualDSP++ Documentation Set

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

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

Hardware Tools Manuals

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

Processor Manuals

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

Data Sheets

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

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

Notation Conventions

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

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

Example	Description
Close command (**File** menu)	Titles in in bold style reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
{this | that}	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as this or that. One or the other is required.
[this | that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional this or that.
[this,…]	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of this.
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
filename	Non-keyword placeholders appear in text with italic style format.

Notation Conventions

Example	Description
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

VisualDSP++ 4.5 Assembler and Preprocessor Manual

1 ASSEMBLER

This chapter provides information on how to use the assembler for developing and assembling programs with SHARC (ADSP-21xxx) processors, TigerSHARC (ADSP-TSxxx) processors, and Blackfin (ADSP-BFxxx) processors.

The chapter contains:

- "Assembler Guide" on page 1-2
  Describes the process of developing new programs in the processor's assembly language

- "Assembler Syntax Reference" on page 1-33
  Provides the assembler rules and conventions of syntax which are used to define symbols (identifiers), expressions, and to describe different numeric and comment formats

- "Assembler Command-Line Reference" on page 1-129
  Provides reference information on the assembler's switches and conventions

ⓘ The code examples in this manual have been compiled using VisualDSP++ 4.5. The examples compiled with other versions of VisualDSP++ may result in build errors or different output although the highlighted algorithms stand and should continue to stand in future releases of VisualDSP++.

Assembler Guide

In VisualDSP++ 4.5, the assembler drivers for each processor family run 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.

VisualDSP++ 4.5 supports the following assembler drivers:

- For SHARC processors – `easm21k.exe` assembler driver

- For TigerSHARC processors – `easmts.exe` assembler driver

- For Blackfin processors – `easmblkfn.exe` assembler driver

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

Software developers using the assembler should be familiar with:

- "Writing Assembly Programs" on page 1-3

- "Using Assembler Support for C Structs" on page 1-21

- "Preprocessing a Program" on page 1-24

- "Using Assembler Feature Macros" on page 1-26

- "Make Dependencies" on page 1-30

- "Reading a Listing File" on page 1-31

- "Statistical Profiling for Assembly Functions" on page 1-31

- "Specifying Assembler Options in VisualDSP++" on page 1-151

For information about the processor architecture, including the 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 1-129.

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 1-151).

Writing Assembly Programs

Assembler directives are coded in assembly source files. 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 on page 1-4 shows a graphical overview of the assembly process. The figure shows the preprocessor processing the assembly source (.ASM) and header (.H) files.
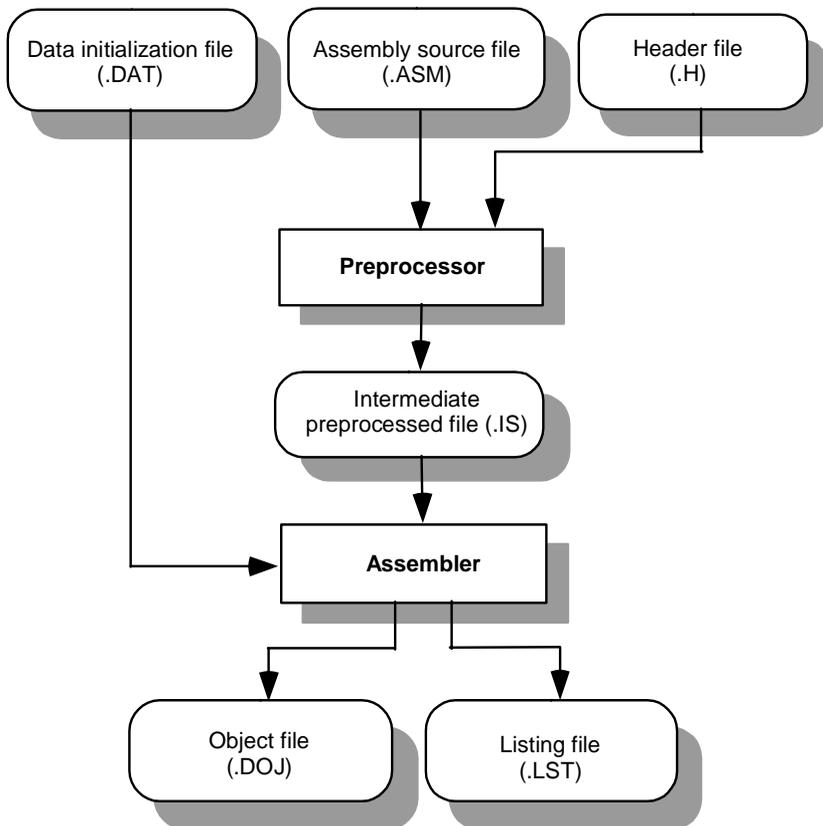
Figure 1-1. Assembler Input and Output Files

Assemble your source files from the VisualDSP++ environment or using any mechanism, such as a batch file or makefile, that will support invoking an appropriate assembler driver 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 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++ 4.5 Linker and Utilities Manual* .

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

The assembler listing 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. The listing file tells which imports were used within the program, followed by a more detailed section. (See the .import directive on page 1-84.) The file shows the name, total size and layout with offset for the members. The information appears at the end of the listing. You must specify the -l listname switch (on page 1-143) to get a listing file.

The assembly source file may contain preprocessor commands, such as #include, that cause the preprocessor to include header files (.H) into the source program. The preprocessor's only output, an intermediate source file (.IS), is the assembler's primary input. In normal operation, the preprocessor output is a temporary file that 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 processor's instruction set syntax documented in the processor's instruction set manual. Each instruction line must be terminated by a semicolon (;). On TigerSHARC processors, each instruction line (which can contain up to 4 instructions) is terminated by an additional semicolon (;;).Figure 1-2 on page 1-10 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 can then refer to this memory location using the label instead of an address. The assembler places no restriction on the number of characters in a label.

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

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

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

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

For example (in Blackfin processors),

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

For a complete description of the assembler's directive set, see "Assembler Directives" on page 1-63.

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-Line Reference" on page 2-36.

Program Structure

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

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

Table 1-1, Table 1-2, and Table 1-3 show suggested input section names for data and code that you could use in your assembly source for various processors. Using these predefined names in your sources makes it easier to take advantage of the default .LDF file included in your DSP system. However, you may also define your own sections. For detailed information on the .LDF files, refer to the *VisualDSP++ 4.5 Linker and Utilities Manual.*.

Table 1-1. Suggested Input Section Names for a SHARC LDF

.SECTION Name	Description
seg_pmco	A section in Program Memory that holds code
seg_dmda	A section in Data Memory that holds data
seg_pmda	A section in Program Memory that holds data
seg_rth	A section in Program Memory that holds system initialization code and interrupt service routines

Table 1-2. Suggested Input Section Names for a TigerSHARC LDF

.SECTION Name	Description
data1	A section that holds data in Memory Block M1.
data2	A section that holds data in Memory Block M2 (specified with the PM memory qualifier).
program	A section that holds code.

Table 1-3. Suggested Input Section Names for a Blackfin LDF

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

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.

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

Use sections in a program to group elements to meet hardware constraints. The example assembly program defines three sections. Each section begins with a `.SECTION` directive and ends with the occurrence of the next `.SECTION` directive or end-of-file.

Table 1-4 lists the following sections in the source program:

Table 1-4. Sections in Source Program

Section	SHARC	TigerSHARC	Blackfin
Data Section Variables and buffers are declared and can be initialized	`seg_dmda`	`data1` `data2`	`data1` `constdata`
Program Section Data, instructions, and possibly other types of statements are in this section, including statements that are needed for conditional assembly	`seg_pmco`	`program`	`seg_rth` `program`

Figure 1-2, Figure 1-3 on page 1-13, and Figure 1-4 on page 1-17 describe assembly code file structure for each of processor families. They shows how a program divides into sections that match the memory segmentation of a DSP system. Notice that an assembly source may contain preprocessor commands, such as `#include` to include other files in your source code, `#ifdef` for conditional assembly, or `#define` to define macros. The `SECTIONS{}` commands define the `.SECTION` placements in the system's physical memory as defined by the linker's `MEMORY{}` command. Assembler directives, such as `.VAR` (or `.BYTE` for Blackfin processors), appear within sections to declare and initialize variables.

Code File Structure and LDF for SHARC Processors

Figure 1-2 describes assembly code file structure for SHARC processors.

```
Preprocessor  ───────────►   #include "const.h"
commands                     #define PI 3.14159

Assembler  ──────────────►
directives                   .PRECISION=40;
                             .ROUND_ZERO:

Data section  ───────────►   .SECTION/DM seg_dmda;
                             .VAR fxd[10] = 1.2.3.4.5.6.7.8.9.0xA:
                             .VAR flt[5] = PI,PI/2,1.0,2.0,2.0/3.0;
                             .VAR rad:
                                 /* instructions */

Code section  ───────────►   .SECTION/PM seg_pmco;
                                 /* instructions */
Preprocessor  ───────────►   #ifdef some_variable
commands for                     /* instructions */
conditional                  #endif
assembly
```
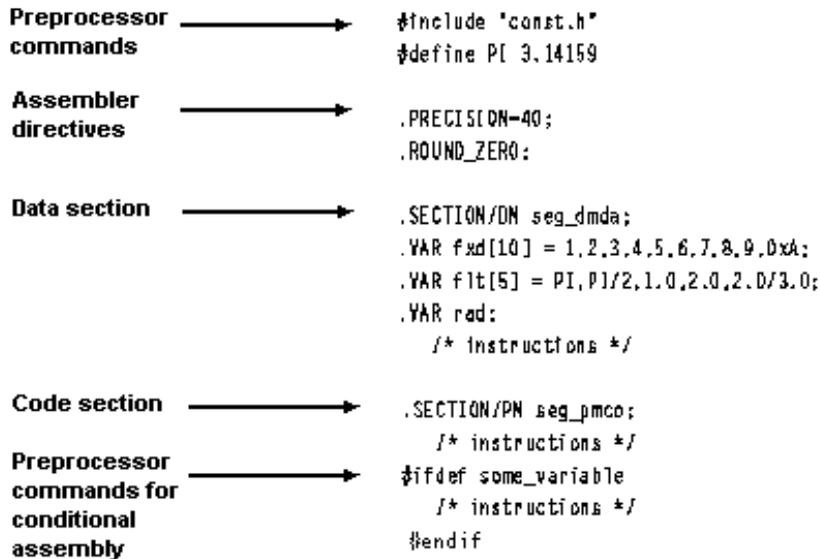
Figure 1-2. Assembly Code File Structure for SHARC Processors

Looking at Figure 1-2, notice that the .PRECISION and .ROUND_ZERO directives tell the assembler to store floating-point data with 40-bit precision and to round a floating-point value to a closer-to-zero value if it does not fit in the 40-bit format.

Listing 1-1 shows a sample user-defined LDF for SHARC Processors. Looking at the LDF's SECTIONS{} command, notice that the INPUT_SECTION commands map to the memory sections' names (such as program, data1, data2, ctor, heaptab, etc.) used in the example assembly sample program.

Listing 1-1. LDF Example for SHARC Processors

```
ARCHITECTURE(ADSP-21062)
SEARCH_DIR( $ADI_DSP\21k\lib )
$LIBRARIES = lib060.dlb, libc.dlb;
$OBJECTS = $COMMAND_LINE_OBJECTS, 060_hdr.doj;

MEMORY {
  seg_rth {TYPE(PM RAM) START(0x20000) END(0x20fff) WIDTH(48)}
  seg_init{TYPE(PM RAM) START(0x21000) END(0x2100f) WIDTH(48)}
  seg_pmco{TYPE(PM RAM) START(0x21010) END(0x24fff) WIDTH(48)}
  seg_pmda{TYPE(DM RAM) START(0x28000) END(0x28fff) WIDTH(32)}
  seg_dmda{TYPE(DM RAM) START(0x29000) END(0x29fff) WIDTH(32)}
  seg_stak{TYPE(DM RAM) START(0x2e000) END(0x2ffff) WIDTH(32)}
          /* memory declarations for default heap */
  seg_heap{TYPE(DM RAM) START(0x2a000) END(0x2bfff) WIDTH(32)}
          /* memory declarations for custom heap */
  seg_heaq{TYPE(DM RAM) START(0x2c000) END(0x2dfff) WIDTH(32)}
  }        // End MEMORY

PROCESSOR p0 {
  LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

  SECTIONS {
    .seg_rth {
      INPUT_SECTIONS( $OBJECTS(seg_rth) $LIBRARIES(seg_rth))
    } > seg_rth
    .seg_init {
      INPUT_SECTIONS( $OBJECTS(seg_init) $LIBRARIES(seg_init))
    } > seg_init
    .seg_pmco {
      INPUT_SECTIONS( $OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
    } > seg_pmco
    .seg_pmda {
      INPUT_SECTIONS( $OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))
    } > seg_pmda
    .seg_dmda {
      INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
    } > seg_dmda
```

```
.stackseg {
   ldf_stack_space = .;
   ldf_stack_length = 0x2000;
} > seg_stak

/* section placement for default heap */
.heap {
   ldf_heap_space = .;
   ldf_heap_end = ldf_heap_space + 0x2000;
   ldf_heap_length = ldf_heap_end - ldf_heap_space;
} > seg_heap

/* section placement for additional custom heap */
.heaq {
   ldf_heaq_space = .;
   ldf_heaq_end = ldf_heaq_space + 0x2000;
   ldf_heaq_length = ldf_heaq_end - ldf_heaq_space;
} > seg_heaq
} // End SECTIONS
} // End P0
```

Code File Structure and LDF for TigerSHARC Processors

Figure 1-3 describes assembly code file structure for TigerSHARC processors. Looking at Figure 1-3, 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 .VAR, appear within sections to declare and initialize variables.

```
Data  Section    ───────►   .SECTION data1;
   Assembler     ───────►   .VAR    buffer1[0x100] = 'buffer.dat';
   Directive
Data  Section    ───────►   .SECTION data2;
                            .VAR    buffer2;

Code  Section    ───────►   .SECTION program;
   Assembler     ───────►   start:
   Label
Preprocessor     ───────►   #ifndef XR0_SET_TO_2
Commands   for              XR0 = 0x00000001;;
Conditional                 #endif
Assembly
                            #ifdef XR0_SET_TO_2
                            XR0 = 0x00000002;;
                            #endif

Assembly         ───────►   J1 = buffer1;;
Instructions                JL1 = 0;;
                            J2 = 1;;
                            LC0 = 0x100;;

                            this_loop: [J1+=J2]=XR0;;
                            IF NLC0E, JUMP this_loop;;
```

Figure 1-3. Assembly Code File Structure for TigerSHARC Processors

---

Listing 1-2 shows a sample user-defined LDF for TigerSHARC processors. Looking at the LDF's SECTIONS{} command, notice that the INPUT_SECTION commands map to the memory sections' names (such as program, data1, data2, ctor, heaptab, etc.) used in the example assembly sample program.

Listing 1-2. Example Linker Description File for TigerSHARC Processors

```
ARCHITECTURE(ADSP-TS101)
SEARCH_DIR( $ADI_DSP\TS\lib )
$OBJECTS = $COMMAND_LINE_OBJECTS;

// Internal memory blocks are 0x10000 (64k)

MEMORY
{
  M0Code  { TYPE(RAM) START(0x00000000) END(0x0000FFFF) WIDTH(32) }
  M1Data  { TYPE(RAM) START(0x00080000) END(0x0008BFFF) WIDTH(32) }
  M1Heap  { TYPE(RAM) START(0x0008C000) END(0x0008C7FF) WIDTH(32) }
  M1Stack { TYPE(RAM) START(0x0008C800) END(0x0008FFFF) WIDTH(32) }
  M2Data  { TYPE(RAM) START(0x00100000) END(0x0010BFFF) WIDTH(32) }
  M2Stack { TYPE(RAM) START(0x0010C000) END(0x0010FFFF) WIDTH(32) }
  SDRAM   { TYPE(RAM) START(0x04000000) END(0x07FFFFFF) WIDTH(32) }
  MS0     { TYPE(RAM) START(0x08000000) END(0x0BFFFFFF) WIDTH(32) }
  MS1     { TYPE(RAM) START(0x0C000000) END(0x0FFFFFFF) WIDTH(32) }
}

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

   SECTIONS
   {       /* List of sections for processor P0 */
     code
     {
         FILL(0xb3c00000)
         INPUT_SECTION_ALIGN(4)
         INPUT_SECTIONS( $OBJECTS(program) )
     } >M0Code
```

```
data1
{
    INPUT_SECTIONS( $OBJECTS(data1) )
} >M1Data

data2
{
    INPUT_SECTIONS( $OBJECTS(data2) )
} >M2Data

// Provide support for initialization, including C++ static
// initialization. This section builds a table of
// initialization function pointers.
ctor
{
    INPUT_SECTIONS( $OBJECTS(ctor0) )
    INPUT_SECTIONS( $OBJECTS(ctor1) )
    INPUT_SECTIONS( $OBJECTS(ctor2) )
    INPUT_SECTIONS( $OBJECTS(ctor3) )
    INPUT_SECTIONS( $OBJECTS(ctor) )
} >M1Data

// Table containing heap segment descriptors
heaptab
{
    INPUT_SECTIONS( $OBJECTS(heaptab) )
} >M1Data

// Allocate stacks for the application.
jstackseg
{
    ldf_jstack_limit = .;
    ldf_jstack_base = . + MEMORY_SIZEOF(M1Stack);
} >M1Stack

kstackseg
{
    ldf_kstack_limit = .;
    ldf_kstack_base = . + MEMORY_SIZEOF(M2Stack);
```

```
    } >M2Stack

    // The default heap occupies its own memory block.
    defheapseg
    {
        ldf_defheap_base = .;
        ldf_defheap_size = MEMORY_SIZEOF(M1Heap);
    } >M1Heap
  }
}
```

Code File Structure and LDF for Blackfin Processors

Figure 1-4 describes the Blackfin processor's assembly code file structure and shows how a program divides into sections that match the memory segmentation of 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.

