

VISUALDSP++[®] 3.5

Assembler and Preprocessor Manual

for SHARC Processors

Revision 4.1, March 2004

Part Number:
82-001964-01

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



Copyright Information

©2004 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

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

Trademark and Service Mark Notice

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

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

CONTENTS

PREFACE

Purpose	xi
Intended Audience	xi
Manual Contents	xii
What's New in this Manual	xii
Technical or Customer Support	xiii
Supported Processors	xiii
Product Information	xiv
MyAnalog.com	xiv
Processor Product Information	xv
Related Documents	xv
Online Technical Documentation	xvi
From VisualDSP++	xvii
From Windows	xvii
From the Web	xviii
Printed Manuals	xviii
VisualDSP++ Documentation Set	xviii
Hardware Tools Manuals	xviii

CONTENTS

Processor Manuals	xix
Data Sheets	xix
Notation Conventions	xx

ASSEMBLER

Assembler Guide	1-2
Assembler Overview	1-3
Writing Assembly Programs	1-3
Program Content	1-5
Program Structure	1-7
Program Interfacing Requirements	1-11
Using Assembler Support for C Structs	1-12
Preprocessing a Program	1-14
Using Assembler Feature Macros	1-15
Make Dependencies	1-17
Reading a Listing File	1-18
Assembler Syntax Reference	1-19
Assembler Keywords and Symbols	1-20
Assembler Expressions	1-25
Assembler Operators	1-26
Numeric Formats	1-28
Fractional Type Support	1-29
1.31 Fracts	1-30
1.0r Special Case	1-30
Fractional Arithmetic	1-30

Mixed Type Arithmetic	1-31
Comment Conventions	1-31
Conditional Assembly Directives	1-32
C Struct Support in Assembly Built-In Functions	1-34
OFFSETOF() Built-In	1-34
SIZEOF() Built-In	1-35
Struct References	1-36
Assembler Directives	1-38
.ALIGN, Specify an Address Alignment	1-41
.EXTERN, Refer to a Globally Available Symbol	1-43
.EXTERN STRUCT, Refer to a Struct Defined Elsewhere ..	1-44
.FILE, Override the Name of a Source File	1-46
.GLOBAL, Make a Symbol Globally Available	1-47
.IMPORT, Provide Structure Layout Information	1-48
.LEFTMARGIN, Set the Margin Width of a Listing File	1-50
.LIST/.NOLIST, Listing Source Lines and Opcodes	1-51
.LIST_DATA/.NOLIST_DATA, Listing Data Opcodes	1-52
.LIST_DATFILE/.NOLIST_DATFILE, Listing Data Initialization Files	1-53
.LIST_DEFTAB, Set the Default Tab Width for Listings ...	1-54
.LIST_LOCTAB, Set the Local Tab Width for Listings	1-55
.LIST_WRAPDATA/.NOLIST_WRAPDATA	1-56
.NEWPAGE, Insert a Page Break in a Listing File	1-57
.PAGELENGTH, Set the Page Length of a Listing File	1-58
.PAGEWIDTH, Set the Page Width of a Listing File	1-59

CONTENTS

.PORT, Legacy Directive	1-60
.PRECISION, Select Floating-Point Precision	1-61
.PREVIOUS, Revert to the Previously Defined Section	1-62
.ROUND_*, Select Floating-Point Rounding	1-63
.SECTION, Declare a Memory Section	1-65
Initialization Section Qualifiers	1-67
.SEGMENT & .ENDSEG, Legacy Directives	1-69
.STRUCT, Create a Struct Variable	1-70
.TYPE, Change Default Symbol Type	1-74
.VAR, Declare a Data Variable or Buffer	1-75
.VAR and ASCII String Initialization Support	1-77
.VCSE_ Optimization Directives	1-79
.WEAK, Support a Weak Symbol Definition and Reference	1-80
Assembler Command-Line Reference	1-81
Running the Assembler	1-82
Assembler Command-Line Switch Descriptions	1-84
-Dmacro[=definition]	1-86
-flags-compiler	1-86
User-Specified Defines Options	1-87
Include Options	1-87
-flags-pp -opt1 [-opt2...]	1-88
-g	1-88
-h[elp]	1-88
-i I directory	1-89

-l filename	1-89
-li filename	1-90
-M	1-90
-MM	1-90
-Mo filename	1-91
-Mt filename	1-91
-no-source-dependency	1-91
-o filename	1-92
-pp	1-92
-proc processor	1-92
-save-temps	1-93
-si-revision version	1-93
-sp	1-94
-v[erbose]	1-94
-version	1-94
-w	1-94
-Wnumber[,number]	1-94
Specifying Assembler Options in VisualDSP++	1-95

PREPROCESSOR

Preprocessor Guide	2-2
Writing Preprocessor Commands	2-3
Header Files and #include Command	2-4
Writing Macros	2-6
Using Predefined Preprocessor Macros	2-8

CONTENTS

Specifying Preprocessor Options	2-11
Preprocessor Command Reference	2-12
Preprocessor Commands and Operators	2-12
#define	2-14
Variable Length Argument Definitions	2-15
#elif	2-17
#else	2-18
#endif	2-19
#error	2-20
#if	2-21
#ifdef	2-22
#ifndef	2-23
#include	2-24
#line	2-26
#pragma	2-27
#undef	2-28
#warning	2-29
# (Argument)	2-30
## (Concatenate)	2-31
? (Generate a Unique Label)	2-32
Preprocessor Command-Line Reference	2-34
Running the Preprocessor	2-34
Preprocessor Command-Line Switches	2-35
-cstring	2-37

-cs!	2-38
-cs/*	2-38
-cs//	2-38
-cs{	2-38
-csall	2-39
-Dmacro[=def]	2-39
-h[elp]	2-39
-i	2-39
-i I directory	2-40
Using the -I- Switch	2-41
-M	2-41
-MM	2-42
-Mo filename	2-42
-Mt filename	2-42
-o filename	2-42
-stringize	2-43
-tokenize-dot	2-43
-Uname	2-43
-v[erbose]	2-43
-version	2-44
-w	2-44
-Wnumber	2-44
-warn	2-44

PREFACE

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

Purpose

The *VisualDSP++ 3.5 Assembler and Preprocessor Manual for SHARC®* (ADSP-21xxx) processors contains information about the assembler for SHARC® (ADSP-21xxx) processors which set a new standard of performance for digital signal processors, combining multiple computation units for floating-point and fixed-point processing as well as wide word widths

The manual describes how to write assembly programs for SHARC 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

This edition of *VisualDSP++ 3.5 Assembler and Preprocessor Manual for SHARC Processors* supports all SHARC processors listed in “[Supported Processors](#)”.

Refer to *VisualDSP++ 3.5 Product Release Bulletin for 32-Bit Processors* for information on all new and updated VisualDSP++ 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 dsptools.support@analog.com
- E-mail processor questions to dsp.support@analog.com
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

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

Supported Processors

The name “SHARC” refers to a family of Analog Devices, Inc. high-performance 32-bit floating-point digital signal processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++[®] currently supports the following SHARC processors:

Product Information

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		

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 means to select the information you want to receive.

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

Processor Product Information

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

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

- E-mail questions or requests for information to dsp.support@analog.com
- Fax questions or requests for information to
1-781-461-3010 (North America)
089/76 903-557 (Europe)
- Access the FTP Web site at
[ftp ftp.analog.com](ftp://ftp.analog.com) or [ftp 137.71.23.21](ftp://137.71.23.21)
<ftp://ftp.analog.com>

Related Documents

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

VisualDSP++ 3.5 Getting Started Guide for 32-Bit Processors

VisualDSP++ 3.5 User's Guide for 32-Bit Processors

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for SHARC Processors

VisualDSP++ 3.5 linker and Utilities Manual for 32-Bit Processors

VisualDSP++ 3.5 Product Release Bulletin for 32-Bit Processors

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

Product Information

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

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

<http://www.analog.com/processors/resources/technicalLibrary>

Online Technical Documentation

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

Each documentation file type is described in the following table.

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 require a browser, such as Internet Explorer 4.0 (or higher).
.PDF	VisualDSP++ tools manuals in Portable Documentation Format; one .PDF file for each manual. Viewing and printing the .PDF files require a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows[®] Explorer, or the Analog Devices Web site.

From VisualDSP++

From 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 (tool-bar buttons, menu commands, and windows).

From Windows

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

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

Using Windows Explorer

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

Using the Windows Start Button

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

Product Information

From the Web

Download manuals at the following Web site:

<http://www.analog.com/processors/resources/technicalLibrary/manuals>

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

Printed Manuals

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

VisualDSP++ Documentation Set

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

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

Hardware Tools Manuals

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

Processor Manuals

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

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. 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)** or downloaded from the Web site.

To have a data sheet faxed to you, call **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. Call the Literature Center first to find out if requested data sheets are available.

Notation Conventions

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

 Additional conventions, which apply only to specific chapters, may appear throughout this document. Code has been formatted to fit this manual's page width.

Example	Description
Close command (File menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the Close command appears on the File menu).
{this that}	Alternative items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as <i>this</i> or <i>that</i> . 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 <i>this</i> or <i>that</i> .
[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 <i>this</i> .
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
<i>filename</i>	Non-keyword placeholders appear in text with italic style format.
	This symbol indicates a note that provides supplementary information on a related topic. In the online version of this book, the word Note appears instead of this symbol.
	This symbol indicates a warning that advises on an inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word Warning appears instead of this symbol.

1 ASSEMBLER

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

The chapter contains:

- [“Assembler Guide” on page 1-2](#)
Describes the process of developing new programs in the SHARC DSP assembly language
- [“Assembler Syntax Reference” on page 1-19](#)
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 1-81](#)
Provides reference information on the assembler’s switches and conventions

Assembler Guide

The `easm21k.exe` assembler driver runs from the VisualDSP++ Integrated Debugging and Development Environment (IDDE) or from an operating system command line. The assembler processes assembly source, data, header files, and produces an object file. Assembler operations depend on two types of controls: assembler directives and assembler switches.

This section describes the process of developing new programs in the SHARC 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 1-12](#)
- [“Preprocessing a Program” on page 1-14](#)
- [“Using Assembler Feature Macros” on page 1-15](#)
- [“Make Dependencies” on page 1-17](#)
- [“Reading a Listing File” on page 1-18](#)
- [“Specifying Assembler Options in VisualDSP++” on page 1-95](#)

For information about the DSP architecture, including the DSP 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 or other), 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 .OBJ 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 1-81](#).

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 1-95](#)).

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 DSP 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.

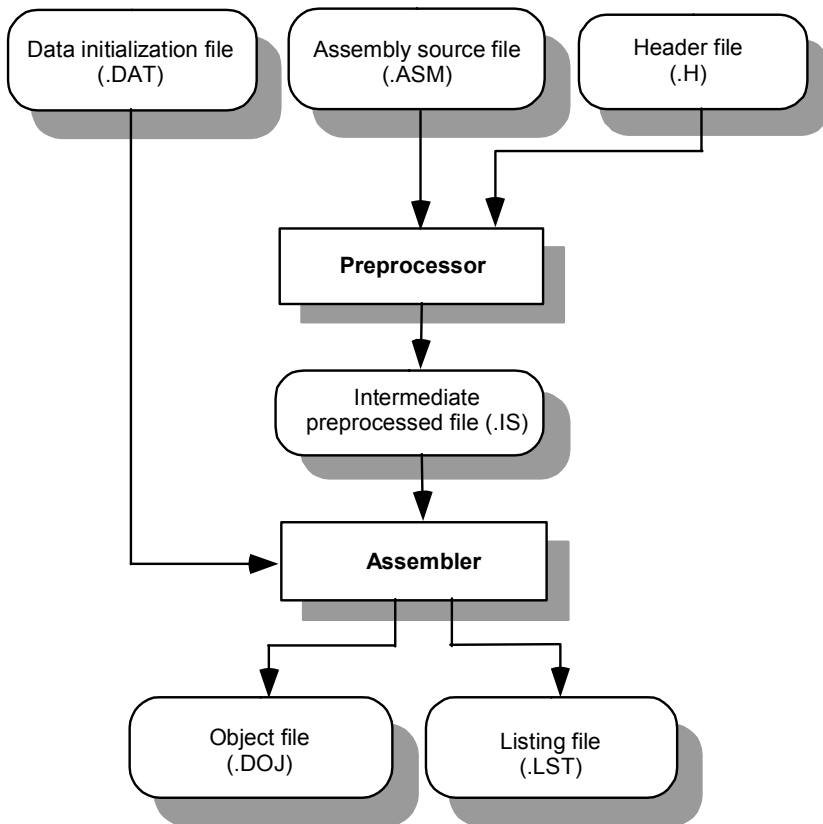


Figure 1-1. Assembler Input and Output Files

Assemble your source files from the VisualDSP++ environment or using any mechanism, such as a batch file or makefile, that will support invoking the assembler driver `eamsm21k.exe` with a specified command-line command. By default, the assembler processes an input file to produce a binary object file (`.DOJ`) and an optional listing file (`.LST`).

Object files produced by the SHARC DSP assembler may be used as input to the linker and archiver. You can archive the output of an assembly process into a library file (`.DLB`), which can then be linked with other objects

into an executable. Use the linker to combine separately assembled object files and objects from library files to produce an executable file. For more information on the linker and archiver, see the *VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors*.

A binary object file (`.OBJ`) 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 1-48](#)). 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` switch ([on page 1-90](#)) to get the information.

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

Program Content

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

Assembly Instructions

Instructions adhere to the processor's instruction set syntax documented in the processor's instruction set manual. Each instruction line must be terminated by a semicolon (;). [Figure 1-2 on page 1-9](#) shows an example assembly source file.

To mark the location of an instruction, place an address label at the beginning of an instruction line or on the preceding line. End the label with a colon (:) before beginning the instruction. Your program 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,

```
outer: DM(I1,M1) = F8;  
Outer: R0=source;
```

Assembler Directives

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



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

For example,

```
.PRECISION 40;  
.ROUND_ZERO;
```

For a complete description of the SHARC assembler’s directive set, see [“Assembler Directives” on page 1-38](#).

Preprocessor Commands

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

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

```
#include "string.h"

#define MAXIMUM 100
```

For more information, see [“Writing Preprocessor Commands” on page 2-3](#). For a list of the preprocessor commands, see [“Preprocessor Command Reference” on page 2-12](#).

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.

Suggested input section names that you could use in your assembly source appear in [Table 1-1 on page 1-7](#). 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 more information on the LDF files, see the *VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors*.

Table 1-1. Suggested Input Section Names

.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_init	A section in Program Memory that holds system initialization data

Table 1-1. Suggested Input Section Names (Cont'd)

.SECTION Name	Description
seg_rth	A section in Program Memory that holds system initialization code and interrupt service routines
seg_stak	A section in Data Memory that holds run-time stack. Required by the C run-time environment
seg_heap	A section in Data Memory that holds run-time heap. Required by the C run-time environment

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. [Figure 1-2](#) shows how a program divides into sections that match the memory segmentation of a DSP system.

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. The source program contains the following sections:

- **Data Section** — `seg_dmda`.
Variables and buffers are declared and can be initialized.
- **Program Section** — `seg_pmco`
Data, instructions, and possibly other types of statements are in this section, including statements that are needed for conditional assembly.

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

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

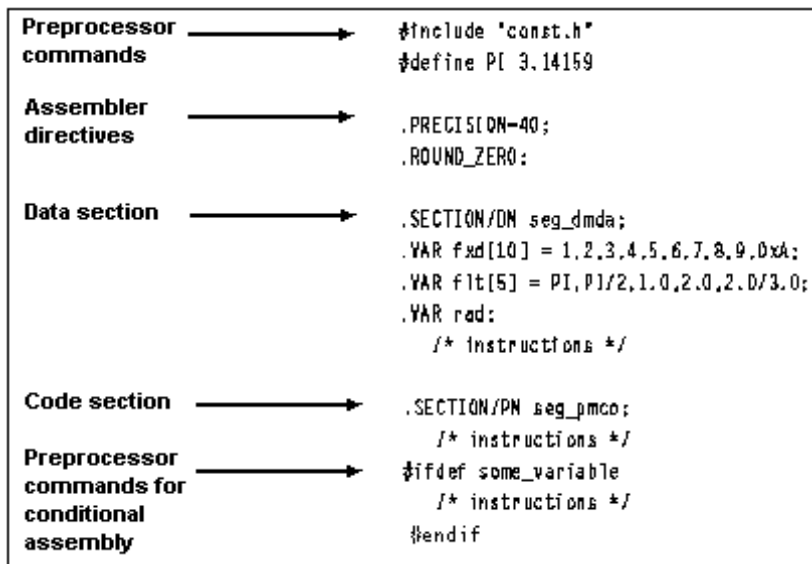


Figure 1-2. Assembly Code File Structure

[Listing 1-1](#) shows a sample user-defined Linker Description File.

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. The `SECTIONS{}` commands define the `.SECTION` placements in the system's physical memory as defined by the linker's `MEMORY{}` command.

Listing 1-1. LDF Example

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

Assembler Guide

```
$OBJECTS = $COMMAND_LINE_OBJECTS, adi_dsp\21k\lib\060_hdr.obj,  
seg_init.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 */
    .heap {
        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

```

Program Interfacing Requirements

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

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

To embed (inline) assembly code in your C or C++ program, use the `asm()` construct. To link together programs that contain C/C++ and assembly routines, use assembly interface macros. These macros facilitate the assembly of mixed routines. For more information about these methods, see the *VisualDSP++ 3.5 C/C++ Compiler and Library Manuals for SHARC 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 SHARC C/C++ compiler is provided in the *VisualDSP++ 3.5 C/C++ Compiler and Library Manuals for SHARC 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:**

<code>.IMPORT</code>	(see on page 1-48)
<code>.EXTERN STRUCT</code>	(see on page 1-44)
<code>.STRUCT</code>	(see on page 1-70)

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

<code>OFFSETOF(struct/typedef,field)</code>	(see on page 1-34)
<code>SIZEOF(struct/typedef)</code>	(see on page 1-35)

- **Struct References:**

<code>struct->field</code> (nesting supported)	(see on page 1-36)
---	-------------------------------------

For more information on C struct support, refer to the “-flags-compiler” command-line switch [on page 1-86](#) and to “Reading a Listing File” [on page 1-18](#).

C structs in assembly features accept the full set of legal C symbol names, including those that are otherwise reserved in the SHARC assembler. For example, `I1`, `I2` and `I3` are reserved keywords in the SHARC assembler, but it is legal to reference them in the context of the C struct in assembly features.