Listing 1-3 on page 1-18 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`.

```
Data Section    ───────►   .SECTION/DATA int_dm1;
                           .VAR     buffer1[0x100] = 'text2.txt';

Data Section    ───────►   .SECTION/DATA dummy;
                           .VAR  buffer2[0x100];

Data Section    ───────►   .SECTION/DATA int_dm3;
                           .VAR     buffer3;

Code Section    ───────►   .SECTION/PM seg_1;
Assembler       ───────►   .VAR/INIT24   pm_buffer1 = 0x123456;
Directive
Code Section    ───────►   .SECTION/CODE seg_rth;
Assembler       ───────►   JUMP start; RTI;RTI;RTI; // begin execution
Label and                          RTI;RTI;RTI;RTI;
Instructions                       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;

Code Section    ───────►   .SECTION/CODE kernel;
Assembler       ───────►   start:
Label
Preprocessor    ───────►   #ifndef AR_SET_TO_2
Commands for               AR = 0x0001;
Conditional                #endif
Assembly
                ───────►   #ifdef AR_SET_TO_2
                           AR = 0x0002;
                           #endif
Assembly        ───────►   I1 = buffer1;
Instructions               L1 = 0;
                           M2 = 1;

                           CNTR = 0x100;
                           DO this_loop UNTIL CE;
                           this_loop:  DM(I1,M2) = AR;
```

Figure 1-4. Assembly Source File Structure for Blackfin Processors

Listing 1-3. Example Linker Description File for Blackfin Processors

```
ARCHITECTURE(ADSP-BF535)
SEARCH_DIR($ADI_DSP\Blackfin\lib)
#define LIBS libc.dlb, libdsp.dlb
$LIBRARIES = LIBS, librt535.dlb;
$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++ 4.5 C/C++ Compiler and Library Manual* for appropriate target 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 processor. These rules for compiled code define the compiler's run-time environment. Complying with a run-time environment means following rules for memory usage, register usage, and variable names.

The definition of the run-time environment for the C/C++ compiler is provided in the *VisualDSP++ 4.5 C/C++ Compiler and Library Manual* for appropriate target 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 1-84)
  `.EXTERN STRUCT` (see on page 1-78)
  `.STRUCT` (see on page 1-118)

- **C Struct in Assembly Built-ins:**
  `OFFSETOF(struct/typedef,field)` (see on page 1-59)
  `SIZEOF(struct/typedef)` (see on page 1-59)

- **Struct References:**
  `struct->field` (nesting supported) (see "Struct References" on page 1-60)

For more information on C struct support, refer to the "-flags-compiler" command-line switch on page 1-139 and to "Reading a Listing File" on page 1-31.

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

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

- In the TigerSHARC assembler, `J1`, `J2` and `J3` use reserved keywords , but it is legal to reference them in the context of the C struct in assembly features.

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

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

Blackfin Example

```
.IMPORT "Coordinate.h";
      // typedef struct Coordinate {
      //    int     X;
      //    int     Y;
      //    int     Z;
      // } Coordinate;

.SECTION data1;

.STRUCT Coordinate Coord1 = {
      X = 1,
      Y = 4,
      Z = 7
      };

.SECTION program;

      P0.l = Coord1->X;
      P0.h = Coord1->X;

      P1.l = Coord1->Y;
      P1.h = Coord1->Y;

      P2.l = Coord1->Z;
      P2.h = Coord1->Z;

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

1-22   VisualDSP++ 4.5 Assembler and Preprocessor Manual

SHARC Example

```
.IMPORT "Samples.h";
     //   typedef struct Samples {
     //       int I1;
     //       int I2;
     //       int I3;
     //   }Samples;

.SECTION/DM seg_dmda;

.STRUCT Samples Sample1 ={
     I1 = 0x1000,
     I2 = 0x2000,
     I3 = 0x3000
     };

.SECTION/PM seg_pmco;
     doubleMe:
     // The code may look confusing, but I2 can be used both
     // as a register and a struct member name
     B2 = Sample1;
     M2 = OFFSETOF(Sample1,I2);
     R0 = DM(M2,I2);
     R0 = R0+R0;
     DM(M2,I2) = R0;
```

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

Preprocessing a Program

The assembler includes a preprocessor that allows the use of C-style pre-processor commands in your assembly source files. The preprocessor automatically runs before the assembler unless you use the assembler's -sp (skip preprocessor) switch. Table 2-5 on page 2-14 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 1-149) to the assembler command line or by selecting **Generate 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 1-151.

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 2-14. For more information on preprocessor usage, see "-flags-pp -opt1 [,-opt2...]" on page 1-140

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 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.

Table 1-5 provides the set of feature macros for SHARC processors.

Table 1-5. Feature Macros for SHARC Processors

`-D_LANGUAGE_ASM=1`	Always present
`-D__ADSP21000__=1`	Always present
`-D__ADSP21020__=1` `-D__2102x__=1`	Present when running `easm21K -proc ADSP-21020` with ADSP-21020 processors
`-D__ADSP21060__=1` `-D__2106x__=1`	Present when running `easm21K -proc ADSP-21060` with ADSP-21060 processors
`-D__ADSP21061__=1` `-D__2106x__=1`	Present when running `easm21K -proc ADSP-21061` with ADSP-21061 processors
`-D__ADSP21062__=1` `-D__2106x__=1`	Present when running `easm21K -proc ADSP-21062` with ADSP-21062 processors
`-D__ADSP21065L__=1` `-D__2106x__=1`	Present when running `easm21K -proc ADSP-21065L` with ADSP-21065L processors
`-D__ADSP21160__=1` `-D__2116x__=1`	Present when running `easm21K -proc ADSP-21160` with ADSP-21160 processors
`-D__ADSP21161__=1` `-D__2116x__=1`	Present when running `easm21K -proc ADSP-21161` with ADSP-21161 processors
`-D__ADSP21261__=1` `-D__2126x__=1`	Present when running `easm21K -proc ADSP-21261` with ADSP-21261 processors
`-D__ADSP21262__=1` `-D__2126x__=1`	Present when running `easm21K -proc ADSP-21262` with ADSP-21262 processors

Table 1-5. Feature Macros for SHARC Processors (Cont'd)

`-D__ADSP21266__=1` `-D__2126x__=1`	Present when running `easm21K -proc ADSP-21266` with ADSP-21266 processors
`-D__ADSP21267__=1` `-D__2126x__=1`	Present when running `easm21K -proc ADSP-21267` with ADSP-21267 processors
`-D__ADSP21363__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21363` with ADSP-21363 processors
`-D__ADSP21364__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21364` with ADSP-21364 processors
`-D__ADSP21365__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21365` with ADSP-21365 processors
`-D__ADSP21366__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21366` with ADSP-21366 processors
`-D__ADSP21367__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21367` with ADSP-21367 processors
`-D__ADSP21368__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21368` with ADSP-21368 processors
`-D__ADSP21369__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21369` with ADSP-21369 processors
`-D__ADSP2137x__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-2137x` with ADSP-2137x processors
`-D__ADSP21371__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21371` with ADSP-21371 processors
`-D__ADSP21375__=1` `-D__2136x__=1`	Present when running `easm21K -proc ADSP-21375` with ADSP-21375 processors

Table 1-6 provides the set of feature macros for TigerSHARC processors.

Table 1-6. Feature Macros for TigerSHARC Processors

`-D_LANGUAGE_ASM =1`	Always present
`-D__ADSPTS__ =1`	Always present

Table 1-6. Feature Macros for TigerSHARC Processors (Cont'd)

`-D__ADSPTS101__ =1`	Present when running `easmts -proc ADSP-TS101` with ADSP-TS101 processor
`-D__ADSPTS201__ =1`	Present when running `easmts -proc ADSP-TS201` with ADSP-TS201 processor
`-D__ADSPTS202__ =1`	Present when running `easmts -proc ADSP-TS202` with ADSP-TS202 processor
`-D__ADSPTS203__ =1`	Present when running `easmts -proc ADSP-TS203` with ADSP-TS203 processor
`-D__ADSPTS20x__ =1`	Present when running `easmts -proc ADSP-TS201` with ADSP-TS201 processor, `easmts -proc ADSP-TS202` with ADSP-TS202 processor, or `easmts -proc ADSP-TS203` with ADSP-TS203 processor.

Table 1-7 provides the set of feature macros for Blackfin processors.

Table 1-7. Feature Macros for Blackfin Processors

`-D_LANGUAGE_ASM=1`	Always present
`-D__ADSPBLACKFIN__ =1`	Always present
`-D__ADSPLPBLACKFIN__ =1`	Always present for non-ADSP-BF535 processors
`-D__ADSPBF531__=1` `-D__ADSP21531__=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__ADSPBF534__=1`	Present when running `easmblkfn -proc ADSP-BF534` with ADSP-BF534 processor.
`-D__ADSPBF535__=1` `-D__ADSP21535__=1`	Present when running `easmblkfn -proc ADSP-BF535` with ADSP-BF535 processor.
`-D__ADSPBF536__=1`	Present when running `easmblkfn -proc ADSP-BF536` with ADSP-BF536 processor.

Table 1-7. Feature Macros for Blackfin Processors (Cont'd)

`-D__ADSPBF537__=1`	Present when running `easmblkfn -proc ADSP-BF537` with ADSP-BF537 processor.
`-D__ADSPBF538__=1`	Present when running `easmblkfn -proc ADSP-BF538` with ADSP-BF538 processor.
`-D__ADSPBF539__=1`	Present when running `easmblkfn -proc ADSP-BF539` with ADSP-BF539 processor.
`-D__ADSPBF561__=1`	Present when running `easmblkfn -proc ADSP-BF561` with ADSP-BF561 processor.
`-D__AD6531__=1`	Present when running `easmblkfn -proc AD6531` with AD6531 processor.
`-D__AD6532__=1`	Present when running `easmblkfn -proc AD6532` with AD6532 processor.
`-D__AD6900__=1`	Present when running `easmblkfn -proc AD6900` with AD6900 processor.
`-D__AD6901__=1`	Present when running `easmblkfn -proc AD6901` with AD6901 processor.
`-D__AD6902__=1`	Present when running `easmblkfn -proc AD6902` with AD6902 processor.
`-D__AD6903__=1`	Present when running `easmblkfn -proc AD6903` with AD6903 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 example,

```
easm21k -proc adsp-21262 assembly    --> cc21K -proc adsp-21262
easmts -proc -ADSP-TS101 assembly     --> ccts -proc ADSP-TS101
easmblkfn -proc ADSP-BF535 assembly       --> ccblkfn -proc
ADSP-BF535
```

> ℹ Use the `-verbose` option to verify what macro is default-defined.
> Refer to Chapter 1 in the *VisualDSP++ 4.5 C/C++ Compiler and
> Library Manual* of the appropriate target processors for more
> information.

Make Dependencies

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

When the user requests make dependencies for the assembly, the assem-
bler produces the dependencies from buffer initializations. The assembler
also invokes the preprocessor to determine the make dependency from
`#include` commands, and the compiler to determine the make dependen-
cies from the `.IMPORT` headers.

For example,

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

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

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

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

Reading a Listing File

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

- Address – The first column contains the offset from the `.SECTION`'s base address.

- Opcode – The second column contains the hexadecimal opcode that the assembler generates for the line of assembly source.

- Line – The third column contains the line number in the assembly source file.

- Assembly Source – The fourth column contains the assembly source line from the file.

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

Statistical Profiling for Assembly Functions

Use the following steps to enable the Statistical Profiling in assembler sources.

1. When using the VisualDSP++ IDDE, use the **Assemble** option from the **Project Options** dialog box (Figure 1-8) to select and/or set assembler functional options.

2. Select **Assemble - Generate Debug Information**.

3. Mark ending function boundaries with `.end` labels in the assembler source. For example:

```
.SECTION program;

.GLOBAL funk1;
funk1:
    ...
    rts;
funk1.end:

.GLOBAL funk2;
funk2:
    ...
    rts;
funk2.end:
```

If you have global functions without ending labels, the assembler provides warnings when debug information is generated.

```
.GLOBAL funk3;
funk3:
    ...
    rts;
```

```
[Warning ea1121] "test.asm":14 funk3: -g assembly with
global function without ending label. Use 'funk3.end' or
'funk3.END' to mark the ending boundary of the function for
debugging information for automated statistical profiling
of assembly functions.
```

4. Add ending labels or selectively disable the warning by adding the `-Wsuppress 1121` option to the **Assembler Additional Options** field in the **Assembly** tab (refer to "WARNING ea1121: Missing End Labels" on page 1-141 for more information).

5. Select **Statistical Profiling - New Profile** or **Linear Profiling - New Profile** options as appropriate. Assembler functions automatically appear in the profiling window along with C functions. Click on the function name to bring up the source containing the function definition.

Assembler Syntax Reference

When you develop a source program in assembly language, include pre-processor 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) and expressions, and to use different numeric and comment formats.

Software developers who write assembly programs should be familiar with:

- "Assembler Keywords and Symbols" on page 1-34

- "Assembler Expressions" on page 1-47

- "Assembler Operators" on page 1-48

- "Numeric Formats" on page 1-52

- "Comment Conventions" on page 1-55

- "Conditional Assembly Directives" on page 1-55

- "C Struct Support in Assembly Built-In Functions" on page 1-59

- "Struct References" on page 1-60

- "Assembler Directives" on page 1-63

Assembler Keywords and Symbols

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

Table 1-8 lists the assembler keywords for SHARC processors.

Table 1-8. SHARC Processor Assembler Keywords

__ADI__	__DATE__	__FILE__	__LastSuffix__	__LINE__
__TIME__				
.ALIGN	.ELIF	.ELSE	.ENDIF	.EXTERN
.FILE	.FILE_ATTR	.GLOBAL	.IF	.IMPORT
.LEFTMARGIN	.LIST	.LIST_DATA	.LIST_DATFILE	.LIST_DEFTAB
.LIST_LOCTAB	.LIST_WRAPDATA	.NEWPAGE	.NOLIST_DATA	.NOLIST_DATFILE
.NOLIST_WRAPDATA	.PAGELENGTH	.PAGEWIDTH	.PRECISION	.ROUND_MINUS
.ROUND_NEAREST	.ROUND_PLUS	.ROUND_ZERO	.PREVIOUS	.SECTION
.STRUCT	.VAR	.WEAK		
ABS	ACS	ACT	ADDRESS	AND
ASHIFT	ASTAT	AV		
B0	B1	B2	B3	B4
B5	B6	B7	B8	B9
B10	B11	B12	B13	B14
B15	BB	BCLR	BF	BIT

Table 1-8. SHARC Processor Assembler Keywords (Cont'd)

BITREV	BM	BSET	BTGL	BTSTS
BY				
CA	CACHE	CALL	CH	CI
CJUMP	CL	CLIP	COMP	COPYSIGN
COS	CURLCNTR			
DADDR	DB	DEC	DEF	DIM
DMA1E	DMA1S	DMA2E	DMA2S	DMADR
DMABANK1	DMABANK2	DMABANK3	DMAWAIT	DO
DOVL				
EB	ECE	EF	ELSE	EMUCLK
EMUCLK2	EMUIDLE	EMUN	ENDEF	EOS
EQ	EX	EXP	EXP2	
F0	F1	F2	F3	F4
F5	F6	F7	F8	F9
F10	F11	F12	F13	F14
F15	FADDR	FDEP	FEXT	FILE
FIX	FLAG0_IN	FLAG1_IN	FLAG2_IN	FLAG3_IN
FLOAT	FLUSH	FMERG	FOREVER	FPACK
FRACTIONAL	FTA	FTB	FTC	FUNPACK
GCC_COMPILED	GE	GT		
I0	I1	I2	I3	I4
I5	I6	I7	I8	I9
I10	I11	I12	I13	I14
I15	IDLEI15	IDLE16	IF	IMASK
IMASKP	INC	IRPTL		
JUMP				
L0	L1	L2	L3	L4

Table 1-8. SHARC Processor Assembler Keywords (Cont'd)

L5	L6	L7	L8	L9
L10	L11	L12	L13	L14
L15	LA	LADDR	LCE	LCNTR
LE	LADDR	LCE	LCNTR	LE
L15	LA	LADDR	LCE	LCNTR
LE	LEFTO	LEFTZ	LENGTH	
LINE	LN	LOAD	LOG2	LOGB
LOOP	LR	LSHIFT	LT	
M0	M1	M2	M3	M4
M5	M6	M7	M8	M9
M10	M11	M12	M13	M14
M15	MANT	MAX	MBM	MIN
MOD	MODE1	MODE2	MODIFY	MROB
MROF	MR1B	MR1F	MR2B	MR2F
MRB	MRF	MS	MV	MROB
MROF				
NE	NOFO	NOFZ	NOP	NOPSPECIAL
NOT	NU			
OFFSETOF	OR			
P20	P32	P40	PACK	PAGE
PC	PCSTK	PCSTKP	PM	PMADR
PMBANK1	PMDAE	PMDAS	POP	POVLO
POVL1	PSA1E	PSA1S	PSA2E	PSA3E
PSA3S	PSA4E	PSA4S	PUSH	PX
PX1	PX2			
R0	R1	R2	R3	R4
RF5	R6	R7	R8	R9

Table 1-8. SHARC Processor Assembler Keywords (Cont'd)

R10	R11	R12	R13	R14
R15	READ	RECIPS	RFRAME	RND
ROT	RS	RSQRTS	RTI	RTS
SCALB	SCL	SE	SET	SF
SIN	SIZE	SIZEOF	SQR	SR
SSF	SSFR	SSI	SSIR	ST
STEP	STKY	STRUCT	STS	SUF
SUFR	SV	SZ		
TAG	TCOUNT	TF	TGL	TPERIOD
TRUE	TRUNC	TST	TYPE	TRAP
UF	UI	UNPACK	UNTIL	UR
USF	USFR	USI	USIR	USTAT1
USTAT2	UUF	UUFR	UUIR	UUIR
VAL				
WITH				
XOR				

Table 1-9 lists the assembler keywords for TigerSHARC processors.

Table 1-9. TigerSHARC Processor Assembler Keywords

__ADI__	__DATE__	__FILE__	__LastSuffix__	__LINE__
__TIME__				
.ALIGN	.ALIGN_CODE	.ELIF	.ELSE	.ENDIF
.EXTERN	.FILE	.FILE_ATTR	.GLOBAL	.IF
.IMPORT	.LEFTMARGIN	.LIST	.LIST_DATA	.LIST_DATFILE
.LIST_DEFTAB	.LIST_LOCTAB	.LIST_WRAPDATA	.MESSAGE	.NOLIST_DATA
.NOLIST_DATFILE	.NOLIST_WRAPDATA	.NEWPAGE	.NOLIST_DATA	.NOLIST_DATFILE
.NOLIST_WRAPDATA	.PAGELENGTH	.PAGEWIDTH	.PREVIOUS	.SECTION
.SEPARATE_MEM_SEGMENTS	.SET	.STRUCT	.VAR	.WEAK
ABS	ACS	ADDRESS	AND	ASHIFT
BCLR	BFOINC	BFOTMP	BITEST	BITFIFO
BKFPT	BR	BSET	BTBDIS	BTBELOCK
BTBEN	BTBLOCK	BTBINV	BTGL	BY
C	CALL	CB	CJMP	CJMP_CALL
CI	CLIP	COMP	COMPACT	COPYSIGN
DAB	DEC	DESPREAD	DO	
ELSE	EMUTRAP	EXP	EXPAND	EXTD
FCOMP	FDEP	FEXT	FIX	FLOAT
FTEST0	FTEST1	FOR		
GETBITS				
IDLE	INC			
JC	JUMP			
KC				

Table 1-9. TigerSHARC Processor Assembler Keywords (Cont'd)

LD0	LD1	LENGTH	LINES	LOGB
LP	LSHIFT	LSHIFTR	LIBSIM_CALL	
MANT	MASK	MAX	MERGE	MIN
NEWPAGE	NOT	NOP	NP	
OFFSETOF	ONES	OR		
PASS	PERMUTE	PRECISION	PUTBITS	
RDS	RECIPS	RESET	RETI	ROT
ROTL	ROTR	ROUND	RSQRTS	RTI
SCALB	SDAB	SE	SECTION	SF0
SF1	SNGL	SIZE	SIZEOF	STRUCT
SUM				
TMAX	TRAP	TYPEVAR		UNTIL
VMIN	VMAX			
XCORRS	XOR	XSDAB		
YDAB	YSDAB			
// JK Register Group				
J0 through J31				
K0 through K31				
JB0	JB1	JB2	JB3	
KB0	KB1	KB2	KB3	
JL0	JL1	JL2	JL3	
KL0	KL1	KL2	KL3	
// RF Register Group				
FR0 through FR31				
MR3:0	MR3:2	MR1:0		
MR0	MR1	MR2	MR3	MR4
PR0	PR1	PR1:0		

Table 1-9. TigerSHARC Processor Assembler Keywords (Cont'd)

R0 through R31				
XSTAT	YSTAT	XYSTAT		
XR0 through XR31				
YR0 through YR31				
// Accelerator Register Group				
TR0 through TR31				
THR0	THR1	THR2	THR3	
// EP Register Group				
BMAX	BMAXC	BUSLK	FLGPIN	FLGPINCL
FLGPINST	SDRCON	SYSCON	SYSCONCL	SYSCONST
SYSCTL	SYSTAT	SYSTATCL		
// Misc. Register Group				
AUTODMA0	AUTODMA1			
BTBCMD	BTBDATA			
BTB0TG0 through BTB0TG31				
BTB1TG0 through BTB1TG31				
BTB2TG0 through BTB2TG31				
BTB3TG0 through BTB3TG31				
BTB0TR0 through BTB0TR31				
BTB1TR0 through BTB1TR31				
BTB2TR0 through BTB2TR31				
BTB3TR0 through BTB3TR31				
BTBLRU0 through BTBLRU31				
CACMD0	CACMD2	CACMD4	CACMD8	CACMD10
CACMDALL				
CADATA0	CADATA2	CADATA4	CADATA8	CADATA10
CADATAALL				

Table 1-9. TigerSHARC Processor Assembler Keywords (Cont'd)

CASTAT0	CASTAT2	CASTAT4	CASTAT8	CASTAT10
CASTATALL				
CCAIR0	CCAIR2	CCAIR4	CCAIR8	CCAIR10
CCAIRALL				
CCNT0	CCNT1	CJMP	CMCTL	
DBGE	DC4 through DC13			
DCD0	DCD1	DCD2	DCD3	DCNT
DCNTCL	DCNTST			
DCS0	DCS1	DCS2	DCS3	
DSTAT	DSTATC			
EMUCTL	EMUDAT	EMUIR	EMUSTAT	
IDCODE	ILATCLH	ILATCLL	ILATH	ILATL
ILATSTH	ILATSTL	IMASKH	IMASKL	INSTAT
INTEN	INTCTL	IVBUSLK	IVDBG	IVHW
IVDMA0 through IVDMA13				
IVIRQ0	IVIRQ1	IVIRQ2	IVIRQ3	IVLINK0
IVLINK1	IVLINK2	IVLINK3	IVSW	IVTIMER0HP
IVTIMER0LP	IVTIMER1HP	IVTIMER1LP		
LBUFRX0	LBUFRX1	LBUFRX2	LBUFRX3	
LBUFTX0	LBUFTX1	LBUFTX2	LBUFTX3	
LC0	LC1	KB2	KB3	
LCTL0	LCTL1	LCTL2	LCTL3	
LRCTL0	LRCTL1	LRCTL2	LRCTL3	
LRSTAT0	LRSTAT1	LRSTAT2	LRSTAT3	
LRSTATC0	LRSTATC1	LRSTATC2	LRSTATC3	
LSTAT0	LSTAT1	LSTAT2	LSTAT3	
LSTATC0	LSTATC1	LSTATC2	LSTATC3	

Table 1-9. TigerSHARC Processor Assembler Keywords (Cont'd)

LTCTL0	LTCTL1	LTCTL2	LTCTL3	
LTSTAT0	LTSTAT1	LTSTAT2	LTSTAT3	
LTSTATC0	LTSTATC1	LTSTATC2	LTSTATC3	
MISR0	MISR1	MISR2	MISRCTL	
RETI	RETIB	RETS	RTI	
OSPID				
PMASKH	PMASKL	PRFM	PRFCNT	
SERIAL_H	SERIAL_L	SFREG	SQCTL	SQCTLST
SQCTLCL	SQSTAT			
TESTMODES	TIMER0L	TIMER1L	TIMER0H	TIMER1H
TMRIN0L	TMRIN0H	TMRIN1L	TMRIN1H	TRCB
TRCBMASK	TRCBPTR	TRCBVAL		
VIRPT				
WP0CTL	WP1CTL	WP2CTL	WP0STAT	WP1STAT
WP2STAT	W0H	W0L	W1H	W1L
W2H	W2L			
// Conditions which may be prefixed with X, Y, XY, NX, NY, XY				
AEQ	ALE	ALT	MEQ	MLE
MLT	SEQ	SF1	SF0	SLT
// Conditions which may be prefixed with J, K, NJ, NK				
EQ	LE	LT	CBQ	CB1
// Conditions which may be prefixed with N				
ISF0	ISF1	LC0E	LC1E	BM
FLAG0_IN	FLAG1_IN	FLAG2_IN	FLAG3_IN	

Table 1-10 lists the assembler keywords for Blackfin processors.

Table 1-10. Blackfin Processor Assembler Keywords

.ALIGN	.ASCII	.ASM_ASSERT	.ASSERT	.BSS
.BYTE	.BYTE2	.BYTE4	.DATA	.ELIF
.ELSE	.ENDIF	.ELSE	.ENDIF	.EXTERN
.FILE	.FILE_ATTR	.GLOBAL	.GLOBL	
.IF	.INC/BINARY	.INCBIN	.IMPORT	
.LEFTMARGIN	.LIST	.LIST_DATA	.LIST_DATFILE	.LIST_DEFTAB
.LIST_LOCTAB	.LIST_WRAPDATA	.LONG EXPRESSION-LIST		
.NEWPAGE	.NOLIST	.NOLIST_DATA	.NOLIST_DATFILE	.NOLIST_WRAPDATA
.PAGELENGTH	.PAGEWIDTH	.PREVIOUS	.SECTION	.SET SYMBOL .SYMBOL
.SHORT EXPRESSION-LIST	.STRUCT	.TEXT	.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
CALL	CARET	CC	CLI	CLIP
C0	CODE	COLON	COMMA	CSYNC

Table 1-10. Blackfin Processor Assembler Keywords (Cont'd)

DATA				
		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	L0	LOOP	LOOP_BEGIN	LOOP_END
LPAREN	LSETUP	LSHIFT	LT	LT0
LT1	LZ			
M	M0	M1	M2	M3
MAX	MIN	MINUS	MNOP	MUNOP
NEG	NO_INIT	NOP	NOT	NS
ONES	OR	OUTC		
P0	P1	P2	P3	P4
P5	PACK	PC	PRNT	PERCENT
PLUS	PREFETCH			
R	R0	R1	R2	R3
R32	R4	R5	R6	R7
RAISE	RBRACE	RBRACK	RETI	RETN
RETS	RETX	RND	RND12	RND20
RNDH	RNDL	ROL	ROR	ROT

Table 1-10. Blackfin Processor Assembler Keywords (Cont'd)

ROT_L_AC	ROT_R_AC	RPAREN	RSDL	RTE
RTI	RTN	RTS	RTX	RUNTIME_INIT
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
ZERO_INIT				
ADI	_DATE_	_FILE_	_LastSuffix_	_LINE_
TIME				

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. A symbol with a '.' as the first character cannot have a digit as the second character. Such symbols will not appear in the symbol table accessible in the debugger. A symbol name in which the first two characters are dots will not appear even in the symbol table of the object.

  The compiler and run times 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 file. The linker uses sections' name symbols to place code and data in processor's memory. For more details, see the *VisualDSP++ 4.5 Linker and Utilities Manual* .

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

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

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

Address label symbols may appear at the beginning of an instruction line or 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 - 0.8r
```

For information about fraction type support, refer to "Fractional Type Support" on page 1-53.

Symbolic Expressions

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

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

Symbols in this type of expression are data variables, data buffers, and program labels. In the first three examples above, the symbol name represents the address of the symbol. The fourth combines that meaning of a symbol with a use of the length operator (see Table 1-12).

Assembler Operators

Table 1-11 lists the assembler's numeric and bitwise operators used in constant expressions and address expressions. These operators are listed in group order from highest to lowest precedence. Operators with highest precedence are evaluated first. When two operators have the same precedence, the assembler evaluates the left-most operator first. Relational operators are only supported in relational expressions in conditional assembly, as described in "Conditional Assembly Directives" on page 1-55.

Table 1-11. 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)	

Table 1-11. Operator Precedence (Cont'd)

Operator	Usage Description	Designation
`|`	Bitwise inclusive OR	
`^`	Bitwise exclusive OR (preprocessor only)	
`&&`	Logical AND	
`||`	Logical OR	

The assembler also supports special operators. Table 1-12 lists and describes these operators used in constant and address expressions.

Table 1-12. Special Assembler Operators

Operator	Usage Description
`ADDRESS(`*`symbol`*`)`	Address of `symbol` **Note:** Used with SHARC and TigerSHARC assemblers only.
`BITPOS(`*`constant`*`)`	Bit position (Blackfin processors ONLY)
`HI(expression)` `LO(expression)`	Extracts the most significant 16 bits of expression. Extracts the least significant 16 bits of expression. **Note:** Used with the Blackfin assembler ONLY where `HI`/`LO` replaces the `ADRRESS()` operator. The expression in the "HI" and "LO" operators can be either symbolic or constant.
`LENGTH(symbol)`	Length of `symbol` in number of elements (in a buffer/array)
`symbol`	Address pointer to `symbol`

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.

Blackfin Processor Example

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

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

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

```
#define DLAB 0x80
#define EPS 0x10
r0 = DLAB | EPS (z);
cc = bitset (r0, BITPOS(DLAB));
```

TigerSHARC Processor Example

The following example demonstrates how the assembler operators are used to load the length and address information into registers (when setting up circular buffers in TigerSHARC processors).

```
.SECTION data1;         // Data segment
.VAR real_data[n];      // n = number of input samples
```

```
    …
    .SECTION program;      // Code segment
                           // Load the base address of the
                           // circular buffer:
    JB3 = real_data;;
                           // Load the index:
    J3=real_data;;
                           // Load the circular buffer length:
    JL3 = LENGTH(real_data);;
                           // Set loop counter 0 with buffer length:
    LC0 = JL3;;
    start:
    XR0 = CB [J3 += 1];;  // Read data from circular buffer
    if NLC0E, jump start;;
```

This code fragment initializes `JB3` and `JL3` to the base address and length, respectively, of the circular buffer `real_data`. The buffer length value contained in `JL3` determines when addressing wraps around the top of the buffer. For further information on circular buffers, refer to the hardware reference manual of the target processor.

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

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

    .SECTION/PM seg_pmco;      // code segment
      B5=real_data;            // buffer base address
                               // I5 loads automatically
      L5=length(real_data);    // buffer's length
      M6=1;                    // post-modify I5 by 1
      LCNTR=length(real_data)
```

```
        ,DO loopend UNTIL LCE;
                                 // loop counter=buffer's length
        FO=DM(I5,M6);            // get next sample
    …
    loopend:  …
```

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

Numeric Formats

Depending on the processor architectures, the assemblers support binary,
decimal, hexadecimal, floating-point, and fractional numeric formats
(bases) within expressions and assembly instructions. Table 1-13 describes
the conventions of notation the assembler uses to distinguish between
numeric formats.

Table 1-13. 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.number[e {+/-} number]	Entry for floating-point number
number	No prefix and no decimal point indicates a decimal number
*number*r	"r" suffix indicates a fractional number

> 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 2-32 and

the preprocessor's "-stringize" command-line switch
(on page 2-44), and to the assembler's "-flags-pp -opt1 [,-opt2...]"
command-line switch (on page 1-140).

Fractional Type Support

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

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

For example,

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

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

(i) In Blackfin processors, 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

Fracts supported by the Analog Devices' processors 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 processors to convert from the
floating-point to fixed-point format uses a scale factor of 31.