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

```
.IMPORT "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 preprocessor commands in your assembly source files. The preprocessor automatically runs before the assembler unless you use the assembler's `-sp` (skip preprocessor) switch. [Table 2-3 on page 2-13](#) 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 1-94](#)) to the assembler command line or by selecting **Verbose output** on the **Assemble** tab (property page) of the **Project Options** dialog box, accessible from the **Project** menu. See [“Specifying Assembler Options in VisualDSP++” on page 1-95](#).

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 DSP 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 2-12](#). For more information on preprocessor usage, see [“`-flags-pp -opt1 \[-opt2...\]`” on page 1-88](#)

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.

The set of feature macros include:

-D__LANGUAGE_ASM=1	Always present
-D__ADSP21000__=1	Always present
-D__ADSP21020__=1 -D__2102x__=1	Present when running <code>easm21K -proc ADSP-21020</code> with ADSP-21020 DSPs
-D__ADSP21060__=1 -D__2106x__=1	Present when running <code>easm21K -proc ADSP-21060</code> with ADSP-21060 DSPs
-D__ADSP21061__=1 -D__2106x__=1	Present when running <code>easm21K -proc ADSP-21061</code> with ADSP-21061 DSPs
-D__ADSP21062__=1 -D__2106x__=1	Present when running <code>easm21K -proc ADSP-21062</code> with ADSP-21062 DSPs
-D__ADSP21065L__=1 -D__2106x__=1	Present when running <code>easm21K -proc ADSP-21065L</code> with ADSP-21065L DSPs
-D__ADSP21160__=1 -D__2116x__=1	Present when running <code>easm21K -proc ADSP-21160</code> with ADSP-21160 DSPs
-D__ADSP21161__=1 -D__2116x__=1	Present when running <code>easm21K -proc ADSP-21161</code> with ADSP-21161 DSPs

-D__ADSP21261__=1 -D__2126x__=1	Present when running <code>easm21K -proc ADSP-21261</code> with ADSP-21261 DSPs
-D__ADSP21262__=1 -D__2126x__=1	Present when running <code>easm21K -proc ADSP-21262</code> with ADSP-21262 DSPs
-D__ADSP21266__=1 -D__2126x__=1	Present when running <code>easm21K -proc ADSP-21266</code> with ADSP-21266 DSPs
-D__ADSP21267__=1 -D__2126x__=1	Present when running <code>easm21K -proc ADSP-21267</code> with ADSP-21267 DSPs
-D__ADSP21363__=1 -D__2136x__=1	Present when running <code>easm21K -proc ADSP-21363</code> with ADSP-21363 DSPs
-D__ADSP21364__=1 -D__2136x__=1	Present when running <code>easm21K -proc ADSP-21364</code> with ADSP-21364 DSPs
-D__ADSP21365__=1 -D__2136x__=1	Present when running <code>easm21K -proc ADSP-21365</code> with ADSP-21365 DSPs

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-21062 assembly --> cc21K -21062
easm21k -proc -adsp-21262 assembly --> cc21K -21265
```



Use the `-verbose` option to verify what macro is default-defined. Refer to Chapter 1 in the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual for SHARC 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 assembler 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 VisualDSP++ requests make dependencies for the assembly, the assembler produces the dependencies from buffer initializations. The assembler also invokes the preprocessor to determine the make dependency from `#include` commands, and the compiler to determine the make dependencies from the `.IMPORT` headers.

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

```
easm21k -proc ADSP-21161 -M -l main.lst main.asm

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

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

// dependencies from the compiler for the .IMPORT headers
main.doj: .\ACME_IFir.h
main.doj: .\ADI_IAlg.h
main.doj: .\VCSE_IBase.h
main.doj: .\vcse.h
```

Reading a Listing File

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

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

The assembler listing file provides information about the imported C data structures. It tells which imports were used within the program, followed by a more detailed section. It shows the name, total size, and layout with offset for the members. The information appears at the end of the listing. You must specify the `-l listname` option (as shown [on page 1-89](#)) to get a listing file.

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 1-20](#)
- [“Assembler Expressions” on page 1-25](#)
- [“Assembler Operators” on page 1-26](#)
- [“Numeric Formats” on page 1-28](#)
- [“Comment Conventions” on page 1-31](#)
- [“Conditional Assembly Directives” on page 1-32](#)
- [“C Struct Support in Assembly Built-In Functions” on page 1-34](#)
- [“Struct References” on page 1-36](#)
- [“Assembler Directives” on page 1-38](#)

Assembler Keywords and Symbols

The assembler supports predefined keywords that include register and bit-field names, assembly instructions, and assembler directives. [Table 1-2](#) lists the assembler keywords for SHARC DSPs. 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 1-2. SHARC DSP Assembler Keywords

__ADI__	__DATE__	__FILE__	__LINE__	__STDC__
__TIME__				
.ALIGN	.ELIF	.ELSE	.ENDIF	.EXTERN
.FILE	.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
BITREV	BM	BSET	BTGL	BTSTS

Table 1-2. SHARC DSP Assembler Keywords (Cont'd)

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	IDLEI16	IF	IMASK
IMASKP	INC	IRPTL		
JUMP				

Assembler Syntax Reference

Table 1-2. SHARC DSP Assembler Keywords (Cont'd)

L0	L1	L2	L3	L4
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	LEFT0	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	NOF0	NOFZ	NOP	NOPSPECIAL
NOT	NU			
OFFSET0F	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

Table 1-2. SHARC DSP Assembler Keywords (Cont'd)

PX1	PX2			
R0	R1	R2	R3	R4
RF5	R6	R7	R8	R9
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				

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 runtimes prepend `'_'` to avoid using symbols in the user name space that begin with an alphabetic character.

- Do not use a reserved keyword to define a symbol.
- Match source and LDF sections' symbols.

Ensure that `.SECTION` name symbols do not conflict with the linker's keywords in the LDF. The linker uses sections' name symbols to place code and data in processor's memory. For more details, see the *VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors*.

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 SHARC DSPs 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.

```
.SECTION/DM seg_dmda;
.VAR xoperand;          // xoperand is a variable
.VAR input_array[10];   // input_array is a 10 element
                        // data buffer

.SECTION/PM seg_pmda;
sub_routine_1:          // sub_routine_1 is a label
.SECTION/PM kernel;     // kernel is a section
```

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 1-29](#).

Assembler Syntax Reference

Symbolic Expressions

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

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

Symbols in this type of expression are data variables, data buffers, and program labels. In the first three examples above, the symbol name represents the address of the symbol. The fourth combines that meaning of a symbol with a use of the length operator (see [Table 1-4](#)).

Assembler Operators

[Table 1-3](#) lists the assembler’s numeric and bitwise operators used in constant expressions and address expressions. These operators are listed in 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 1-32](#).

Table 1-3. Operator Precedence

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

Table 1-3. Operator Precedence (Cont'd)

Operator	Usage Description	Designation
+	Addition	Plus
-	Subtraction	Minus
<< >>	Shift left Shift right	
&	Bitwise AND (preprocessor only)	
	Bitwise inclusive OR	
^	Bitwise exclusive OR (preprocessor only)	
&&	Logical AND	
	Logical OR	

The assembler also supports special “symbol” and “length of” operators. [Table 1-4](#) lists and describes these operators used in constant and address expressions.

Table 1-4. Special Assembler Operators

Operator	Usage Description
ADDRESS(<i>symbol</i>)	Address of <i>symbol</i>
<i>symbol</i>	Address pointer to <i>symbol</i>
LENGTH(<i>symbol</i>)	Length of <i>symbol</i> in words

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

Assembler Syntax Reference

The following code 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.

```
.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 loop UNTIL LCE;

                           // loop counter=buffer's length
F0=DM(I5,M6);              // get next sample
...
loop:  ...
```

For further information on circular buffers, refer to the hardware reference manual of the target processor.

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

Numeric Formats

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

 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 “#

Table 1-5. Numeric Formats

Convention	Description
<code>0xnumber</code>	“0x” prefix indicates a hexadecimal number
<code>B#number</code> <code>b#number</code>	“B#” or “b#” prefix indicates a binary number
<code>number.number[e {+/-} number]</code>	Entry for floating-point number
<code>number</code>	No prefix and no decimal point indicates a decimal number
<code>numberr</code>	“r” suffix indicates a fractional number

(Argument)” on page 2-30 and the preprocessor’s “-stringize” command-line switch (on page 2-43), and to the assembler’s “-flags-pp -opt1 [-opt2...]” command-line switch (on page 1-88).

Fractional Type Support

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

The legal range is [-1...1). This means the values must be greater than or equal -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 ea1036] "file.asm":4 Fract constant '1.5r' is
out of range.
Fract constants must be greater than or equal to -1 and
less than 1. */
```

1.31 Fracts

The SHARC-supported fracts use 1.31 format, meaning a sign bit and “31 bits of fraction”. This is -1 to $+1-2^{-31}$. For example, 1.31 maps the constant 0.5r to 2^{-31} .

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

For example,

```
.VAR 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 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 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 1-25](#).