For example,

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

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

.VAR/R32 myFract = -1.72471041E-03r;
   // Fract output for -1.72471041E-03 is 0xFFC77C15
   // sign bit + 31 bits
   //  1111 1111 1100 0111 0111 1100 0001 0101
   //   F    F    C    7    7    C    1    5
```

1.0r Special Case

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

Fractional Arithmetic

The assembler provides support for arithmetic expressions using operations on fractional constants, consistent with the support for other numeric types in constant expressions, as described in "Assembler Expressions" on page 1-47.

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

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

```
                           // Result .6r is within the legal range
    .VAR abc = 1.5r;       // Error: 1.5r out of range
```

Mixed Type Arithmetic

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

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

Comment Conventions

The assemblers support C- and C++ -style formats for inserting comments in assembly sources. The assemblers do not support nested comments. Table 1-14 lists and describes assembler comment conventions.

Table 1-14. 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 structs in assembly built-in functions 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 1-15 shows examples of conditional directives.

Table 1-15. 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);

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`.

(i) 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-in functions. 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-in functions (see "C Struct Support in Assembly Built-In Functions" on page 1-59) 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 preprocessor. Therefore, preprocessor's #if directives cannot use the assembler built-ins (see "C Struct Support in Assembly Built-In Functions" on page 1-59).

- The conditional assembly directives are processed by the assembler in a later pass. Therefore, you are able to write a relational or logical expression whose value depends on the value of a #define. For example,

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

If you have "`#define tryit 2`", 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`, and so on.

C Struct Support in Assembly Built-In Functions

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

OFFSETOF() Built-In Function

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

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

where:

> struct/typedef – struct VAR or a typedef can be supplied as the first argument
>
> memberName – a member name within the struct or typedef (second argument)

For SHARC and TigerSHARC processors, OFFSETOF() units are in words. For Blackfin processors, OFFSETOF() units are in bytes.

SIZEOF() Built-In Function

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

```
SIZEOF(struct/typedef/C base type);
```

where:

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

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

- An aggregate type (structure)

- A C base type (`int`, `char`, and so on)

- A member of a structure (any type)

For example (Blackfin processor code),

```
.IMPORT "Celebrity.h";
.EXTERN STRUCT Celebrity StNick;
l3 = SIZEOF(Celebrity);    // typedef
l3 = SIZEOF(StNick);       // struct var of typedef Celebrity
l3 = SIZEOF(char);         // C built-in type
l3 = SIZEOF(StNick->Town);    // member of a struct var
l3 = SIZEOF(Celebrity->Town);  // member of a struct typedef
```

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

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

Struct References

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

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 recalculates the offset when the source is reassembled 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 (for Blackfin processors)

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

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

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

C Code

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

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

Assembly Code (for Blackfin processors)

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

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

Assembler Directives

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

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

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

Table 1-16 lists all currently supported assembler directives. A description of each directive appears in the following sections. These directives were added for GNU compatibility.

Table 1-16. Assembler Directive Summary

Directive	Description
`.ALIGN` (see on page 1-68)	Specifies an alignment requirement for data or code
`.ALIGN_CODE` (see on page 1-70)	Specifies an alignment requirement for code. **NOTE:** TigerSHARC processors ONLY.
`.ASCII` (see on page 1-72)	Initializes ASCII strings **NOTE:** Blackfin processors ONLY.
`.BYTE| .BYTE2| .BYTE4` (see on page 1-73)	Defines and initializes one-, two-, and four-byte data objects, respectively. **NOTE:** Blackfin processors ONLY.
`.ELSE` (see on page 1-55)	Conditional assembly directive
`.ENDIF` (see on page 1-55)	Conditional assembly directive

Table 1-16. Assembler Directive Summary (Cont'd)

Directive	Description
.ENDSEG (see on page 1-117)	Legacy directive. Marks the end of a section. Used with legacy directive .SEGMENT that begins a section. **NOTE:** SHARC processors ONLY.
.EXTERN (see on page 1-77)	Allows reference to a global symbol
.EXTERN STRUCT (see on page 1-78)	Allows reference to a global symbol (struct) that was defined in another file
.FILE (see on page 1-80)	Overrides filename given on the command line. Used by C compiler
.FILE_ATTR (see on page 1-81)	Creates a attribute in the generated object file.
.GLOBAL (see on page 1-82)	Changes a symbol's scope from local to global
.IF (see on page 1-55)	Conditional assembly directive
.IMPORT (see on page 1-84)	Provides the assembler with the structure layout (C struct) information
.INC/BINARY (see on page 1-86)	Includes the content of file at the current location. **NOTE:** Blackfin processors ONLY
.LEFTMARGIN (see on page 1-87)	Defines the width of the left margin of a listing
.LIST (see on page 1-88)	Starts listing of source lines
.LIST_DATA (see on page 1-89)	Starts listing of data opcodes
.LIST_DATFILE (see on page 1-90)	Starts listing of data initialization files
.LIST_DEFTAB (see on page 1-91)	Sets the default tab width for listings
.LIST_LOCTAB (see on page 1-92)	Sets the local tab width for listings

Table 1-16. Assembler Directive Summary (Cont'd)

Directive	Description
.LIST_WRAPDATA (see on page 1-93)	Starts wrapping opcodes that don't fit listing column
.MESSAGE (see on page 1-94)	Alters the severity of an error, warning or informational message generated by the assembler
.NEWPAGE (see on page 1-98)	Inserts a page break in a listing
.NOLIST (see on page 1-88)	Stops listing of source lines
.NOLIST_DATA (see on page 1-89)	Stops listing of data opcodes
.NOLIST_DATFILE (see on page 1-90)	Stops listing of data initialization files
.NOLIST_WRAPDATA (see on page 1-93)	Stops wrapping opcodes that do not fit listing column
.PAGELENGTH (see on page 1-99)	Defines the length of a listing page
.PAGEWIDTH (see on page 1-100)	Defines the width of a listing page
.PORT (see on page 1-101)	Legacy directive. Declares a memory-mapped I/O port. **NOTE:** SHARC processors ONLY.
.PRECISION (see on page 1-102)	Defines the number of significant bits in a floating-point value. **NOTE:** SHARC processors ONLY.
.PREVIOUS (see on page 1-103)	Reverts to a previously described .SECTION
.PRIORITY (see on page 1-104)	Allows prioritized symbol mapping in the linker
.REFERENCE (see on page 1-107)	???. **NOTE:** Blackfin processors ONLY.
.ROUND_NEAREST (see on page 1-108)	Specifies the Round-to-Nearest mode. **NOTE:** SHARC processors ONLY.

Table 1-16. Assembler Directive Summary (Cont'd)

Directive	Description
.ROUND_MINUS (see on page 1-108)	Specifies the Round-to-Negative Infinity mode. **NOTE:** SHARC processors ONLY.
.ROUND_PLUS (see on page 1-108)	Specifies the Round-to-Positive Infinity mode. **NOTE:** SHARC processors ONLY.
.ROUND_ZERO (see on page 1-108)	Specifies the Round-to-Zero mode. **NOTE:** SHARC processors ONLY.
.SECTION (see on page 1-111)	Marks the beginning of a section
.SEGMENT (see on page 1-117)	Legacy directive. Replaced with the .SECTION directive. **NOTE:** SHARC processors ONLY.
.SEPARATE_MEM_SEGMENTS (see on page 1-117)	Specifies two buffers that should be placed into different memory segments by the linker. **NOTE:** TigerSHARC processors ONLY.
.SET (see on page 1-118	Sets symbolic aliases.
.STRUCT (see on page 1-118)	Defines and initializes data objects based on C typedefs from .IMPORT C header files
.TYPE (see on page 1-122)	Changes the default data type of a symbol; used by C compiler
.VAR (see on page 1-123)	Defines and initializes 32-bit data objects
.WEAK (see on page 1-128)	Creates a weak definition or reference

Table 1-17 lists assembler directives supported only on Blackfin processors.

Table 1-17. VisualDSP++ 4.5 Blackfin-Only Assembler Directives

Directive	Description
`.BSS`	Equivalent to `.SECTION/zero_init bsz;` Refer to ".SECTION, Declare a Memory Section" on page 1-111 for more information.
`.DATA`	Equivalent to `.SECTION data1;` Refer to ".SECTION, Declare a Memory Section" on page 1-111 for more information.
`.GLOBL`	Equivalent to `.GLOBAL`. Refer to ".GLOBAL, Make a Symbol Globally Available" on page 1-82 for more information.
`.INCBIN`	Includes binary files directly for section output. `.INCBIN` is equivalent to `.INC/BINARY` (see on page 1-86).
`.LONG EXPRESSION-LIST`	Supports four-byte data initializer lists.
`.SHORT EXPRESSION-LIST`	Supports two-byte data initializer lists.
`.TEXT`	Equivalent to `.SECTION program;` Refer to ".SECTION, Declare a Memory Section" on page 1-111 for more information.

.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 the *VisualDSP++ 4.5 Linker and Utilities Manual* for more information on section alignment.

Syntax:

```
.ALIGN expression;
```

where

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

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

Example

```
…
.ALIGN 1;       // no alignment requirement
…
.SECTION data1;
.ALIGN 2;
.VAR single;
```

```
          /* aligns the data item on the word boundary,
          at the location with the address value that can be
          evenly divided by 2 */
.ALIGN 4;
.VAR samples1[100]="data1.dat";
          /* aligns the first data item on the double-word
          boundary, at the location with the address value
          that can be evenly divided by 4;
          advances other data items consecutively */
```

(i) The Blackfin assembler uses .BYTE instead of .VAR.

.ALIGN_CODE, Specify an Address Alignment

(i) Used with TigerSHARC processors ONLY.

The `.ALIGN_CODE` directive forces the address alignment of an instruction within the `.SECTION` it is used. It is similar to the `.ALIGN` directive, but whereas `.ALIGN` causes the code to be padded with 0s, `.ALIGN_CODE` pads with NOPs. The `.ALIGN_CODE` directive is used when aligning instructions.

Refer to Chapter 2 "Linker" in the *VisualDSP++ 4.5 Linker and Utilities Manual* for more information on section alignment.

Syntax:

```
.ALIGN_CODE expression;
```

where

> *expression* – evaluates to an integer. It specifies an alignment requirement; its value must be a power of 2. In TigerSHARC processors, the *expression* value is usually 4. When aligning a data item or instruction, the assembler adjusts the address of the current location counter to the next address that is divisible by the value of the *expression*. The expression set to 0 or 1 signifies no address alignment requirement.

(i) In the absence of the `.ALIGN_CODE` directive, the default address alignment is 1.

Example

```
.ALIGN_CODE 0;   /* no alignment requirement */
…
.ALIGN_CODE 1;   /* no alignment requirement */
   …
.SECTION program;
```

```
.ALIGN_CODE 4;
JUMP LABEL;;
   /* Jump instruction aligned to four word boundary.
      If necessary, padding will be done with NOPs */
```

.ASCII

(i) Used with Blackfin processors ONLY.

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

- There is no "=" sign

- The string is enclosed in double-quotes, not single quotes

Syntax:

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

Example:

```
.SECTION data1;

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

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

.BYTE, Declare a Byte Data Variable or Buffer

(i) Used with Blackfin processors ONLY.

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 assembler will look in the directories in the processor include path. You may

use the -I switch (see on page 1-142) 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 1-53).

The following lines of code demonstrate .BYTE directives:

```
Buffer1:
    .TYPE Buffer1, STT_OBJECT;
    .BYTE = 5, 6, 7;
    // initialize three 8-bit memory locations for data label
Buffer1
.Buffer1.end:
.BYTE samples[] = 123, 124, 125, 126, 127;
    // declare an implicit-length buffer and initialize it
    // with five 1-byte constants
.BYTE4/R32 points[] = 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 .

ASCII String Initialization Support

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

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

String initialization takes one of the following forms:

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

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

Example:

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

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

.EXTERN, Refer to a Globally Available Symbol

The .EXTERN directive allows a code module to reference global data structures, symbols, and so on. that are declared as .GLOBAL in other files. For additional information, see the .GLOBAL directive .

Syntax:

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

where

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

Example:

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

.EXTERN STRUCT, Refer to a Struct Defined Elsewhere

The `.EXTERN STRUCT` directive allows a code module to reference a struct 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 (SHARC code):

```
.IMPORT "MyCelebrities.h";
   // 'Celebrity' is the typedef for struct var 'StNick'
   // .EXTERN means that '_StNick' is referenced within this
   // file, but not locally defined. This example assumes StNick
   // was declared in a C file and it must be referenced with
   // a leading underscore.
.EXTERN STRUCT Celebrity _StNick;
   // 'isSeniorCitizen' is one of the members of the 'Celebrity'
   // type
P3.1 = LO( _StNick->isSeniorCitizen);
P3.h = HI(_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.

.FILE_ATTR, Create an attribute in the object file

The `.FILE_ATTR` directive instructs the assembler to place an attribute in the object file which can be referenced in the LDF file when linking. See the *VisualDSP++ 4.5 Linker and Utilities Manual* for more information.

Syntax:

```
.FILE_ATTR attrName1 [= attrValue1] [ , attrName2 [= attrValue2] ]
```

where

> `attrName` – the name of the attribute. Attribute names must follow the same rules for naming symbols.
>
> `attrValue` – sets the attribute to this value. If omitted, "1" is used. The value must be double quoted unless it follows the rules for naming symbols (as described in "Assembler Keywords and Symbols" on page 1-34).

Examples:

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

.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 1-77. 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 on one line, separated by commas.

Example (SHARC and TigerSHARC code):

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

Example (Blackfin code):

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

.IMPORT, Provide Structure Layout Information

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

- The names of `typedefs` and `structs` available

- The name of each data member

- The sequence and offset of the data members

- Information as provided by the C compiler for the size of C base types (alternatively, for 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. Allocating space requires the `.STRUCT` directive (see on page 1-118).

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 rebuild is needed. Use the `-flags-compiler` assembler switch option (on page 1-139) 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 .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 1-77 and the .STRUCT directive on page 1-118.

Example:

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

.INC/BINARY, Include Contents of a File

(i) Used with Blackfin processors ONLY.

The `.INC/BINARY` directive includes the content of file at the current location. You can control the search paths used via the `-i` command-line switch (on page 1-142).

Syntax:

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

where

> `symbol` – the name of a symbol to associate with the data being included from the file
>
> `filename` – the name of the file to include. The argument is enclosed in double quotes.
>
> The `skip` argument skips a number of bytes from the start of the file.
>
> The `count` argument indicates the maximum number of bytes to read.

Example:

```
.SECTION data1;

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

.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 -1 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. */
```

(i) 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, are listed until a `.LIST`directive is encountered in the same source, at the same nesting level. The `.NOLIST` directive operates on the next source line, so that the line containing a `.NOLIST` appears in the listing and accounts for the missing lines.

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

Syntax:

```
.LIST;
```

```
.NOLIST;
```

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

.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).

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).

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 .

.LIST_WRAPDATA/.NOLIST_WRAPDATA

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

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

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

- If `.NOLIST_WRAPDATA` is in effect, the printout is what fits in the opcode column.

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

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

Syntax:

```
.LIST_WRAPDATA;

.NOLIST_WRAPDATA;
```

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

.MESSAGE, Alter the severity of an assembler message

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

Syntax:

```
.MESSAGE/qualifier warnid1[,warnid2,…];
.MESSAGE/qualifier warnid1[,warnid2,…] UNTIL sym;
.MESSAGE/qualifier warnid1[,warnid2,…] FOR n LINES;
.MESSAGE/DEFAULT/qualifier warnid1[,warnid2,…];
```

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

A qualifier can be:

- ERROR – change messages to errors

- WARN – change messages to warnings

- INFO – change messages to informational

- SUPPRESS – do not output the messages

- RESTORE_CL – change the severity of the messages back to the default values they had at the beginning of the source file, after the command line arguments were processed, but before any DEFAULT directives have been processed.

- RESTORE – change the severity of the messages back to the default values they had at the beginning of the source file, after the command line arguments were processed, and after any DEFAULT directives have been processed.

- POP – change the severity of the messages back to what it was prior to the previous .MESSAGE directive.

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

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

The simple form of the .MESSAGE directive changes the severity of messages until another .MESSAGE directive is seen. It can be placed anywhere in a source file. Messages that could not be associated with a source line can be reported with line number 0. These cannot be altered in severity by a .MESSAGE directive. This should be done by using the -Werror, -Wwarn, -Winfo, or -Wsuppress assembler switches. (See "Assembler Command-Line Switch Descriptions" on page 1-132.)

Example:

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

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

Example (for TigerSHARC processors):

```
.SECTION program;
.VAR one=1.0r;                          // generates a warning
.MESSAGE/ERROR 1177  UNTIL sym;
.VAR two=1.0r;                          // generates an error
sym:
.VAR three=1.0r;                        // generates a warning
.MESSAGE/ERROR 1177  FOR 3 LINES;
```

```
.VAR four=1.0r;                          // generates an error
.VAR five=1.0r;                          // generates a warning
```

The POP qualifier changes the severity of the messages back to previous severities.

Example (for TigerSHARC processors):

```
.MESSAGE/INFO 3012;
.SECTION program;
RETI;;                             // generates an informational
.MESSAGE/ERROR 3012;
 RETI;;                            // generates an error
.MESSAGE/INFO 3012;
RETI;;                             // generates an informational
.MESSAGE/POP 3012;
RETI;;                    // generates an error - 2nd directive
.MESSAGE/POP 3012;
RETI;;            // generates an informational - 1st directive
.MESSAGE/POP 3012;
RETI;;      // generates a warning - the default for this message
```

The DEFAULT qualifier is used to redefine the default severity for messages. It can be placed anywhere in a source file. It only takes affect when the message severity has not been changed by a .MESSAGE directive.

Example (for TigerSHARC processors):

```
.MESSAGE/DEFAULT/ERROR 1177;
.MESSAGE/DEFAULT/INFO 1177;
.SECTION program;
.VAR one=1.0r;                  // generates an informational
.MESSAGE/ERROR 1177;
.VAR two=1.0r;                  // generates an error
.MESSAGE/RESTORE 1177;
.VAR three=1.0r;                // generates an informational
```

```
.MESSAGE/RESTORE_CL 1177;
.VAR four=1.0r;                     // generates a warning
```

> (i) The `-Werror`, `-Wwarn`, `-Winfo`, or `-Wsuppress` assembler switches
> have the same affect as the DEFAULT form of .MESSAGE. (See "Assem-
> bler Command-Line Switch Descriptions" on page 1-132.)

Many error messages cannot be altered in severity as the assembler behav-
ior is unknown.

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

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

.NEWPAGE, Insert a Page Break in a Listing File

The .NEWPAGE directive inserts a page break in the printed listing file
(.LST), which the assembler produces when you use the -l switch
(on page 1-143). 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

Syntax:

```
.PAGELENGTH expression;
```

where

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

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

Example:

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

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

.PAGEWIDTH, Set the Page Width of a Listing File

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

Syntax:

> `.PAGEWIDTH` *expression*;

where

> *expression* – evaluates to an integer.

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

LEFTMARGIN value plus 46   for Blackfin processors
LEFTMARGIN value plus 49   for TigerSHARC processors
LEFTMARGIN value plus about 66   for SHARC processors

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

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

Example:

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

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

.PORT, Legacy Directive

(i) Used with SHARC processors ONLY.

The .PORT legacy directive assigns port name symbols to I/O ports. Port name symbols are global symbols; they correspond to memory-mapped I/O ports defined in the Linker Description File (.LDF).

The .PORT directive uses the following syntax:

```
.PORT portName;
```

where:

> *portName* – a globally available port symbol.

Example:

```
.PORT p1;   // declares I/O port p1
.PORT p2;   // declares I/O port p2
```

To declare a port using the SHARC assembler syntax, use the .VAR directive (for port-identifying symbols) and the Linker Description File (for corresponding I/O sections). The linker resolves port symbols in the .LDF file.

For more information on the Linker Description File, see the *VisualDSP++ 4.5 Linker and Utilities Manual* .

.PRECISION, Select Floating-Point Precision

(i) Used with SHARC processors ONLY.

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

Use one of the following options:

```
.PRECISION [=] 32;
.PRECISION [=] 40;
```

where:

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

Example:

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

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

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

The `.ROUND_` directives (on page 1-108) specify how the assembler converts a value of many significant bits to fit into the selected precision.

.PREVIOUS, Revert to the Previously Defined Section

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

Syntax:

```
.PREVIOUS;
```

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

Example of Illegal Directive Use

```
.SECTION data1;        // data
.SECTION code;         // 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             // invoke macro
.PREVIOUS;
    .VAR vc = 3;
```

evaluates as:

```
.SECTION data1;        // data
    .VAR va = 1;
.SECTION program;      // instructions
```

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

.PRIORITY, Allow Prioritized Symbol Mapping in the Linker

The .PRIORITY directive allows prioritized symbol mapping in the linker. The directive can be specified in three ways:

- For a symbol defined in the same file as the directive

- For a globally defined symbol

- For a local symbol in a different source file

Syntax:

```
.PRIORITY symbolName, priority;

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

where:

In the first case *symbolname* is either a global symbol or locally defined symbol. In the second case, *symbolname* is a symbol defined in *'sourcefile'*.

Example:

```
.PRIORITY _foo, 35;          // Symbol with highest priority
.PRIORITY _main, 15;          // Symbol with medium priority
.PRIORITY bar, 'barFile.asm', -10;        // Symbol with lowest
priority
```

Linker Operation

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

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

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

Example of assembler code:

```
section program;
    _func1:

    _func2:


section L1_code;
    _L1_func:
    ...

priority _L1 func,10;
priority _func1,11;
priority _func2,12;
```

Example of LDF code:

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

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

1. Because _func2 is assigned the highest priority (12) in the assembler code, the linker first tries to map it into the L1_B memory segment. If _func2 does not fit into L1_B , it tries the L2 segment.

2. Because _func1 is assigned the middle priority (11) in the assembler code, the linker first tries to map it into the L1_B memory segment . If _func2 does not fit into L1_B , it tries the L2 segment.

3. Because _L1_func is assigned the lowest priority (10) in the assembler code, the linker first tries to map it into the L1_A memory segment . If _L1_func does not fit into L1_A , it tries the L1_B segment.

.REFERENCE,

(i) Used with Blackfin processors ONLY.

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

The .REFERENCE directive uses the following syntax:

```
.REFERENCE symbol;
```

where:

      symbol —

Example:

```
.REFERENCE p1;   //
.REFERENCE p2;   //
```

.ROUND_, Select Floating-Point Rounding

(i) Used with SHARC processors ONLY.

The `.ROUND_` directives control how the assembler interprets literal floating-point numeric data after `.PRECISION` is defined. The `.PRECISION` directive determines the number of bits to be truncated to match the number of significant bits (see on page 1-102).

The `.ROUND_` directives determine only how the assembler handles the floating-point values in constant declarations and variable initializations. To configure the floating-point rounding modes of the target processor system, you must set up control registers of the chip using the instructions that specific to the processor core.

The `.ROUND_` directives use the following syntax:

```
.ROUND_mode;
```

where:

> The mode string specifies the rounding scheme used to fit a value in the destination format. Use one of the following IEEE standard modes:
>
> > `.ROUND_NEAREST;`   (default)
> >
> > `.ROUND_PLUS;`
> >
> > `.ROUND_MINUS;`
> >
> > `.ROUND_ZERO;`

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

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

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

     1.2581 rounds to 1.258
     8.5996 rounds to 8.599
     -5.3298 rounds to -5.329
     -6.4974 rounds to -6.497
     */

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

     1.2581 rounds to 1.259
     8.5996 rounds to 8.600
     -5.3298 rounds to -5.329
     -6.4974 rounds to -6.497
     */

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

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

.SECTION, Declare a Memory Section

The `.SECTION` directive marks the beginning of a logical section 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.

TigerSHARC and Blackfin Syntax:

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

SHARC Syntax:

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

(i) All qualifiers are optional and more than one can be used.

Common .SECTION Attributes

The following are common syntax attributes used by the assembler:

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

  Note: Some sections starting with "." names have certain meaning within the linker. Do not use the dot (.) 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. To avoid ambiguity, ensure that your section names do not begin with ".`rela`.".

- *sectionType* – an optional ELF section type identifier. The assembler uses the default SHT_PROGBITS when this identifier is absent. For example,   .SECTION program SHT_DEBUGINFO;

  Supported ELF section types are SHT_PROGBITS, SHT_DEBUGINFO, and SHT_NULL. These 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++ 4.5 Linker and Utilities Manual* .

  🚫 If you select an invalid common qualifier or specify no common qualifier, the assembler exits with an error message.

Blackfin Example:

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

DOUBLE* Qualifiers

The DOUBLE* qualifier can be one of:

Table 1-18. DOUBLE Qualifiers

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

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

(i) Use of `DOUBLEANY` in a section implies that `doubles` are not used in this section in any way that would require consistency checking with any other section.

TigerSHARC-Specific Qualifiers

In addition, the TigerSHARC-specific *qualifier1*, *qualifier2...* can be one of the following, listed in Table 1-19:

Table 1-19. TigerSHARC-Specific Qualifiers

CHAR8	CHAR32	CHARANY
`CHARs` are represented as 8-bit types. `Shorts` are represented as 16-bit types.	`CHARs` are represented as 32-bit types. `Shorts` are represented as 32-bit types.	Section does not include code that depends on the size of `CHAR`.

The `char` size qualifiers are used to ensure that object files are consistent when linked together and with run-time libraries. A section may have a `double` size qualifier and a `char` size qualifier. It cannot have two `char` size qualifiers. Sections in the same file do not have to have the same type size qualifiers.

(i) **Note:** Use of `CHARANY` in a section implies that `char` and `shorts` are not used in this section in any way that would require consistency checking with any other section.

SHARC-Specific Qualifiers

For the SHARC assembler, the .SECTION directive supports qualifiers that specify the size of data words in the section, and a qualifier that may be used to specify restricted placement for the section. Each section that defines data or code must bear an appropriate size qualifier; the placement qualifier is optional. Table 1-20 lists the SHARC-specific qualifiers.

Table 1-20. SHARC-Specific Qualifiers

Memory/Section Type	Description
PM or Code	Section contains instructions and/or data, in 48-bit words
DM or Data	Section contains data in 40-bit words.
DATA64	Section defines data in 64-bit words.
DMAONLY	Section is to be placed in memory that can be accessed only through DMA

The DMAONLY qualifier does enforce that access to the section contents occurs through DMA alone; this qualifier passes to the linker the request that this section is to be placed in a memory segment that has the DMAONLY qualifier, which applies to memory accessed through the External Parallel Port of ADSP-2126x and some ADSP-2136x devices. For example,

```
.SECTION/DM/DMAONLY seg_extm;
.VAR _external_var[100];
```

Initialization Section Qualifiers

The `.SECTION` directive may identify "how/when/if" a section is initialized. The initialization qualifiers, common for all supported assemblers, are:

- The `/NO_INIT` qualifier:
  Section is "sized" to have enough space to contain all data elements placed in this section. No data initialization is happening for this memory section.

- The `/ZERO_INIT` qualifier:
  Similar to `NO_INIT`, except that the memory space for this section is initialized to zero at either "load" or "runtime", if invoked with the linker's `-meminit` switch. If the (linker's) `-meminit` switch is not used, the memory is initialized at "load" time when the `.DXE` file is loaded via VisualDSP++ IDDE, or boot-loaded by the boot kernel. If the memory initializer is invoked, the C/C++ run-time library (CRTL) processes embedded information to initialize the memory space during the CRTL initialization process.

- The `/RUNTIME_INIT` qualifier:
  If the memory initializer is not run, this qualifier has no effect. If the memory initializer is invoked, the data for this section is set during the CRTL initialization process.

For example,

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

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

Initialized data in a `/NO_INIT` or `/ZERO_INIT` section is ignored. For example, the assembler can generate a warning for the `.VAR zz` initialization.

```
.SECTION/NO_INIT seg_bss;
.VAR xx[1000];
.VAR zz = 25;    // [Warning ea1141] "example.asm":3 'zz':
```

```
Data directive with assembly-time initializers found
in .SECTION 'seg_bss' with qualifier /NO_INIT.
```

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

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

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

Table 1-21. Section Qualifiers, Section-Header-Types, and Section-Header-Flags

.SECTION Qualifier	ELF SHT_* (Elf.h) Section-Header-Type	ELF SHF_* (Elf.h) Section-Header-Flags
.SECTION/NO_INIT	SHT_NOBITS	SHF_ALLOC
.SECTION/ZERO_INIT	SHT_NOBITS	SHF_ALLOC, SHF_INIT
.SECTION/RUNTIME_INIT	SHT_PROGBITS	SHF_ALLOC, SHF_INIT

For more information, see the *VisualDSP++ 4.5 Linker and Utilities Manual* .

.SEGMENT & .ENDSEG, Legacy Directives

(i) Used with SHARC processors ONLY.

Releases of the ADSP-210xx DSP development software prior to VisualDSP 4.1 used the .SEGMENT and .ENDSEG directives to define the beginning and end of a section of contiguous memory addresses.

Although these directives have been replaced with the .SECTION directive, the source code written with .SEGMENT/.ENDSEG legacy directives is accepted by the ADSP-21xxx assembler.

.SEPARATE_MEM_SEGMENTS

(i) Used with TigerSHARC processors ONLY.

The .SEPARATE_MEM_SEGMENTS directive allows you to specify two buffers the linker should try to place into different memory segments.

Syntax:

```
.SECTION data1;
.VAR buf1;
.VAR buf2;
.EXTERN buf3;
.SEPARATE_MEM_SEGMENTS buf1, buf2
.SEPARATE_MEM_SEGMENTS buf1, buf3
```

You can also use the compiler's separate_mem_segments pragma to, perform the same function. For more information, refer to Chapter 2 of the *VisualDSP++ 4.5 Linker and Utilities Manual*.

.SET, Set a Symbolic Alias

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

Syntax:

```
.SET symbol1, symbol2
```

Where where symbol1 becomes an alias to symbol2.

Example

```
.SET symbol1, symbol1
```

.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 is 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* is 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 Data Variable or Buffer

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

When declaring or initializing variables:

- A `.VAR` directive 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, on one line.

  > 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 `z` in the processor 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;
           L0 = 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* – user-defined symbols that identify variables

- *bufferName* – user-defined symbols that identify buffers

- *fileName* parameter – indicates that the elements of a buffer get their initial values from the *fileName* data file. The <fileName> can consist of the actual name and path specification for the data file. If the initialization file is in the current directory of your operating system, only the *fileName* 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 (…) – 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.

(i) With Blackfin processors, the assembler uses a /R32 qualifier (.VAR/R32) to support 32-bit initialization for use with 1.31 fracts (see on page 1-53).

The following lines of code demonstrate some .VAR directives:

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

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

Blackfin Code Example:

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

```
.SECTION data1;
.ALIGN 4;
.VAR x0;
.VAR x1;
.BYTE b0;
.ALIGN 4;    // aligns the following data item y0 on a word
             // boundary; advances other data items
             // consequently
.VAR y0;
.VAR y1;
.BYTE b1;
.ALIGN 4;    // aligns the following data item y2 on a word
             // boundary
.VAR y2;
```

.VAR and ASCII String Initialization Support

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

On SHARC and TigerSHARC processors, the characters are stored in the upper byte of 32-bit words. The least significant bits (LSBs) are cleared.

When using the 16-bit Blackfin processors, refer to the .BYTE directive description for more information.

String initialization takes one of the following forms:

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

.VAR symbolString[] = {'initString', 0};
```

Note that the number of initialization characters defines length of a string (and implicit-size initialization requires the use of curly brackets).

For example,

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

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

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

The assemblers also accept ASCII characters within comments. Please note special characters handling:

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

.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 does not extract objects from archives to satisfy weak references. Such references, left unresolved, have the value 0.

(i) The .WEAK (or .GLOBAL scope) directive is required to make symbols available for placement through RESOLVE commands in the .LDF file.

Assembler Command-Line Reference

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

This section contains:

- "Running the Assembler" on page 1-130

- "Assembler Command-Line Switch Descriptions" on page 1-132

Command-line switches control certain aspects of the assembly process, including debugging information, listing, and preprocessing. Because the assembler automatically runs the preprocessor as your program 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 1-151 for more information.

Running the Assembler

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

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

Table 1-22 explains these arguments.

Table 1-22. Assembler Command Line Arguments

Argument	Description
easm21K easmts easmblkfn	Name of the assembler program for SHARC, TigerSHARC, and Blackfin processors, respectively.
-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 1-23 summarizes file extension conventions accepted by the VisualDSP++ environment.

Table 1-23. File Name Extension Conventions

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

Assembler command-line switches are case-sensitive. For example, the following command line

```
easmblkfn -proc ADSP-BF535 -l pList.lst -Dmax=100 -v -o bin\p1.doj p1.asm
```

runs the assembler with

   `-proc ADSP-BF535` – specifies assembles instructions unique to ADSP-BF535 processors.

   `-l pListing.lst` – directs the assembler to output the listing file.

   `-Dmax=100` – defines the preprocessor macro `max` to be 100.

-v – displays verbose information on each phase of the assembly.

-o bin\p1.doj – specifies the name and directory for the assembled object file.

p1.asm – identifies the assembly source file to assemble.

Assembler Command-Line Switch Descriptions

This section describes the assembler command-line switches in ASCII collation order. A summary of the assembler switches appears in Table 1-24. Detailed description of each assembler switch starts on page 1-135.

Table 1-24. Assembler Command-Line Switch Summary

Switch Name	Purpose
-align-branch-lines (on page 1-135)	Align branch lines to avoid ADSP-TS101 processor sequencer anomaly. **NOTE:** TigerSHARC processors ONLY.
-char-size-8 (on page 1-136)	Adds /CHAR8 to .SECTIONs in the source file. **NOTE:** TigerSHARC processors ONLY.
-char-size-32 (on page 1-136)	Adds /CHAR32 to .SECTIONs in the source file. **NOTE:** TigerSHARC processors ONLY.
-char-size-any (on page 1-136)	Adds /CHARANY to .SECTIONs in the source file. **NOTE:** TigerSHARC processors ONLY.
-default-branch-np (on page 1-136)	Make branch lines default to NP to avoid ADSP-TS101 processor sequencer anomaly. **NOTE:** TigerSHARC processors ONLY.
-default-branch-p (on page 1-137)	Make branch lines default to the Branch Target Buffer (BTB). **NOTE:** TigerSHARC processors ONLY.
-Dmacro[=definition] (on page 1-137)	Passes macro definition to the preprocessor.
-double-size-32 (on page 1-137)	Adds /DOUBLE32 to the .SECTIONs in the source file.