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/DM seg_dmda;
.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/DM seg_dmda;
.VAR myFract = 1 - 0.5r;
[Error ea1998] "fract.asm":2 Assembler Error: Illegal
mixing of types in expression.
```

Comment Conventions

The assembler supports C- and C++ -style formats for inserting comments in assembly sources. The SHARC assembler does not support nested comments. [Table 1-6](#) lists and describes assembler comment conventions.

Table 1-6. Comment Conventions

Convention	Description
<code>/* comment */</code>	A <code>“/* */”</code> string encloses a multiple-line comment
<code>// comment</code>	A pair of slashes <code>“//”</code> 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-ins `sizeof()` and `offsetof()` that return integers.

The conditional assembly directives are:

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

All conditional assembly blocks begin with an `.IF` directive and end with an `.ENDIF` directive. [Table 1-7 on page 1-34](#) shows examples of conditional directives.

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



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

The `.IF` conditional assembly directives must be used to query about C structs in assembly using the `sizeof()` and/or `offsetof()` built-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 1-34) can be used in relational expressions. Different code sequences can be included based on the result of the expression. For example, a `sizeof(struct/type-def/C base type)` is permitted.

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](#)”).
- 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>
.ENDIF;
```

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

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

Table 1-7. Relational Operators for Conditional Assembly

Relational Operators	Conditional Directive Examples
not !	.if !0;
greater than >	.if (sizeof(myStruct) > 16);
greater than equal >=	.if (sizeof(myStruct) >= 16);
less than <	.if (sizeof(myStruct) < 16);
less than equal <=	.if (sizeof(myStruct) <= 16);
equality ==	.if (8 == sizeof(myStruct));
not equal !=	.if (8 != sizeof(myStruct));
logical or	.if (2 !=4) (5 == 5);
logical and &&	.if (sizeof(char) == 2 && sizeof(int) == 4);

C Struct Support in Assembly Built-In Functions

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

OFFSETOF() Built-In

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

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

where:

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

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

sizeof() Built-In

The `sizeof()` built-in 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`, etc.)
- A member of a structure (any type)

For example,

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



The `sizeof()` built-in function returns the size in the units appropriate for its processor. For SHARC processors, units are in words.

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

```

.EXTERN STRUCT myStructTag _myStruct;
IO =_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

```

// in assembly file
.EXTERN STRUCT myStructTagWithPtr _myStructWithPtr;
B0 = _myStructWithPtr->pNestedOne;
M0 = OFFSETOF(Location,State);
R0 = DM(M0,I0);

```

Assembler Directives

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

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

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

The SHARC assembler supports the directives shown in [Table 1-8](#). A description of each directive appears in the following sections.

Table 1-8. Assembler Directive Summary

Directive	Description
.ALIGN (see on page 1-41)	Specifies an alignment requirement for data or code
.ENDSEG (see on page 1-69)	Legacy directive. Marks the end of a section. Used with legacy directive .SEGMENT that begins a section.
.EXTERN (see on page 1-43)	Allows reference to a global symbol
.EXTERN STRUCT (see on page 1-44)	Allows reference to a global symbol (struct) that was defined in another file
.FILE (see on page 1-46)	Overrides filename given on the command line. Used by C compiler
.GLOBAL (see on page 1-47)	Changes a symbol's scope from local to global
.IMPORT (see on page 1-48)	Provides the assembler with the structure layout (C struct) information

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

Directive	Description
<code>.LEFTMARGIN</code> (see on page 1-50)	Defines the width of the left margin of a listing
<code>.LIST</code> (see on page 1-51)	Starts listing of source lines
<code>.LIST_DATA</code> (see on page 1-52)	Starts listing of data opcodes
<code>.LIST_DATFILE</code> (see on page 1-53)	Starts listing of data initialization files
<code>.LIST_DEFTAB</code> (see on page 1-54)	Sets the default tab width for listings
<code>.LIST_LOCTAB</code> (see on page 1-55)	Sets the local tab width for listings
<code>.LIST_WRAPDATA</code> (see on page 1-56)	Starts wrapping opcodes that don't fit listing column
<code>.NEWPAGE</code> (see on page 1-57)	Inserts a page break in a listing
<code>.NOLIST</code> (see on page 1-51)	Stops listing of source lines
<code>.NOLIST_DATA</code> (see on page 1-52)	Stops listing of data opcodes
<code>.NOLIST_DATFILE</code> (see on page 1-53)	Stops listing of data initialization files
<code>.NOLIST_WRAPDATA</code> (see on page 1-56)	Stops wrapping opcodes that don't fit listing column
<code>.PAGELENGTH</code> (see on page 1-58)	Defines the length of a listing page
<code>.PAGEWIDTH</code> (see on page 1-59)	Defines the width of a listing page
<code>.PORT</code> (see on page 1-60)	Legacy directive. Declares a memory-mapped I/O port.

Assembler Syntax Reference

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

Directive	Description
<code>.PRECISION</code> (see on page 1-61)	Defines the number of significant bits in a floating-point value.
<code>.PREVIOUS</code> (see on page 1-62)	Reverts to a previously described <code>.SECTION</code>
<code>.ROUND_NEAREST</code> (see on page 1-63)	Specifies the Round-to-Nearest mode.
<code>.ROUND_MINUS</code> (see on page 1-63)	Specifies the Round-to-Negative Infinity mode.
<code>.ROUND_PLUS</code> (see on page 1-63)	Specifies the Round-to-Positive Infinity mode.
<code>.ROUND_ZERO</code> (see on page 1-63)	Specifies the Round-to-Zero mode.
<code>.SECTION</code> (see on page 1-65)	Marks the beginning of a section
<code>.SEGMENT</code> (see on page 1-69)	Legacy directive. Replaced with the <code>.SECTION</code> directive.
<code>.STRUCT</code> (see on page 1-70)	Defines and initializes data objects based on C typedefs from <code>.IMPORT C</code> header files
<code>.TYPE</code> (see on page 1-74)	Changes the default data type of a symbol; used by C compiler
<code>.VAR</code> (see on page 1-75)	Defines and initializes 32-bit data objects
<code>.VCSE_METHODCALL_START</code> <code>.VCSE_METHODCALL_END</code> <code>.VCSE_METHODCALL_RETURN</code> (see on page 1-79)	Used as optimization directives for VCSE components
<code>.WEAK</code> (see on page 1-80)	Creates a weak definition or reference

.ALIGN, Specify an Address Alignment

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

Refer to Chapter 2 “Linker” in the *VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors* 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/DM seg_dmda;
.ALIGN 2;
.VAR single;
```

Assembler Syntax Reference

```
        /* aligns the data item in DM 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 in DM on the double-  
        word boundary, at the location with the address  
        value that can be evenly divided by 4; advances  
        other data items consecutively */
```

.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 [on page 1-47](#).

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 `_` is necessary.

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

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

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

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

Example:

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

.FILE, Override the Name of a Source File

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

Syntax:

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

where

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

Example:

```
.FILE "vect.c";           // the argument may be a *.c file
.SECTION/DM seg_dmda;
...
...
```

.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. The `.GLOBAL` directive is used to make an identifier global to all files.

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. Local symbols in different files can have the same name, and the linker considers them to be independent entities. Global symbols are recognizable from other file; 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 1-43](#). 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:

```
.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
```

.IMPORT, Provide Structure Layout Information

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

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

Syntax:

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

where

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

The `.IMPORT` directive does not allocate space for a variable of this type—that requires the `.STRUCT` directive (see [on page 1-70](#)).

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 1-86](#)) 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 1-43](#) and the `.STRUCT` directive [on page 1-70](#).

Example:

```
.IMPORT "CHeaderFile.h";
.IMPORT "ACME_IIfir.h", "ACME_IFir.h";
.SECTION/PM seg_pmco;
    // ... code that uses CHeaderFile, ACME_IIfir, and
    // ACME_IFir C structs
```

.LEFTMARGIN, Set the Margin Width of a Listing File

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

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

Syntax:

```
.LEFTMARGIN expression;
```

where

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

Example:

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



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

.LIST/.NOLIST, Listing Source Lines and Opcodes

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

If `.NOLIST` is in effect, no lines in the current source, or any nested source, 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 [on page 1-55](#)).

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.
A value of 0 sets the current default tab width to the assembler's initial default tab width. In the absence of a `.LIST_DEFTAB` directive, the default tab width defaults to 4.

Example:

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

.LIST_LOCTAB, Set the Local Tab Width for Listings

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

The `.LIST_LOCTAB` directive sets the local tab width, and the `.LIST_DEFTAB` directive sets the default tab width (see [on page 1-54](#)).

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

Syntax:

```
.LIST_LOCTAB expression;
```

where

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

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

Example: See the `.LIST_DEFTAB` example [on page 1-54](#).

.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.

.NEWPAGE, Insert a Page Break in a Listing File

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

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

Syntax:

```
.NEWPAGE;
```

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

.PAGELength, Set the Page Length of a Listing File

The `.PAGELength` directive controls the page length of the listing file produced by the assembler when you use the `-l` switch ([on page 1-89](#))

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
```



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 the `LEFTMARGIN` value plus about 66. You cannot set this integer to be less than 66. 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 66.

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.
```



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

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` is a globally available port symbol.

Example:

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

To declare a port using the SHARC assembler 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++ 3.5 Linker and Utilities Manual for 32-Bit Processors*.

.PRECISION, Select Floating-Point Precision

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 DSP 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. */
```

-  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 1-63](#)) 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/PM sec_one;
.SECTION/DM sec_two;    // instructions
.PREVIOUS;              // previous section ends, back to sec_one
.PREVIOUS;              // no previous section to set to
```

Example of Legal Directive Use

```
.SECTION/PM sec_one;    // data & instructions
.SECTION/DM sec_two;    // data
.PREVIOUS;              // data & instructions
```

evaluates as:

```
.SECTION/PM sec_one;    // instructions
.SECTION/DM sec_two;    // data
.SECTION/PM sec_one;    // instructions
```

.ROUND_*, Select Floating-Point Rounding

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 1-61](#)).