Table 1-24. Assembler Command-Line Switch Summary (Cont'd)

Switch Name	Purpose
`-double-size-64` (on page 1-138)	Adds `/DOUBLE64` to the `.SECTION`s in the source file.
`-double-size-any` (on page 1-138)	Adds `/DOUBLEANY` to the `.SECTION`s in the source file.
`-file-attr attr[=value]` (on page 1-138)	Creates an attribute in the generated object file.
`-flags-compiler -opt1...` (on page 1-139)	Passes each comma-separated option to the compiler. (Used when compiling `.IMPORT` C header files.)
`-flags-pp -opt1...` (on page 1-140)	Passes each comma-separated option to the preprocessor.
`-g` (on page 1-141)	Generates debug information (DWARF-2 format).
`-h[elp]` (on page 1-142)	Outputs a list of assembler switches.
`-i|-I directory pathname` (on page 1-142)	Searches a directory for included files.
`-l filename` (on page 1-143)	Outputs the named listing file.
`-li filename` (on page 1-144)	Outputs the named listing file with `#include` files expanded.
`-M` (on page 1-144)	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 1-144)	Generates make dependencies for `#include` and data files. Use `-MM` for make dependencies with assembly.
`-Mo filename` (on page 1-145)	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 1-24. Assembler Command-Line Switch Summary (Cont'd)

Switch Name	Purpose
`-Mt` *filename* (on page 1-145)	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 1-145)	Treats multi-issue conflicts as warnings. **NOTE:** Blackfin processors ONLY.
`-no-source-dependency` (on page 1-145)	Suppresses output of the source filename in the dependency output produced when "`-M`" or "`-MM`" has been specified.
`-o` *filename* (on page 1-146)	Outputs the named object [binary] file.
`-pp` (on page 1-146)	Runs the preprocessor only; does not assemble.
`-proc` *processor* (on page 1-146)	Specifies a target processor for which the assembler should produce suitable code.
`-save-temps` (on page 1-147)	Saves intermediate files
`-si-revision` *version* (on page 1-147)	Specifies silicon revision of the specified processor.
`-sp` (on page 1-148)	Assembles without preprocessing.
`-stallcheck=(none|cond|all)` (on page 1-148)	Displays stall information: • `none` - no messages • `cond` - conditional stalls only (default) • `all` - all stall information **NOTE:** Blackfin processors ONLY.
`-v[erbose]` (on page 1-149)	Displays information on each assembly phase.
`-version` (on page 1-149)	Displays version information for the assembler and preprocessor programs.
`-w` (on page 1-149)	Disables all assembler-generated warnings.

VisualDSP++ 4.5 Assembler and Preprocessor Manual

Table 1-24. Assembler Command-Line Switch Summary (Cont'd)

Switch Name	Purpose
`-Werror number[,number ...]` (on page 1-149)	Selectively turn assembler messages into errors.
`-Winfo number[,number ...]` (on page 1-149)	Selectively turns assembler messages into informationals.
`-Wno-info` (on page 1-149)	Does not display informational assembler messages..
`-Wnumber[,number ...]` (on page 1-150)	Selectively disables warnings by one or more message numbers. For example, `-W1092` disables warning message `ea1092`.
`-Wsuppress number[,number...]` (on page 1-150)	Selectively turns off assembler messages.
`-Wwarn number[,number ...]` (on page 1-150)	Selectively turns assembler messages into warnings.
`-Wwarn-error` (on page 1-150)	Display all assembler warning messages as errors.

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.

-align-branch-lines

(i) This switch is used with TigerSHARC processors ONLY.

The `-align-branch-lines` switch directs the assembler to align branch instructions (`JUMP`, `CALL`, `CJMP`, `CJMP_CALL`, `RETI`, and `RTI`) on quad-word boundaries by inserting `NOP` instructions prior to the branch instruction. This may be done by adding NOPs in free slots in previous instruction lines.

-char-size-8

The `-char-size-8` switch directs the assembler to add `/CHAR8` to
`.SECTION`s in the source file that do not have `char` size qualifiers.
For `.SECTION`s in the source file that already have a `char` size qualifier, this
option is ignored and a warning is produced. For more information, see
".SECTION, Declare a Memory Section" on page 1-111.

ⓘ This switch is used with TigerSHARC processors ONLY.

-char-size-32

The `-char-size-32` switch directs the assembler to add `/CHAR32` to
`.SECTION`s in the source file that do not have `char` size qualifiers.
For `.SECTION`s in the source file that already have a `char` size qualifier, this
option is ignored and a warning is produced. For more information, see
".SECTION, Declare a Memory Section" on page 1-111.

ⓘ This switch is used with TigerSHARC processors ONLY.

-char-size-any

The `-char-size-any` switch directs the assembler to add `/CHARANY` to
`.SECTION`s in the source file that do not have `char` size qualifiers.
For `.SECTION`s in the source file that already have a `char` size qualifier, this
option is ignored and a warning is produced. For more information, see
".SECTION, Declare a Memory Section" on page 1-111.

ⓘ This switch is used with TigerSHARC processors ONLY.

-default-branch-np

The `-default-branch-np` (branch lines default to NP) switch directs the
assembler to stop branch instructions (`JUMP`, `CALL`) from using the Branch
Target Buffer (BTB). This can be used to avoid a sequencer anomaly

present on the ADSP-TS101 processor only. It is still possible to make branch instructions use the BTB when `-default-branch-np` is used by adding the (`P`) instruction option; for example, `JUMP lab1 (P);;`.

(i) This switch is used with TigerSHARC processors ONLY.

-default-branch-p

The `-default-branch-p` switch makes branch instructions (`JUMP`, `CALL`) use the Branch Target Buffer (BTB). This is the default behavior. It is still possible to make branch instructions not use the BTB when `-default-branch-p` is used by adding the (`NP`) instruction option; for example, `JUMP labe1 (NP);;`.

(i) This switch is used with TigerSHARC processors ONLY.

-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 1-26 for the list of predefined macros. For example,

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

-double-size-32

The `-double-size-32` switch directs the assembler to add `/DOUBLE32` to `.SECTION`s in the source file that do not have `double` size qualifiers. For `.SECTION`s in the source file that already have a `double` size qualifier, this option is ignored and a warning is produced. For more information, see ".SECTION, Declare a Memory Section" on page 1-111.

-double-size-64

The `-double-size-64` switch directs the assembler to add `/DOUBLE64` to `.SECTION`s in the source file that do not have `double` size qualifiers. For `.SECTION`s in the source file that already have a `double` size qualifier, this option is ignored and a warning is produced. The `-double-size-any` flag should be used to avoid a linker warning when compiling C/C++ sources with `-double-size-64`.

Warning Example:

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

For more information, see ".SECTION, Declare a Memory Section" on page 1-111.

-double-size-any

The `-double-size-any` switch directs the assembler to add `/DOUBLEANY` to `.SECTION`s in the source file that do not have `double` size qualifiers. For `.SECTION`s in the source file that already have a `double` size qualifier, this option is ignored and a warning is produced. For more information, see ".SECTION, Declare a Memory Section" on page 1-111.

-file-attr attr[=val]

The `-file-attr` (file attribute) switch directs the assembler to add an attribute, `attr`, to the object file. The attribute will be given the value, `val`, or "1" if the value is omitted. `Attr` should follow the rules for naming symbols. `Val` should be double quoted unless it follows the rules for naming symbols. See "Assembler Keywords and Symbols" on page 1-34 for more information on naming conventions.

-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";
easmbkln -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 -c -debug-types -DbDef -DbDefTwo=2 myHeader.h
```

(i) See "Using Assembler Feature Macros" on page 1-26 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 -c -debug-types -I/cPath -I. myHeader.h
```

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

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

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

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

(i) Use `-flags-pp` with caution. For example, if the `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 complete data type information for arrays, functions, and the C structs. This switch also generates DWARF2 function information with starting and ending ranges based on the myFunc: … myFunc.end: label boundaries, as well as line number and symbol information in DWARF2 binary format, allowing you to debug the assembly source files.

When the assembler's -g debugging is in effect, the assembler produces a warning when it is unable to match a *.end label to a matching beginning label. This feature can be disabled using the -Wnnnn switch (see).

WARNING ea1121: Missing End Labels

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

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

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

To suppress a specific assembler warning by unique warning number, the assembler provides the following option:

```
-Wsuppress 1121
```

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

Functions (Code)

```
_gfxeng_vertspan:

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

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

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

-h[elp]

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

-i|I directory

The `-idirectory` or `-Idirectory` (include directory path) 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. Directory for assembly program.)

2. `...\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 the directory where the assembly service is, not the directory of the project. Usage of full path names for the `-I` switch on the command line is recommended.

For example,

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

-l filename

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

For example,

```
easmblkfn -proc ADSP-BF533 -I\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 1-31.

-li filename

The `-li` (listing) switch directs the assembler to generate the named listing file with `#include` files. The file name is a required argument to the `-li` option. For more information, see "Reading a Listing File" on page 1-31.

-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 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 quotation marks (" ").

The `-Mo` *filename* option takes precedence over the `-o` *filename* option.

-Mt filename

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

-micaswarn

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

This switch is used with Blackfin processors ONLY.

-no-source-dependency

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

-o filename

The `-o filename` (output file) 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`). By default, the assembler uses the root input file name for the output and appends a `.doj` extension.

Some examples of this switch syntax are:

```
easmblkfn -proc ADSP-BF535 -pp -o test1.is test.asm
        // preprocessed output goes into test1.is

easmblkfn -proc ADSP-BF535 -o -debug/prog3.doj prog3.asm
        // specify directory and filename 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 produces code suitable for the specified processor.

The `processor` identifiers directly supported by VisualDSP++ 4.5 are listed in "Supported Processors".

For example,

```
easm21K -proc ADSP-21161 -o bin\p1.doj p1.asm

easmts -proc ADSP-TS201 -o bin\p1.doj p1.asm

easmblkfn -proc ADSP-BF533 -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.

(i) See also the "-si-revision version" switch description for more information on silicon revision of the specified processor.

-save-temps

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

-si-revision version

The `-si-revision` *version* (silicon revision) switch directs the assembler to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The parameter "*version*" represents a silicon revision for the processor specified by the `-proc` switch (on page 1-146).

For example,

```
easmblkfn -proc ADSP- BF533 -si-revision 0.1
```

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

If the `-si-revision` switch is not used, the assembler will build for the latest known silicon revision for the target processor and any errata workarounds which are appropriate for the latest silicon revision will be enabled.

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

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

-sp

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

-stallcheck

The `-stallcheck = option` switch provides the following choices for displaying stall information:

Table 1-25. -stallcheck Options

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

This switch is used with Blackfin processors ONLY.

-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.

-Werror number[,number]

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

(i) Many error messages cannot be altered in severity as the assembler behavior is unknown.

-Winfo number[,number]

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

(i) Many error messages cannot be altered in severity as the assembler behavior is unknown.

-Wno-info

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

-Wnumber[,number]

The `-Wnumber` (warning suppression) switch selectively disables warnings specified by one or more message numbers. For example, `-W1092` disables warning message `ea1092`. This switch optionally accepts a list, such as `[,number ...]`. See also "-g" on page 1-141.

-Wsuppress number[,number]

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

(i) Many error messages cannot be altered in severity as the assembler behavior is unknown.

-Wwarn number[,number]

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

(i) Many error messages cannot be altered in severity as the assembler behavior is unknown.

-Wwarn-error

The `-Wwarn-error` switch displays all the assembler warning messages as errors.

Specifying Assembler Options in VisualDSP++

Within the VisualDSP++ IDDE, specify tool settings for project builds. Use the **Project** menu to open **Project Options** dialog box

For example, Figure 1-5 shows the project option selections for SHARC processors.
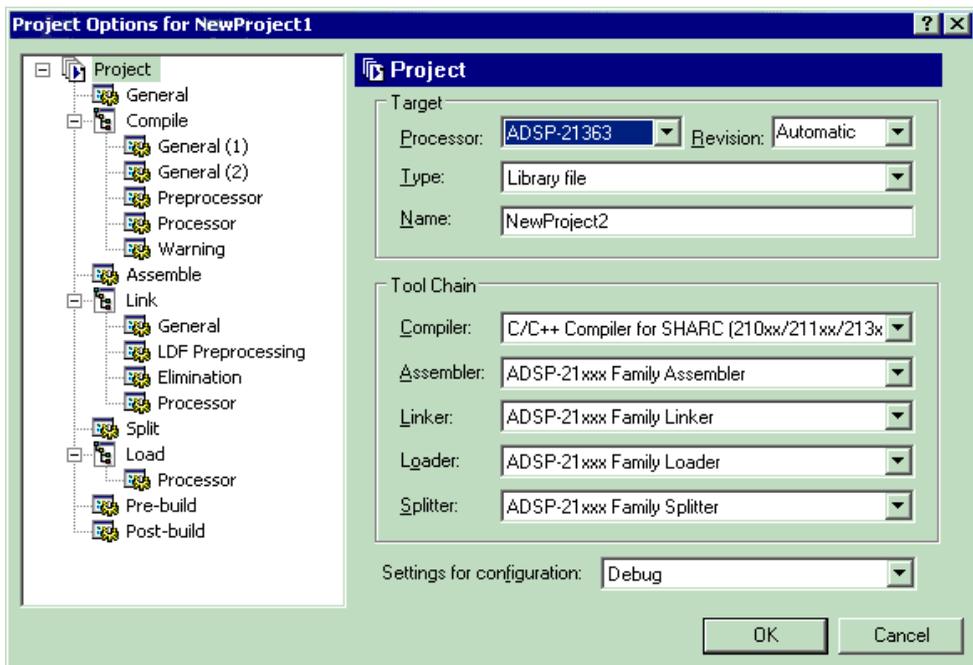
Figure 1-5. Project Options Dialog Box (SHARC Processors)

Figure 1-6 shows the project option selections for TigerSHARC processors.
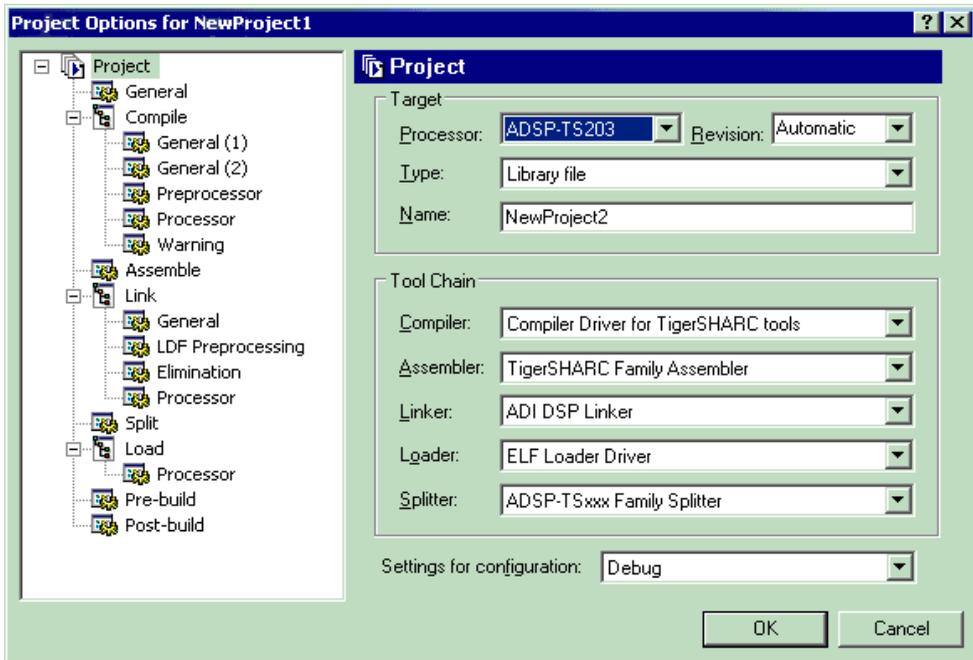
Figure 1-6. Project Options Dialog Box (TigerSHARC Processors)

Figure 1-7 shows the project option selections for Blackfin processors.
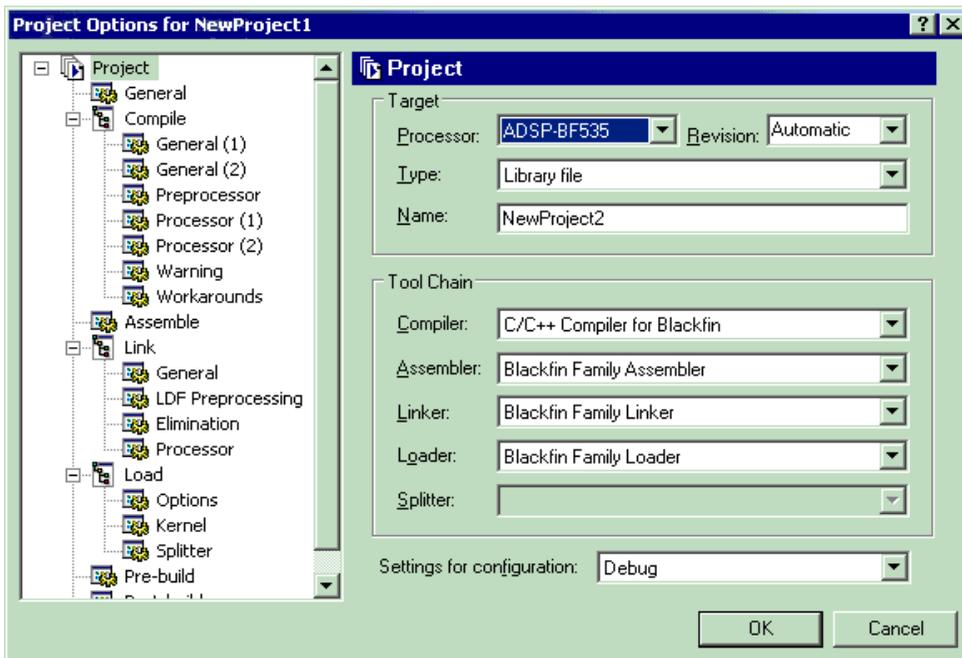
Figure 1-7. Project Options Dialog Box (Blackfin Processors)

These dialog boxes allow you to select the target processor, type and name of the executable file, as well as VisualDSP++ tools available for use with the selected processor.