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 DSP 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).
```

Assembler Syntax Reference

```
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.

Syntax:

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

where

- The `/type` keyword specifies the size of a data in the section. This data mapping should follow from the chip's memory architecture. The `type` must match the memory type of the input section of the same name used by the `.LDF` file to place the section. One of the following types is required for each `.SECTION` directive:

Table 1-9. Memory and Section Types

Memory/Section Type	Description
PM	Section contains instructions and possibly data, in 48-bit words.
DM	Section contains data in 40-bit words.
DATA64	Section defines data in 64-bit words..

- `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 `.\21k\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 SHARC assembler implicitly pre-fix the name of the section with the “.rela.” string to form a relocatable section. For example, for the sections named `rela.seg_dmda` and `rela.seg_pmco`, the relocation section are `.rela.rela.seg_dmda` and `.rela.rela.seg_pmco` respectively. To avoid ambiguity, ensure that your sections’ names do not begin with “.rela.”.

- *sectionType*—an optional ELF section type identifier. The assembler uses the default `SHT_PROGBITS` when this identifier is absent. Valid *sectionTypes* are described in the `ELF.h` header file, which is available from third-party software development kits. For more information on the ELF file format, see the *VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors*.

Example:

```
/* Data section and program section declared below correspond
   to the default LDF's input sections. */
.SECTION/DM seg_dmda;      // data section
...
.SECTION/PM seg_pmco;      // program section
...
```



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

Initialization Section Qualifiers

The `.SECTION` directive may identify “how/when/if” a section is initialized. The qualifiers 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 `-meminit` switch is not used, the memory is initialized at “load” time when the `.DXX` 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];
```

Assembler Syntax Reference

```
.VAR zz = 25;    // [Warning eal141] "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.

.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++ 3.5 Linker and Utilities Manual for 32-Bit Processors*.

.SEGMENT & .ENDSEG, Legacy Directives

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 DSP assembler.

.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:

- Supplies 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 continues to offset to the correct location for the specified initialization and zero-initializes the new member.

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

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

then `member2` is initialized to zero 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 DSP 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/DM seg_dmda;
.VAR buffer [] = {1,2,3,4};
```

Assembler Syntax Reference

```
.SECTION/PM seg_pmco;  
    LO = LENGTH( buffer );    // Returns 4
```

Syntax:

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

```
.VAR varName1[,varName2,...];  
.VAR = initExpression1, initExpression2,...;  
.VAR varName1 = initexpression1 [,varName2 = initexpression2,...];  
.VAR bufferName[] = {initExpression1, initExpression2,...};  
.VAR bufferName[] = {"fileName"};  
.VAR bufferName[length] = "fileName";  
.VAR bufferName1[length] [,bufferName2[length],...];  
.VAR bufferName[length] = initExpression1,initExpression2,...;
```

where:

- *varName* —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.

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

```
.VAR buf1=0x1234, buf2=0x5678, ...;
    // 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 and ASCII String Initialization Support

The assembler supports ASCII string initialization. This allows the full use of the ASCII character set, including digits and special characters. The characters are stored in the upper byte of 32-bit words. The LSBs are cleared. The assembler also accepts ASCII characters within comments.

Assembler Syntax Reference

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.

.VCSE_ Optimization Directives

The `.VCSE_` directives are the optimization directives for VisualDSP++ Component Software Engineering (VCSE) components. You are able to see them generated in the VCSE assembler code for the purposes of providing the linker with sufficient information to enable space efficient and speed optimizations that are otherwise missed.

The `.VCSE_METHODCALL_START` and `.VCSE_METHODCALL_END` directives mark VCSE methods for linker code/data elimination. The linker is provided the interface name and actual offset of the corresponding entry in the method table.

The `.VCSE_RETURNS` directive is used for marking VCSE constant methods.

.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.



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 1-82](#)
- [“Assembler Command-Line Switch Descriptions” on page 1-84](#)

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 1-95](#) for more information.

Running the Assembler

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

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

runs the assembler with

<code>easm21K</code>	Name of the assembler program for SHARC processors.
<code>-switch</code>	Switch (or switches) to process. The command-line interface offers many optional switches that select operations and modes for the assembler and pre-processor. Some assembler switches take a file name as a required parameter.
<code><i>sourceFile</i></code>	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 1-10 summarizes file extension conventions accepted by the VisualDSP++ environment.

Table 1-10. 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

```
easm21K -proc ADSP-21161 -l pList.lst -Dmax=100 -v -o bin\p1.doj p1.asm
```

runs the assembler with

-proc ADSP-21161—specifies assembles instructions unique to ADSP-21161 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 1-11](#). Detailed description of each assembler switch starts [on page 1-86](#).

Table 1-11. Assembler Command-Line Switch Summary

Switch Name	Purpose
<code>-Dmacro[=definition]</code> (on page 1-86)	Passes macro definition to the preprocessor.
<code>-flags-compiler -opt1 [, -opt2...]</code> (on page 1-86)	Passes each comma-separated option to the compiler. (Used when compiling <code>.IMPORT C</code> header files.)
<code>-flags-pp -opt1 [, -opt2...]</code> (on page 1-88)	Passes each comma-separated option to the preprocessor.
<code>-g</code> (on page 1-88)	Generates debug information (DWARF-2 format).
<code>-h[elp]</code> (on page 1-88)	Outputs a list of assembler switches.
<code>-i -I directory pathname</code> (on page 1-89)	Searches a directory for included files.
<code>-l filename</code> (on page 1-89)	Outputs the named listing file.
<code>-li filename</code> (on page 1-90)	Outputs the named listing file with <code>#include</code> files expanded.
<code>-M</code> (on page 1-90)	Generates make dependencies for <code>#include</code> and data files only; does not assemble. For example, <code>-M</code> suppresses the creation of an object file.
<code>-MM</code> (on page 1-90)	Generates make dependencies for <code>#include</code> and data files. Use <code>-MM</code> for make dependencies with assembly.
<code>-Mo filename</code> (on page 1-91)	Writes make dependencies to the <code>filename</code> specified. The <code>-Mo</code> option is for use with either the <code>-M</code> or <code>-MM</code> option. If <code>-Mo</code> is not present, the default is <code><stdout></code> display.

Table 1-11. Assembler Command-Line Switch Summary (Cont'd)

Switch Name	Purpose
<code>-Mt filename</code> (on page 1-91)	Specifies the make dependencies target name. The <code>-Mt</code> option is for use with either the <code>-M</code> or <code>-MM</code> option. If <code>-Mt</code> is not present, the default is base name plus 'DOJ'.
<code>-no-source-dependency</code> (on page 1-91)	Suppresses output of the source filename in the dependency output produced when <code>"-M"</code> or <code>"-MM"</code> has been specified.
<code>-o filename</code> (on page 1-92)	Outputs the named object [binary] file.
<code>-pp</code> (on page 1-92)	Runs the preprocessor only; does not assemble.
<code>-proc processor</code> (on page 1-92)	Specifies a target processor for which the assembler should produce suitable code.
<code>-save-temps</code> (on page 1-93)	Saves intermediate files
<code>-si-revision version</code> (on page 1-93)	Specifies silicon revision of the specified processor.
<code>-sp</code> (on page 1-94)	Assembles without preprocessing.
<code>-v[erbose]</code> (on page 1-94)	Displays information on each assembly phase.
<code>-version</code> (on page 1-94)	Displays version information for the assembler and preprocessor programs.
<code>-w</code> (on page 1-94)	Disables all assembler-generated warnings.
<code>-Wnumber[,number ...]</code> (on page 1-94)	Selectively disables warnings by one or more message numbers. For example, <code>-W1092</code> disables warning message <code>ea1092</code> .

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

-Dmacro[=definition]

The `-D` (define macro) switch directs the assembler to define a macro and pass it to the preprocessor. See [“Using Assembler Feature Macros” on page 1-15](#) for the list of predefined macros. For example,

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

-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";
easm21K -proc ADSP-21161 -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";
easm21k -proc ADSP-21062 -DaDef -flags-compiler -DbDef -DbDefTwo=2. file.asm
// -DaDef is not passed to the compiler
cc21k -proc ADSP-21062 -c -debug-types -DbDef -DbDefTwo=2 myHeader.h
```



See [“Using Assembler Feature Macros” on page 1-15](#) for the list of predefined macros, including default macros.

Include Options

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

For example,

```
easm21K -proc ADSP-21161 -I\Path -DaDef -flags-compiler -I\cPath,-I. file.asm
cc21k -proc ADSP-21161 -I\Path -DaDef -flags-compiler -I\cPath,-I. myHeader.h
```

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

Assembler Command-Line Reference

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

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

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



Use `-flags-pp` with caution. For example, if 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 [on page 1-94](#)).

-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. Current project directory (.DPJ)
2. ...\\21K\\include subdirectory of the VisualDSP++ installation directory
3. Specified directory (or list of directories). The order of the list defines the order of multiple searches.

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

For example,

```
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,

```
easm21K -proc ADSP-21161 -I\\path -I. -l file.lst file.asm
```

Assembler Command-Line Reference

The file name is a required argument to the `-l` option. For more information, see [“Reading a Listing File” on page 1-18](#).

-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 1-18](#).

-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.

-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 must be used in conjunction with the `-M` or `-MM` options (see [on page 1-90](#)).

Assembler Command-Line Reference

-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:

```
easm21K -proc ADSP-21161 -pp -o test1.is test.asm
// preprocessed output goes into test1.is

easm21K -proc ADSP-21161 -o "C:\bin\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. For SHARC processors, the *processor* identifier has an “ADSP-21xxx” format.

The *processor* identifiers directly supported by VisualDSP++ 3.5 are listed in [“Supported Processors”](#).

For example,

```
easm21K -proc ADSP-21161 -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.



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 defines the silicon revision for a particular processor. The parameter “*version*” represents a silicon revision of the processor specified by the `-proc` switch ([on page 1-92](#)).

The supported revisions are “0.0”, “0.1”, “1.0” and “none”. The assembler uses this information when dealing with issues which are specific to a particular silicon revision of a processor. If “none” is used, these issues are ignored. If the `-si-revision` option is not used, the assembler assumes the latest silicon revision known to it.

The assembler sets the `__SILICON_REVISION__` macro to 0x0 for revision “0.0”, 0x1 for revision “0.1” and 0x100 for revision “1.0”. The assembler does not set the `__SILICON_REVISION__` macro for “none”.

Assembler Command-Line Reference

If the revision set is not known to the assembler, it assumes the latest known revision. The `__SILICON_REVISION__` macro is then set using two hexadecimal digits each for the major and minor numbers in the revision. For example, 9.0 becomes 0x900 and 10.21 becomes 0xa15.

-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 (`.i.s`) is created.

-v[erbose]

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

-version

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

-w

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

-Wnumber[,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 ...]`.

Specifying Assembler Options in VisualDSP++

When using the VisualDSP++ IDDE, use the **Assemble** tab from the **Project Options** dialog box to set assembler functional options.

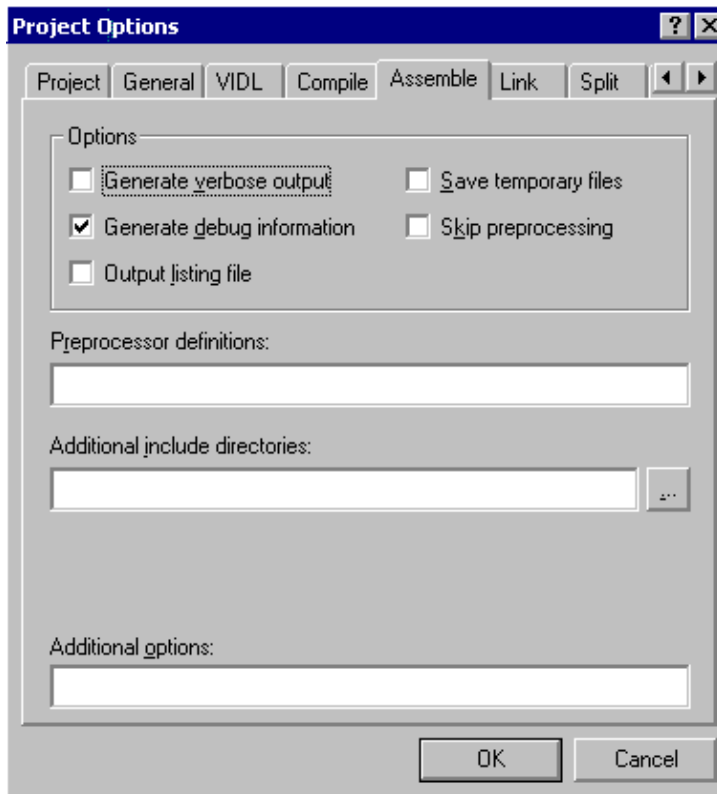


Figure 1-3. Project Options – Assemble Tab

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

Assembler Command-Line Reference

Callouts in [Figure 1-3](#) refer to the corresponding assembler command-line switches described in [“Assembler Command-Line Switch Descriptions” on page 1-84](#).

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

The assembler options apply to directing calls to `eam21k` when assembling `*.asm` files. Changing assembler options in VisualDSP++ does not affect the assembler calls made by the compiler during the compilation of `*.c`/`*.cpp` files.

2 PREPROCESSOR

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

The preprocessor is run by other build tools (assembler and linker) from the operating system’s command line or within the VisualDSP++ 3.5 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” on page 2-2](#)
Contains the information on building programs
- [“Preprocessor Command Reference” on page 2-12](#)
Describes the preprocessor’s commands, with syntax and usage examples
- [“Preprocessor Command-Line Reference” on page 2-34](#)
Describes the preprocessor’s command-line switches, with syntax and usage examples

Preprocessor Guide

This section contains the `pp` preprocessor information on how to build programs from a command line or from the VisualDSP++ 3.0 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 2-8](#)
- [“Specifying Preprocessor Options” on page 2-11](#)

The compiler has its 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++ 3.5 C/C++ Compiler and Library Manuals* for the target DSPs.

The assembler preprocessor differs from the ANSI C standard preprocessor in several ways. First, the assembler preprocessor supports a “?” operator (see [on page 2-32](#)) 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 [“Preprocessor Command Reference” on page 2-12](#).

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\21K\include` folder.

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:

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

1. include path specified by the `-I` switch
2. ...\\VisualDSP\\21K\\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\\...21K\\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 [“#include” on page 2-24](#).

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.

Example:

```
#define false 0
#define ccall(x) \
    r2=i6; i6=i7; \
    jump (pc, x) (db); \
    dm(i7,m7)=r2; \
    dm(i7, m7)=PC
```

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,

```
#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 2-14](#).

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.



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 1-77](#)).

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

Table 2-1. Predefined Preprocessor Macros

Macro	Definition
ADI	Defines ADI as 1.
__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'.
__STDC__	Defines __STDC__ as 1.
__TIME__	Defines __TIME__ as current time in the 24-hour format 'hh:mm:ss', for example, '06:54:35'.
__DATE__	Defines __DATE__ as current date in the format 'Mm dd yyyy', for example, 'Oct 02 2000'.

Table 2-2. Feature Preprocessor Macros

Macro	Definition
__ADSP21000__	Always 1 for SHARC DSP 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. Feature Preprocessor Macros (Cont'd)

Macro	Definition
__ADSP21161__	Present when running <code>easmts -proc ADSP-21161</code> with ADSP-21161 processor
__ADSP2106x__	Present when running <code>easmts -proc ADSP-2106x</code> with ADSP-2106x processor
__ADSP2116x__	Present when running <code>easmts -proc ADSP-2116x</code> with ADSP-2116x processor
__ADSP21261__	Present when running <code>easmts -proc ADSP-21261</code> with ADSP-21261 processor
__ADSP21262__	Present when running <code>easmts -proc ADSP-21262</code> with ADSP-21262 processor
__ADSP21266__	Present when running <code>easmts -proc ADSP-21266</code> with ADSP-21266 processor
__ADSP21267__	Present when running <code>easmts -proc ADSP-21267</code> with ADSP-21267 processor
__ADSP21363__	Present when running <code>easmts -proc ADSP-21363</code> with ADSP-21363 processor
__ADSP21364__	Present when running <code>easmts -proc ADSP-21364</code> with ADSP-21364 processor
__ADSP21365__	Present when running <code>easmts -proc ADSP-21365</code> with ADSP-21365 processor
_LANGUAGE_ASM	Always set to 1
_LANGUAGE_C	Equal 1 when used for C compiler calls to specify <code>.IMPORT</code> headers. Replaces <code>_LANGUAGE_ASM</code> .

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 DSP tools can receive input for the preprocessor program and direct its operation. The way the preprocessor options are set depends on the environment used to run the DSP development software.

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

- From the operating system command line, select the preprocessor's command-line switches. For more information on these switches, see [“Preprocessor Command-Line Switches” on page 2-35](#).
- 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 1-95](#) for the **Assemble** tab.

For more information, see the *VisualDSP++ 3.5 User's Guide for 32-Bit Processors* and online Help.

Preprocessor Command Reference

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

The preprocessor reads code from a source file (`.ASM` 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
- Can contain comments with the backslash character (`\`)
- 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 2-3](#) lists the preprocessor command set. [Table 2-4](#) lists the preprocessor operator set. Sections that begin [on page 2-14](#) describe each of the preprocessor commands and operators.

Table 2-3. Preprocessor Command Summary

Command/Operator	Description
<code>#define</code> (on page 2-14)	Defines a macro
<code>#elif</code> (on page 2-17)	Subdivides an <code>#if ... #endif</code> pair
<code>#else</code> (on page 2-18)	Identifies alternative instructions within an <code>#if ... #endif</code> pair
<code>#endif</code> (on page 2-19)	Ends an <code>#if ... #endif</code> pair
<code>#error</code> (on page 2-20)	Reports an error message
<code>#if</code> (on page 2-21)	Begins an <code>#if ... #endif</code> pair
<code>#ifdef</code> (on page 2-22)	Begins an <code>#ifdef ... #endif</code> pair and tests if macro is defined
<code>#ifndef</code> (on page 2-23)	Begins an <code>#ifndef ... #endif</code> pair and tests if macro is not defined
<code>#include</code> (on page 2-24)	Includes contents of a file
<code>#line</code> (on page 2-26)	Sets a line number during preprocessing
<code>#pragma</code> (on page 2-27)	Takes any sequence of tokens
<code>#undef</code> (on page 2-28)	Removes macro definition
<code>#warning</code> (on page 2-29)	Reports a warning message

Table 2-4. Preprocessor Operator Summary

Command/Operator	Description
<code>#</code> (on page 2-30)	Converts a macro argument into a string constant. By default, this operator is OFF. Use the command-line switch “-stringize” on page 2-43 to enable it.
<code>##</code> (on page 2-31)	Concatenates two tokens
<code>?</code> (on page 2-32)	Generates unique labels for repeated macro expansions
<code>...</code> (on page 2-15)	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) \
r0=DM(src); \
PM(dest)=r0
/* Defines a macro named copy with two arguments.
   The definition includes two instructions that copy a
   word from data memory to program memory.
   For example,
       copy(0x3f,0xC0);
   calls the macro, passing parameters to it.

   The preprocessor replaces the macro with the code:
       r0=DM(0x3f);
       PM(0xC0)=r0
*/
```