When using the VisualDSP++ IDDE, use the **Assemble** option from the
Project Options dialog box (Figure 1-8) to select and/or set assembler
functional options.
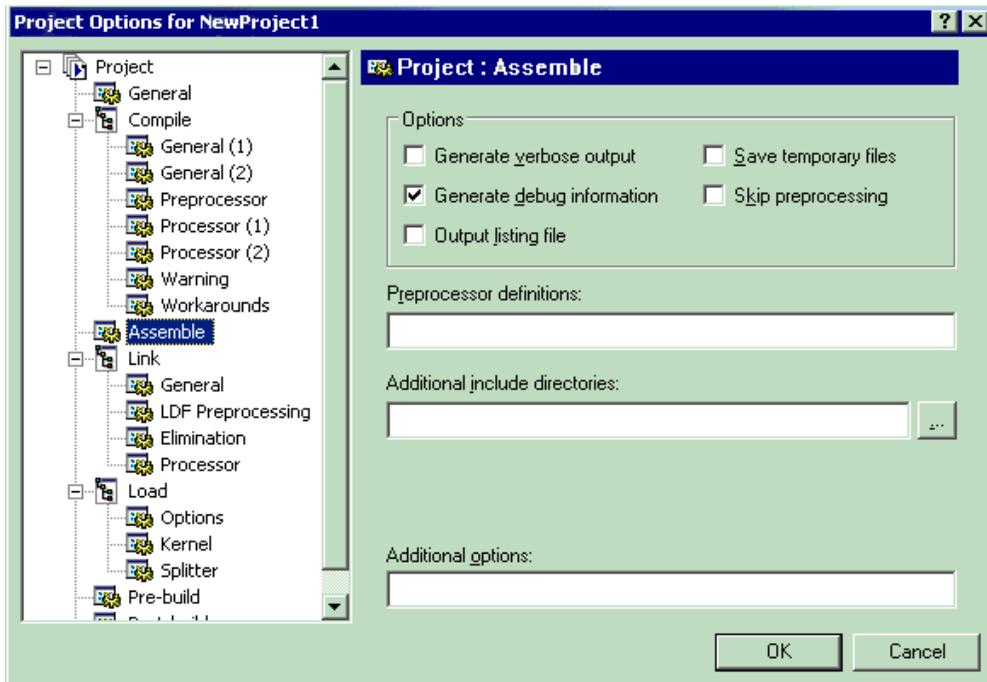
Figure 1-8. Project Options – Assemble Tab

Most setup options have corresponding assembler command-line switches
described in "Assembler Command-Line Switch Descriptions" on
page 1-132.

For more information, use the VisualDSP++ context-sensitive online Help
for each target architecture to select information on assembler options you
can specify in VisualDSP++. To do that, click on the **?** button and then
click in a field or box you need information about.

The **Additional options** field is used to enter the appropriate file names and options that do not have corresponding controls on the **Assemble** page but are available as assembler switches.

The assembler options apply to directing calls to an assembler 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 on all supported processors. 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 preprocessor is run by other build tools (assembler and linker) from the operating system's command line or within the VisualDSP++ 4.5 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.

This chapter contains:

Preprocessor Guide

This section contains the `pp` preprocessor information on how to build programs from a command line or from the VisualDSP++ 4.5 environment. Software developers using the preprocessor should be familiar with:

- "Writing Preprocessor Commands" on page 2-3

- "Header Files and #include Command" on page 2-4

- "Writing Macros" on page 2-6

- "Using Predefined Preprocessor Macros" on page 2-9

- "Specifying Preprocessor Options" on page 2-13

The compiler 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 preprocessor and has some features that may not be used within your assembly source files. For more information, see the *VisualDSP++ 4.5 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 2-34) 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;
```

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

The standard C preprocessor treats '.VAR' as two tokens, "." and "VAR", and makes the following substitution:

```
.my_var x;
```

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

Finally, 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 .

For example,

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

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

- Including system and user-defined header files

- Defining macros and symbolic constants

- Providing conditional assembly

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 macro references with corresponding expansions

The following cases are notable exceptions to the described substitutions:

- The preprocessor does not recognize comments or macros within the file name delimiters of an #include command.

- The preprocessor does not recognize comments or predefined macros within a character or string constant.

Header Files and #include Command

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 three forms for the #include command:

1. System Header Files
Syntax: #include <filename>
where a file name is within angle brackets. The file name in this form is interpreted as a "system" header file. These files are used to declare global definitions, especially memory-mapped registers, system architecture and processors.

Example:

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

System header files are installed in the **.../VisualDSP/Blackfin/include** folder for the processor family.

2. User Header Files

Syntax: `#include "filename"`

where a file name is within double quotes. The file name in this form is interpreted as a "user" header file. These files contain declarations for interfaces between the source files of the program.

Example:

```
#include "defTS.h"
#include "fft_ovly.h"
```

3. Sequence of Tokens

Syntax: `#include text`

In this case, "text" is a sequence of tokens that is subject to macro expansion by the preprocessor. It is an error if after macro expansion the text does not match one of the two header file forms.

In other words, if the text on the line after the "`#include`" is not included in either double quotes (as a user header) or angle brackets (as a system header), then the preprocessor performs macro expansion on the text. After that expansion, the line needs to have either of the two header file forms. It is important to note that unlike most preprocessor commands, the text after the `#include` is available for macro expansion.

Examples:

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

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

```
#include syshdr(adi)
// above evaluates to #include <adi.h>
```

Include Path Search

It is 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.

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

1. include path specified by the -I switch

2. .../VisualDSP/Blackfin/include folders

The #include "file" search order is:

1. local directory – the directory in which the source file resides

2. include path specified by the -I switch

3. ...VisualDSP/Blackfin/include folders

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

Writing Macros

The preprocessor processes macros in your C, C++, 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 for the last line, 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 occurs in the source file, instructions, commands, or memory descriptions. The macro definition may also have other macro names that are 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.

Blackfin Code Example:

```
#define false 0
#define xchg(xv,yv)\
    p0=xv;\
    p1=yv;\
    r0=[p0];\
    r1=[p1];\
    [p1]=r0;\
    [p0]=r1;
```

SHARC Code Example:

```
#define ccall(x) \
    r2=i6; i6=i7; \
    jump (pc, x) (db); \
```

```
      dm(i7,m7)=r2; \
      dm(i7, m7)=PC
         <in code section>
.
.         <instruction code here>
.
      ccall(label);
.
.           <instruction code here>
      .label:nop;
           <instruction code here>
```

TigerSHARC Code Example:

```
  #define copy (src,dest) \
      j0 = src;;        \
      j1 = dest;;       \
      R0 =[j0+0];;      \
      [j1+0] = R0;;
```

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 (;), the semicolon is not needed when it appears in an assembly statement. However, if a macro does not end with a semicolon character (";"), 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,

Blackfin Code Example:

```
#define mac mrf = mrf+r2*r5(ssfr)   // macro definition
   r2=r1-r0;                        // set parameters
   r5=dm(i1,m0);
   mac;                            // macro invocation
```

For more syntax information and usage examples for the #define preprocessor command, see "#define" on page 2-16.

Using Predefined Preprocessor Macros

In addition to macros you define, the pp preprocessor provides a set of predefined and feature macros that can be used in the assembly code. The preprocessor automatically replaces each occurrence of the macro reference found throughout the program with the specified (predefined) value. The DSP development tools also define feature macros that can be used in your code.

(i) 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 1-126).

Table 2-1 describes the common predefined macros provided by the pp preprocessor. Table 2-2, Table 2-3, and Table 2-4 list processor-specific feature macros that are defined by the project development tools to specify the architecture and language being processed.

Table 2-1. Common Predefined Preprocessor Macros

Macro	Definition
ADI	Defines ADI as 1.
__LastSuffix__	Specifies the last value of suffix that was used to build preprocessor generated labels.
__LINE__	Replaces __LINE__ 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'.
__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'.
_LANGUAGE_ASM	Always set to 1
_LANGUAGE_C	Equal 1 when used for C compiler calls to specify .IMPORT headers. Replaces _LANGUAGE_ASM.

Table 2-2. SHARC Feature Preprocessor Macros

Macro	Definition
__ADSP21000__	Always 1 for SHARC processor tools
__ADSP21020__	Present when running easmts -proc ADSP-21020 with ADSP-21020 processor
__ADSP21060__	Present when running easmts -proc ADSP-21060 with ADSP-21060 processor
__ADSP21061__	Present when running easmts -proc ADSP-21061 with ADSP-21061 processor
__ADSP21062__	Present when running easmts -proc ADSP-21062 with ADSP-21062 processor
__ADSP21065L__	Present when running easmts -proc ADSP-21065L with ADSP-21065L processor
__ADSP21160__	Present when running easmts -proc ADSP-21160 with ADSP-21160 processor

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

Macro	Definition
__ADSP21161__	Present when running `easmts -proc ADSP-21161` with ADSP-21161 processor
__ADSP2106x__	Present when running `easmts -proc ADSP-2106x` with ADSP-2106x processor
__ADSP2116x__	Present when running `easmts -proc ADSP-2116x` with ADSP-2116x processor
__ADSP21261__	Present when running `easmts -proc ADSP-21261` with ADSP-21261 processor
__ADSP21262__	Present when running `easmts -proc ADSP-21262` with ADSP-21262 processor
__ADSP21266__	Present when running `easmts -proc ADSP-21266` with ADSP-21266 processor
__ADSP21267__	Present when running `easmts -proc ADSP-21267` with ADSP-21267 processor
__ADSP21363__	Present when running `easmts -proc ADSP-21363` with ADSP-21363 processor
__ADSP21364__	Present when running `easmts -proc ADSP-21364` with ADSP-21364 processor
__ADSP21365__	Present when running `easmts -proc ADSP-21365` with ADSP-21365 processor
__ADSP21366__	Present when running `easmts -proc ADSP-21366` with ADSP-21366 processor
__ADSP21367__	Present when running `easmts -proc ADSP-21367` with ADSP-21367 processor
__ADSP21368__	Present when running `easmts -proc ADSP-21368` with ADSP-21368 processor
__ADSP21369__	Present when running `easmts -proc ADSP-21369` with ADSP-21369 processor

Table 2-3. TigerSHARC Feature Preprocessor Macros

Macro	Definition
__ADSPTS__	Always 1 for TigerSHARC processor tools
__ADSPTS101__	Equal 1 when used with ASDP-TS101 processor
__ADSPTS201__	Equal 1 when used with ASDP-TS201 processor
__ADSPTS202__	Equal 1 when used with ASDP-TS202 processor
__ADSPTS203__	Equal 1 when used with ASDP-TS203 processor

Table 2-4. Blackfin Feature Preprocessor Macros

Macro	Definition
__ADSPBLACKFIN__	Always 1 for Blackfin processor 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__ __ADSP21533__=1	Present when running `easmblkfn -proc ADSP-BF533` with ADSP-BF533 processor.
__ADSPBF534__ __ADSP21534__=1	Present when running `easmblkfn -proc ADSP-BF534` with ADSP-BF534 processor.
__ADSPBF535__ __ADSP21535__=1	Present when running `easmblkfn -proc ADSP-BF535` with ADSP-BF535 processor.
__ADSPBF536__	Present when running `easmblkfn -proc ADSP-BF536` with ADSP-BF536 processor.
__ADSPBF537__	Present when running `easmblkfn -proc ADSP-BF537` with ADSP-BF537 processor.
__ADSPBF538__	Present when running `easmblkfn -proc ADSP-BF538` with ADSP-BF538 processor.
__ADSPBF539__	Present when running `easmblkfn -proc ADSP-BF539` with ADSP-BF539 processor.

Table 2-4. Blackfin Feature Preprocessor Macros (Cont'd)

Macro	Definition
__ADSPBF561__	Present when running `easmblkfn -proc ADSP-BF561` with ADSP-BF561 processor.
__AD6531__	Present when running `easmblkfn -proc AD6531` with AD6531 processor.

Specifying Preprocessor Options

When developing a DSP project, it may be useful to modify the preprocessor's default options. Because the assembler, compiler, and linker automatically run the preprocessor as your program is built (unless you skip the processing entirely), these project development tools can receive input for the preprocessor program and direct its operation. The way the preprocessor options are set depends on the environment used to run the project development software.

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

- From the operating system command line, select the preprocessor's command-line switches. For more information on these switches, see "Preprocessor Command-Line Switches" on page 2-37.

- In the VisualDSP++ environment, select the preprocessor's options in the **Assemble** or **Link** tabs (property pages) of the **Project Options** dialog boxes, accessible from the **Project** menu. Refer to "Specifying Assembler Options in VisualDSP++" on page 1-151 for the **Assemble** tab.

  For more information, see the *VisualDSP++ 4.5 User's Guide* and online Help.

Preprocessor Command Reference

This section provides reference information about the processor'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` or `.LDF`), 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 a binary object file (`.DOJ`) is created.

Preprocessor command syntax must conform to these rules:

- Must be the first non-whitespace character on its line

- Cannot be more than one line in length unless the backslash character (`\`) is inserted

- Cannot come from a macro expansion

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

Preprocessor Commands and Operators

Table 2-5 lists the preprocessor command set. Table 2-6 lists the preprocessor operator set. Sections that begin on page 2-16 describe each of the preprocessor commands and operators.

Table 2-5. Preprocessor Command Summary

Command/Operator	Description
`#define` (on page 2-16)	Defines a macro
`#elif` (on page 2-19)	Subdivides an `#if` … `#endif` pair

Table 2-5. Preprocessor Command Summary

Command/Operator	Description
#else (on page 2-20)	Identifies alternative instructions within an #if … #endif pair
#endif (on page 2-21)	Ends an #if … #endif pair
#error (on page 2-22)	Reports an error message
#if (on page 2-23)	Begins an #if … #endif pair
#ifdef (on page 2-24)	Begins an #ifdef … #endif pair and tests if macro is defined
#ifndef (on page 2-25)	Begins an #ifndef … #endif pair and tests if macro is not defined
#include (on page 2-26)	Includes contents of a file
#line (on page 2-28)	Sets a line number during preprocessing
#pragma (on page 2-29)	Takes any sequence of tokens
#undef (on page 2-30)	Removes macro definition
#warning (on page 2-31)	Reports a warning message

Table 2-6. Preprocessor Operator Summary

Command/Operator	Description
# (on page 2-32)	Converts a macro argument into a string constant. By default, this operator is OFF. Use the command-line switch "-stringize" on page 2-44 to enable it.
## (on page 2-33)	Concatenates two tokens
? (on page 2-34)	Generates unique labels for repeated macro expansions
… (on page 2-17)	Specifies a variable length argument list

#define

The #define command defines macros.

When defining macros in your source code, the preprocessor substitutes each occurrence of the macro with the defined text. Defining this type of macro has the same effect as using the **Find/Replace** feature of a text editor, although it does not replace literals in double quotation marks (" ") and does not replace a match within a larger token.

For macro definitions 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 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 macro named BUFFER_SIZE and sets its
        value to 1020. */

#define copy(src,dest)xr0=[j31+src ];; /
[        j31+dest] = xr0;;
        /* define a macro named copy with two arguments.
        The definition includes two instructions that copy
        a word from memory to memory.
        For example,
            copy (0x3F,0xC0);
        calls the macro, passing parameters to it.
        The preprocessor replaces the macro with the code:
            [xr0 = [j31+0x3F];;
            [j31+0xC0] = xr0;;
        */
```