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,

<code>test(1)</code>	Error; the macro must have at least one more argument than formal parameters, not counting "..."
<code>test(1,2)</code>	Valid entry
<code>test(1,2,3,4,5)</code>	Valid entry

Preprocessor Command Reference

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

For example,

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

expands as

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

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

#elif

The `#elif` command (else if) is used within an `#if ... #endif` pair. The `#elif` includes an alternative condition to test when the initial `#if` condition evaluates as `FALSE`. The preprocessor tests each `#elif` condition inside the pair and processes instructions that follow the first true `#elif`. There can be an unlimited number of `#elif` commands inside one `#if ... #endif` 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 if
    the test is true and excludes all alternatives within
    #if ... elif pair. */
#else
...
#endif
```

Preprocessor Command Reference

#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. */
```

Preprocessor Command Reference

#error

The `#error` command causes the preprocessor to raise an error. The preprocessor uses the text following the `#error` command as the error message.

Syntax:

```
#error messageText
```

where

messageText—user-defined text

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

Example:

```
#ifndef __ADSP21062__
#error \
    MyError:\
    Expecting a ADSP-21062 . \
    Check the Linker Description File!
#endif
```


#if

The `#if` command begins an `#if ... #endif` pair. Statements inside an `#if ... #endif` pair can include other preprocessor commands and conditional expressions. The preprocessor processes instructions inside the `#if ... #endif` pair only when *condition* that follows the `#if` evaluates as TRUE. Every `#if` command must be terminated with an `#endif` command.

Syntax:

```
#if condition
```

where

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

Example:

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

More examples:

```
#if (x!=100) && (y==20)
...
#endif
#if defined(__ADSP21062__)
...
#endif
```

Preprocessor Command Reference

#ifdef

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

Syntax:

```
#ifdef macroSymbol
```

where

macroSymbol—macro identifying symbol

Example:

```
#ifdef __ADSP21062__  
    /* Includes text after #ifdef only when __ADSP21062__ 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. The number of `#ifndef` commands must equal the number of `#endif` commands.

Syntax:

```
#ifndef macroSymbol
```

where

macroSymbol—macro identifying symbol

Example:

```
#ifndef __ADSP21062__  
    /* Includes text after #ifndef only when __ADSP21062__ has  
    been 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 preprocessor. After macro expansion, the text must match one of the header file forms.

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

- System Header `<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 __ADSP21062__
    /* Tests that __ADSP21062__ has been defined */
#include <stdlib.h>

#endif
```

Preprocessor Command Reference

#line

The `#line` command directs the preprocessor to set the internal line counter to the specified value. Use this command for error tracking purposes.

Syntax:

```
#line lineNumber "sourceFile"
```

where

lineNumber—number of the source line that you want to output

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

Example:

```
#line 7 "myFile.c"
```



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

#pragma

The `#pragma` is the implementation-specific command that 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 pragmas; therefore, it ignores any information in the `#pragma`.

Syntax:

```
#pragma any_sequence_of_tokens
```

Example:

```
#pragma disable_warning 1024
```

Preprocessor Command Reference

#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 backslash character (\) at the end of each line except for the last line.

Example:

```
#ifndef __ADSP21062__  
#warning \  
    MyWarning: \  
    Expecting a ADSP-21062. \  
    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 2-43](#)) 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 [on page 2-9](#)). 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 compiler from the assembler or linker. You can also run the preprocessor independently from its own command line.

This section contains:

- [“Running the Preprocessor”](#)
- [“Preprocessor Command-Line Switches” on page 2-35](#)

Running the Preprocessor

To run the preprocessor from the command line, type the name of the program followed by arguments in any order.

```
pp [ -switch1 [-switch2 ... ]] [sourceFile]
```

where

<code>pp</code>	Name of the preprocessor program.
<code>-switch</code>	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.
<code><i>sourceFile</i></code>	Name of the source file to process. The preprocessor supports relative and absolute path names. The <code>pp.exe</code> outputs a list of command-line switches when runs without this argument..

For example, the following command line

```
pp -Dfilter_taps=100 -v -o bin\p1.is p1.asm
```

runs the preprocessor with

`-Dfilter_taps=100`—defines the macro `filter_taps` as equal to 100

`-v`—displays verbose information for each phase of the preprocessing

`-o bin\p1.is`—specifies the name and directory for the intermediate preprocessed file

`p1.asm`—specifies the assembly source file to preprocess



Most switches without arguments can be negated by prepending `-no` to the switch; for example, `-nowarn` turns off warning messages, and `-nocsl` turns off omitting “!” style comments.

Preprocessor Command-Line Switches

The preprocessor is controlled through the switches (or VisualDSP++ options) of other DSP development tools, such as the compiler, assembler, and linker. Note that the preprocessor (`pp.exe`) can operate independently from the command line with its own command-line switches.

[Table 2-5](#) lists the `pp.exe` switches. A detailed description of each switch appears beginning [on page 2-37](#).

Table 2-5. Preprocessor Command-Line Switch Summary

Switch Name	Description
<code>-cstring</code> on page 2-37	Enables the stringization operator and provides C compiler-style preprocessor behavior
<code>-csl</code> on page 2-38	Treats as a comment all text after “!” on a single line
<code>-cs/*</code> on page 2-38	Treats as a comment all text within <code>/* */</code>

Preprocessor Command-Line Reference

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

<code>-cs//</code> on page 2-38	Treats as a comment all text after <code>/ /</code>
<code>-cs{</code> on page 2-38	Treats as a comment all text within <code>{ }</code>
<code>-csall</code> on page 2-39	Accepts comments in all formats
<code>-Dmacro[=definition]</code> on page 2-39	Defines <i>macro</i>
<code>-h[elp]</code> on page 2-39	Outputs a list of command-line switches
<code>-i</code> on page 2-39	Outputs only makefile dependencies for <code>include</code> files specified in double quotes
<code>-i Idirectory</code> on page 2-40	Searches <i>directory</i> for included files
<code>-M</code> on page 2-41	Makes dependencies only
<code>-MM</code> on page 2-42	Makes dependencies and produces preprocessor output
<code>-Mo filename</code> on page 2-42	Specifies <i>filename</i> for the make dependencies output file
<code>-Mt filename</code> on page 2-42	Makes dependencies for the specified source file
<code>-o filename</code> on page 2-42	Outputs named object file
<code>-stringize</code> on page 2-43	Enables stringization (includes a string in double quotes)
<code>-tokenize-dot</code> (on page 2-43)	Treats “.” (dot) as an operator when parsing identifiers
<code>-Uname</code> on page 2-43	Undefines a macro on the command line
<code>-v[erbose]</code> on page 2-43	Displays information about each preprocessing phase

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

<code>-version</code> on page 2-44	Displays version information for the preprocessor
<code>-w</code> (on page 2-44)	Removes all preprocessor-generated warnings
<code>-Wnumber</code> (on page 2-44)	Suppresses any report of the specified warning
<code>-warn</code> (on page 2-44)	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 2-9](#) 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 2-43](#)).

Preprocessor Command-Line Reference

- 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 `{ }`.

-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 assembly language 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 [on page 2-24](#)).



Note that no space is allowed between `-i` or `-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 "incl.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.

Preprocessor Command-Line Reference

-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. If the named file is not in the current directory, you must provide the path name in the double quotation marks (“ ”). The the “[-o filename](#)” 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 `-M filename` switch overrides the default `base.is`. See “[-M](#)” for more information.

-o filename

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

-stringize

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 2-30) 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 2-37 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 `-cpredef` 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 compiler.

-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.

Preprocessor Command-Line Reference

-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, `-W1092` disables warning message `ea1092`.