Variable Length Argument Definitions

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

```
#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,

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. See Table 2-7.

For example,

Table 2-7. Sample Variable Length Argument List

Sample Argument List	Description
test(1)	Error; the macro must have at least one more argument than formal parameters, not counting "..."
test(1,2)	Valid entry
test(1,2,3,4,5)	Valid entry

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

Syntax:

```
#elif condition
```

where

condition – expression to evaluate as TRUE (nonzero) or FALSE (zero)

Example:

```
#if X == 1
   …
#elif X == 2
   ...
      /* The preprocessor includes text within the section
      and excludes all other text before #else when X=2. */
#else
#endif
```

#else

The #else command is used within an #if … #endif pair. It adds an alternative instruction to the #if … #endif pair. Only one #else command can be used inside the pair. The preprocessor executes instructions that follow #else after all the preceding conditions are evaluated as FALSE (zero). If no #else text is specified, and all preceding #if and #elif conditions are FALSE, the preprocessor does not include any text inside the #if … #endif pair.

Syntax:

```
#else
```

Example:

```
#if X == 1
    …
#elif X == 2
    …
#else
    …
    /* The preprocessor includes text within the section
    and excludes all other text before #else when
    x!=1 and x!=2. */
#endif
```

#endif

The #endif command is required to terminate #if … #endif, #ifdef … #endif, and #ifndef … #endif pairs. 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 pre-processor 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 terminated with an `#endif` command.

Syntax:

```
#if condition
```

where

> *condition* – expression to evaluate as `TRUE` (nonzero) or `FALSE` (zero)

Example:

```
#if x!=100      /* test for TRUE condition */
…
      /* The preprocessor includes text within the section
      if the test is true. */
#endif
```

More examples:

```
#if (x!=100) && (y==20)
#if defined(__ADSPBF535__)
```

#ifdef

The #ifdef (if defined) command begins an #ifdef … #endif pair and instructs the preprocessor to test whether the macro is defined. Each #ifdef command must have a matching #endif command.

Syntax:

```
#ifdef macroSymbol
```

where

> macroSymbol – macro identifying symbol

Example:

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

#ifndef

The #ifndef (if not defined) command begins an #ifndef ... #endif pair and directs the preprocessor to test for an undefined macro. The preprocessor considers a macro undefined if it has no defined value. Each #ifndef command must have a matching #endif command.

Syntax:

```
#ifndef macroSymbol
```

where

> macroSymbol – macro identifying symbol

Example:

```
#ifndef __ADSPBF535__
      /* Includes text after #ifndef only when __ADSPBF535__ is
      not defined */
#endif
```

#include

The `#include` command directs the preprocessor to insert the text from a header file at the command location. There are two types of header files: system and user. However, the `#include` command may be presented in three forms:

- `#include <filename>` – used with system headers

- `#include "filename"` – used with user headers

- `#include text` – used with a sequence of tokens
  The sequence of tokens is subject to macro expansion by the pre-processor. After macro expansion, the text must match one of the header file forms.

The only difference to the preprocessor between the two types of header files is the way the preprocessor searches for them.

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

- User Header "`fileName`" – The preprocessor searches for a user header file in this order:

  1. Current directory – the directory where the source file that has the `#include` command(s) lives

  2. Directories you specify

  3. Standard list of system directories

Refer to "Header Files and #include Command" on page 2-4 for more information.

Syntax:

```
#include <fileName>   // include a system header file
#include "fileName"   // include a user header file

#include macroFileNameExpansion
    /* Include a file named through macro expansion.
    This command directs the preprocessor to expand the
    macro. The preprocessor processes the expanded text,
    which must match either <fileName> or "fileName". */
```

Example:

```
#ifdef __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* – line number of the source line

> *sourceFile* – name of the source file included in double quotation marks. The *sourceFile* entry can include the drive, directory, and file extension as part of the file name.

Example:

#line 7 "myFile.c"

ⓘ All assembly programs have #line directives after preprocessing. They always have a first line with #line 1 "filename.asm" and they will also have #line directives to establish correct line numbers for text that came from include files as a result of the processed #include directives.

#pragma

The #pragma is the implementation-specific command that modifies the preprocessor behavior. The #pragma command can take any sequence of tokens. This command is accepted for compatibility with other VisualDSP++ software tools. The pp preprocessor currently does not support any pragmas; therefore, it ignores 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 causes the preprocessor to issue a warning. The preprocessor uses the text following the #warning command as the warning message.

Syntax:

```
#warning messageText
```

where

messageText – user-defined text

To break a long messageText without changing its meaning, place the slash character (/) at the end of each line except for the last line.

Example:

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

(Argument)

The # (argument) "stringization" operator directs the preprocessor to convert a macro argument into a string constant. The preprocessor converts an argument into a string when macro arguments are substituted into the macro definition.

The preprocessor handles white space in string-to-literal conversions by:

- Ignoring leading and trailing white spaces
- Converting 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 2-44) to enable it.

C Code Example:

```
#define WARN_IF(EXP)\
fprintf (stderr,"Warning:"#EXP "/n")
   /*Defines a macro that takes an argument and converts the
   argument to a string */
WARN_IF(current <minimum);
   /* Invokes the macro passing the condition. */
fprintf (stderr,"Warning:""current <minimum""/n");
   /* Note that the #EXP has been changed to current <minimum
   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;
```

The last numeric suffix used to generate unique labels is maintained by the preprocessor and is available through a preprocessor predefined macro `__LastSuffix__` (see). This value can be used to generate references to labels in the last macro expansion.

The following example assumes the macro "`loop`" from the previous example.

```
// Some macros for appending a suffix to a label
#define makelab(a, b) a##b
#define Attach(a, b) makelab(a##_, b)
#define LastLabel(foo) Attach( foo, __LastSuffix__)

// jump back to label in the previous expansion
JUMP LastLabel(mylabel);
```

The above expands to (the last macro expansion had a suffix of 3):

```
JUMP mylabel_3;
```

Preprocessor Command-Line Reference

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

This section contains:

- "Running the Preprocessor"
- "Preprocessor Command-Line Switches" on page 2-37

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]
```

Table 2-8 summarizes these arguments.

Table 2-8. Preprocessor Command Line Argument Summary

Argument	Description
pp	Name of the preprocessor program.
-switch	Switch (or switches) to process. The preprocessor offers several switches that are used to select its operation and modes. Some preprocessor switches take a file name as a required parameter.
sourceFile	Name of the source file to process. The preprocessor supports relative 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

(i) Most switches without arguments can be negated by prefixing `-no` to the switch; for example, `-nowarn` turns off warning messages, and `-nocs!` turns off omitting "`!`" style comments.

Preprocessor Command-Line Switches

The preprocessor is 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 2-9 lists the `pp.exe` switches. A detailed description of each switch appears beginning on page 2-39.

Table 2-9. Preprocessor Command-Line Switch Summary

Switch Name	Description
`-cstring` on page 2-39	Enables the "stringization" operator and provides C compiler-style preprocessor behavior

Table 2-9. Preprocessor Command-Line Switch Summary (Cont'd)

`-cs!` on page 2-40	Treats as a comment all text after "!" on a single line
`-cs/*` on page 2-40	Treats as a comment all text within `/*  */`
`-cs//` on page 2-40	Treats as a comment all text after `/ /`
`-cs{` on page 2-40	Treats as a comment all text within `{ }`
`-csall` on page 2-41	Accepts comments in all formats
`-Dmacro[=definition]` on page 2-41	Defines `macro`
`-h[elp]` on page 2-41	Outputs a list of command-line switches
`-i` on page 2-41	Outputs only makefile dependencies for `include` files specified in double quotes
`-i|Idirectory` on page 2-41	Searches `directory` for included files
`-M` on page 2-43	Makes dependencies only
`-MM` on page 2-44	Makes dependencies and produces preprocessor output
`-Mo filename` on page 2-44	Specifies `filename` for the make dependencies output file
`-Mt filename` on page 2-44	Makes dependencies for the specified source file
`-o filename` on page 2-44	Outputs named object file
`-stringize` on page 2-44	Enables stringization (includes a string in double quotes)
`-tokenize-dot` (on page 2-45)	Treats "." (dot) as an operator when parsing identifiers

Table 2-9. Preprocessor Command-Line Switch Summary (Cont'd)

`-Uname` on page 2-45	Undefines a macro on the command line
`-v[erbose]` on page 2-45	Displays information about each preprocessing phase
`-version` on page 2-46	Displays version information for the preprocessor
`-w` on page 2-46	Removes all preprocessor-generated warnings
`-Wnumber` on page 2-46	Suppresses any report of the specified warning
`-warn` on page 2-46	Prints warning messages (default)

The following sections describe preprocessor's command-line switches.

-cstring

The `-cstring` switch directs the preprocessor to produce "C compiler"-style strings in all cases. Note that by default, the preprocessor produces assembler-style strings within single quotes (for examples, '*string*') unless the `-cstring` switch is used.

The `-cstring` switch sets these three C compiler-style behaviors:

- Directs the preprocessor to use double quotation marks rather than the default single quotes as string delimiters for any preprocessor-generated strings. The preprocessor generates strings for predefined macros that are expressed as string constants, and as a result of the stringize operator in macro definitions (see Table 2-1 on page 2-10 for the predefined macros).

- Enables the stringize operator (#) in macro definitions. By default, the stringize operator is disabled to avoid conflicts with constant definitions (see "-stringize" on page 2-44).

- Parses identifiers using C language rules instead of assembler rules. In C, the character "." is an operator and is not considered part of an identifier. In the assembler, the "." is considered part of a directive or label. With `-cstring`, the preprocessor treats "." as an operator.

The following example shows the difference in effect of the two styles.

```
#define end last
// what label.end looks like with -cstring
label.last      // "end" parsed as ident and macro expanded

// what label.end looks like without -cstring (asm rules)
label.end       // "end" not parsed separately
```

-cs!

The `-cs!` switch directs the preprocessor to treat as a comment all text after "!" on a single line.

-cs/*

The `-cs/*` switch directs the preprocessor to treat as a comment all text within /* */ on multiple lines.

-cs//

The `-cs//` switch directs the preprocessor to treat as a comment all text after / / on a single line.

-cs{

The `-cs{` switch directs the preprocessor to treat as a comment all text within { } on multiple lines..

-csall

The `-csall` switch directs the preprocessor to accept comments in all formats.

-Dmacro[=def]

The `-Dmacro` switch directs the preprocessor to define a `macro`. If you do not include the optional definition string (`=def`), the preprocessor defines the macro as value 1. 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 _LANGUAGE_ASM as 1
```

-h[elp]

The `-help` switch directs the preprocessor to send to standard output the list of command-line switches with a syntax summary.

-i

The `-i` (less includes) switch may be used with the `-M` or `-MM` switches to direct the preprocessor to *not* output dependencies on any system files. System files are any files that are brought in using `#include < >`. Files included using `#include " "` (double quotes) are included in the dependency list.

-i|I directory

The `-Idirectory` or `-Idirectory` switch 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).

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

The preprocessor searches for included files delimited by " " in this order:

1. The source directory, that is the directory in which the original source file resides.

2. The directories in the search path supplied by the `-I` switch. If more than one directory is supplied by the `-I` switch, they are searched in the order that they appear on the command line.

3. The system directory, that is the `.../include` subdirectory of the VisualDSP++ installation directory.

ⓘ   Current directory is the directory where the source file lives, not the directory of the assembler program. Usage of full path names for the `-I` switch on the command line (omitting the disk partition) is recommended.

The preprocessor searches for included files delimited by < > in this order:

1. The directories in the search path supplied by the `-I` switch (subject to modification by the `-I-` switch, as shown in "Using the -I-Switch". If more than one directory is supplied by the `-I` switch, the directories are searched in the order that they appear on the command line.

2. The system directory, that is the `...\include` subdirectory of the VisualDSP++ installation directory.

Using the -I- Switch

The `-I-` switch indicates where to start searching for include files delimited by < >, sometimes called system include files. If there are several directories in the search path, the `-I-` switch indicates where in the path the search for system include files begins. For example,

```
pp -Idir1 -Idir2 -I- -Idir3 -Idir4 myfile.asm
```

When searching for

```
#include "inc1.h"
```

the preprocessor searches in the source directory, then `dir1`, `dir2`, `dir3`, and `dir4` in that order. When searching for

```
#include <inc2.h>
```

the preprocessor searches for the file in `dir3` and then `dir4`. The `-I-` switch marks the point where the system search path starts.

-M

The `-M` switch directs the preprocessor to output a rule (generate make rule only), 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" switch 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

The `-MM` switch 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

The `-Mo` switch specifies the name of the make dependencies file (output make rule) that the preprocessor generates when using the `-M` or `-MM` switch. The switch overrides default of make dependencies to `stdout`.

-Mt filename

The `-Mt` switch specifies the name of the target file (output make rule for the named source) for which the preprocessor generates the make rule using the `-M` or `-MM` switch. The `-Mt` *fileneme* switch overrides the default `base.is`. See "-M" for more information.

-o filename

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

-stringize

The `-stringize` switch 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 2-32) which, by default, is 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
```

-tokenize-dot

The `-tokenize-dot` switch parses identifiers using C language rules instead of assembler rules, without the need of other C semantics (see "-cstring" on page 2-39 for more information).

When the `-tokenize-dot` switch is used, the preprocessor treats "." as an operator and not as part of an identifier. If the `-notokenize-dot` switch is used, it returns the preprocessor to the default behavior. The only benefit to the negative version is that if it appears on the command line after the `-cstring` switch, it can turn off the behavior of "." without affecting other C semantics.

-Uname

The `-Uname` switch directs the preprocessor to undefine a macro on the command line. The "undefine macro" switch applies only to macros that were defined on the same command line. The functionality provides a way for users to undefine feature macros specified by the assembler or linker.

-v[erbose]

The `-v[erbose]` switch directs the preprocessor to output the version of the preprocessor program and information for each phase of the preprocessing.

-version

The `-version` switch directs the preprocessor to display the version information for the preprocessor program.

The `-version` switch 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.

-w

The `-w` (disable all warnings) switch directs the assembler not to display warning messages generated during assembly. Note that `-w` has the same effect as the `-nowarn` switch.

-Wnumber

The `-Wnumber` (warning suppression) switch selectively disables warnings specified by one or more message numbers. For example, `-W74` disables warning message `pp0074`.

-warn

The `-warn` switch generates (prints) warning messages (this switch is on by default). The `-nowarn` switch option negates this action.

I INDEX

Symbols

assembler
-Werror number, 1-149
-Winfo number, 1-149
-Wsuppress number, 1-150
-Wwarn number, 1-150
-Werror number, 1-149
-Winfo number, 1-149
-Wsuppress number, 1-150
-Wwarn number, 1-150
? preprocessor operator, 2-34

Numerics

1.0r fract, 1-54
1.15 fract, 1-53
1.31 fract, 1-53
1.31 fracts, 1-74
32-bit initialization
used with 1.31 fracts, 1-74

A

absolute address, 1-60
__AD6532__ macro, 2-13
address alignment, 1-68
ADDRESS () assembler operator, 1-49
ADI macro, 2-10
__ADSP21000__ macro, 2-10
__ADSP21020__ macro, 2-10
__ADSP21060__ macro, 2-10
__ADSP21061__ macro, 2-10

__ADSP21062__ macro, 2-10
__ADSP21065L__ macro, 2-10
__ADSP2106x__ macro, 2-11
__ADSP21160__ macro, 2-10
__ADSP21161__ macro, 2-11
__ADSP2116x__ macro, 2-11
__ADSP21261__ macro, 2-11
__ADSP21262__ macro, 2-11
__ADSP21266__ macro, 2-11
__ADSP21267__ macro, 2-11
__ADSP21363__ macro, 2-11
__ADSP21364__ macro, 2-11
__ADSP21365__ macro, 2-11
__ADSP21367__ macro, 2-11
__ADSP21368__ macro, 2-11
__ADSP21369__ macro, 2-11
__ADSPBF531__ macro, 2-12
__ADSPBF532__ macro, 2-12
__ADSPBF533__ macro, 2-12
__ADSPBF535__ macro, 2-12
__ADSPBF535__ preprocessor macro, 2-12
__ADSPBF536__ macro, 2-12
__ADSPBF537__ macro, 2-12
__ADSPBF538__ macro, 2-12
__ADSPBF539__ macro, 2-12
__ADSPBF561__ macro, 2-13
__ADSPBLACKFIN macro, 2-12
__ADSPTS101__ macro, 2-12
__ADSPTS201__ macro, 2-12
__ADSPTS202__ macro, 2-12
__ADSPTS203__ macro, 2-12
__ADSPTS__ macro, 2-12

INDEX