-warn

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

I INDEX

Symbols

- # (stringization) preprocessor operator, [2-30](#)
- ## (concatenate) preprocessor operator, [2-31](#)
- #define (macro) preprocessor command, [2-14](#)
 - variable argument list, [2-15](#)
- #elif (else if) preprocessor command, [2-17](#)
- #else (alternate instruction) preprocessor command, [2-18](#)
- #endif (termination) preprocessor command, [2-19](#)
- #error (error message) preprocessor command, [2-20](#)
- #if (test if true) preprocessor command, [2-21](#)
- #ifdef (test if defined) preprocessor command, [2-22](#)
- #ifndef (test if not defined) preprocessor command, [2-23](#)
- #include (insert a file) preprocessor command, [2-24](#)
- #include preprocessor command, [2-4](#)
- #line (output line number) preprocessor command, [2-26](#)
- #pragma preprocessor command, [2-27](#)
- #undef (undefine) preprocessor command, [2-28](#)
- #warning (warning message) preprocessor command, [2-29](#)
- ... preprocessor operator, [2-15](#)
- .ALIGN (address alignment) assembler directive, [1-41](#)
- .ELIF conditional assembly directive, [1-32](#)
- .ELSE conditional assembly directive, [1-32](#)
- .ENDIF conditional assembly directive, [1-32](#)
- .ENDSEG assembler directive, [1-69](#)
- .EXTERN (global label) assembler directive, [1-43](#)
- .EXTERN STRUCT assembler directive, [1-44](#)
- .FILE (override filename) assembler directive, [1-46](#)
- .GLOBAL (global symbol) assembler directive, [1-47](#)
- .IF conditional assembly directive, [1-32](#)
- .IMPORT assembler directive, [1-48](#)
- .IMPORT header files, [1-48](#)

INDEX

- .LEFTMARGIN assembler directive, [1-50](#)
- .LIST assembler directive, [1-51](#)
- .LIST_DATA assembler directive, [1-52](#)
- .LIST_DATFILE assembler directive, [1-53](#)
- .LIST_DEFTAB assembler directive, [1-54](#)
- .LIST_LOCTAB assembler directive, [1-55](#)
- .LIST_WRAPDATA assembler directive, [1-56](#)
- .NEWPAGE assembler directive, [1-57](#)
- .NOLIST assembler directive, [1-51](#)
- .NOLIST_DATA assembler directive, [1-52](#)
- .NOLIST_DATFILE assembler directive, [1-53](#)
- .NOLIST_WRAPDATA assembler directive, [1-56](#)
- .PAGELENGTH assembler directive, [1-58](#)
- .PAGewidth assembly directive, [1-59](#)
- .PORT (declare port) assembler legacy directive, [1-60](#)
- .PRECISION assembler directive, [1-61](#)
- .PREVIOUS assembler directive, [1-62](#)
- .ROUND_MINUS (rounding mode) assembler directive, [1-63](#)
- .ROUND_NEAREST (rounding mode) assembler directive, [1-63](#)
- .ROUND_PLUS (rounding mode) assembler directive, [1-63](#)
- .ROUND_ZERO (rounding mode) assembler directive, [1-63](#)
- .SECTION (start or embed a section) assembler directive, [1-65](#)
- .SECTION assembler directive initialization qualifiers, [1-67](#)
- .SEGMENT (legacy directive) assembler directive, [1-69](#)
- .STRUCT (struct variable) assembler directive, [1-70](#)
- .TYPE (change default type) assembler directive, [1-74](#)
- .VAR (data variable) assembler directive, [1-75](#)
- .VCSE_METHODCALL_END assembler directive, [1-79](#)
- .VCSE_METHODCALL_START assembler directive, [1-79](#)
- .VCSE_RETURNS assembler directive, [1-79](#)
- .WEAK assembler directive, [1-80](#)
- ? preprocessor operator, [2-32](#)
- __DATE__ macro, [2-9](#)
- __FILE__ macro, [2-9](#)
- __LASTSUFFIX__ macro, [2-9](#), [2-33](#)
- __LINE__ macro, [2-9](#)
- __SILICON_REVISION__ macro, [1-93](#)

__STDC__ macro, [2-9](#)
 __TIME__ macro, [2-9](#)
 __VA_ARGS__ identifier, [2-16](#)

A

absolute address, [1-36](#)
 address alignment, [1-41](#)
 ADI macro, [2-9](#)
 archiver
 object file input to, [1-5](#)
 arithmetic
 fractional, [1-30](#)
 mixed fractional, [1-31](#)
 ASCII string initialization support,
 [1-77](#)
 assembler
 command-line switch
 -D (define macro), [1-86](#)
 -D (defines) option, [1-87](#)
 -flags-compiler, [1-86](#)
 -flags-pp, [1-88](#)
 -g (generate debug info), [1-88](#)
 -h (help), [1-88](#)
 -i (include directory path), [1-89](#)
 -I option, [1-87](#)
 -l (listing file) switch, [1-89](#)
 -li (listing with include) switch,
 [1-90](#)
 -M (make rule only), [1-90](#)
 -MM (generate make rule and
 assemble), [1-90](#)
 -Mo (output make rule), [1-91](#)
 -Mt (output make rule for
 named object), [1-91](#)

 -no-source-dependency, [1-91](#)
 -o (output), [1-92](#)
 -pp (proceed with preprocess-
 ing), [1-92](#)
 -proc processor, [1-92](#)
 -save-temps (save intermediate
 files), [1-93](#)
 -si-revision version (silicon revi-
 sion), [1-93](#)
 -sp (skip preprocessing), [1-94](#)
 -v (verbose), [1-94](#)
 -version (display version), [1-94](#)
 -w (skip warning messages),
 [1-94](#)
 -wnumber (warning suppres-
 sion), [1-94](#)
 command-line syntax, [1-82](#)
 directive syntax, [1-6](#), [1-38](#)
 expressions, constant and address,
 [1-25](#)
 feature macros, [1-15](#)
 file extensions, [1-83](#)
 instruction set, [1-5](#)
 keywords, [1-20](#)
 numeric bases, [1-28](#)
 operators, [1-26](#)
 predefined macros, [1-15](#)
 program content, [1-5](#)
 running from command line,
 [1-82](#)
 run-time environment, [1-2](#)
 source files
 (.ASM), [1-3](#)
 special operators, [1-27](#)

INDEX

- symbols, [1-23](#)
- assembler directives
 - [.ALIGN](#), [1-41](#)
 - [.EXTERN](#), [1-43](#)
 - [.EXTERN STRUCT](#), [1-44](#)
 - [.FILE](#) (override filename), [1-46](#)
 - [.GLOBAL](#), [1-47](#)
 - [.IMPORT](#), [1-48](#)
 - [.LEFTMARGIN](#), [1-50](#)
 - [.LIST](#), [1-51](#)
 - [.LIST_DATA](#), [1-52](#)
 - [.LIST_DATFILE](#), [1-53](#)
 - [.LIST_DEFTAB](#), [1-54](#)
 - [.LIST_LOCTAB](#), [1-55](#)
 - [.LIST_WRAPDATA](#), [1-56](#)
 - [.NEWPAGE](#), [1-57](#)
 - [.NOLIST](#), [1-51](#)
 - [.NOLIST_DATA](#), [1-52](#)
 - [.NOLIST_DATFILE](#), [1-53](#)
 - [.NOLIST_WRAPDATA](#), [1-56](#)
 - [.PAGELength](#), [1-58](#)
 - [.PAGewidth](#), [1-59](#)
 - [.PORT](#), [1-60](#)
 - [.PRECISION](#), [1-61](#)
 - [.PREVIOUS](#), [1-62](#)
 - [.ROUND_MINUS](#), [1-63](#)
 - [.ROUND_NEAREST](#), [1-63](#)
 - [.ROUND_PLUS](#), [1-63](#)
 - [.ROUND_ZERO](#), [1-63](#)
 - [.SECTION](#), [1-65](#)
 - [.SEGMENT/.ENDSEG](#), [1-69](#)
 - [.STRUCT](#), [1-70](#)
 - [.TYPE](#), [1-74](#)
 - [.VAR](#), [1-75](#)
 - [.VCSE_METHODCALL_END](#), [1-79](#)
 - [.VCSE_METHODCALL_STAR](#)
[T](#), [1-79](#)
 - [.VCSE_RETURNS](#), [1-79](#)
 - [.WEAK](#), [1-80](#)
 - conditional, [1-32](#)
- assembly language constant, [2-39](#)
- B**
 - backslash character, [2-14](#)
 - block initialization section
 - qualifiers, [1-67](#)
 - built-in functions
 - [OFFSETOF\(\)](#), [1-32](#), [1-34](#)
 - [SIZEOF\(\)](#), [1-32](#), [1-35](#)
- C**
 - C and assembly, interfacing, [1-11](#)
 - C compiler struct initialization,
[1-70](#)
 - C structs
 - in assembler, [1-12](#)
 - C/C++run-time library
 - initializing, [1-67](#)
 - comma-separated option, [1-88](#)
 - concatenate (##) preprocessor
operator, [2-31](#)
 - conditional assembly directives
 - [.ELIF](#), [1-32](#)
 - [.ELSE](#), [1-32](#)
 - [.ENDIF](#), [1-32](#)
 - [.IF](#), [1-32](#)
 - constant expression, [1-25](#)

conventions

- comment strings, [1-31](#)
- file extensions, [1-83](#)
- file names, [1-82](#)
- numeric formats, [1-28](#)
- user-defined symbols, [1-23](#)
- cpredef (C-style definitions)
 - preprocessor switch, [2-37](#)
- cs! ("!" comment style)
 - preprocessor switch, [2-38](#)
- cs/* ("/* */" comment style)
 - preprocessor switch, [2-38](#)
- cs// ("//" comment style)
 - preprocessor switch, [2-38](#)
- cs{ ("{" comment style)
 - preprocessor switch, [2-38](#)
- csall (all comment styles)
 - preprocessor switch, [2-39](#)
- cstring (C style) preprocessor switch, [2-37](#)
- custom processors, [1-93](#)

D

- D (define macro) assembler switch, [1-86](#)
- D (define macro) preprocessor switch, [2-39](#)
- D (defines) command-line option, see -flags-compiler switch, [1-87](#)
- D__2102x__ macro, [1-15](#)
- D__2106x__ macro, [1-15](#)
- D__2116x__ macro, [1-15](#)
- D__2126x__ macro, [1-16](#)
- D__2136x__ macro, [1-16](#)

- D__2636x__ macro, [1-16](#)
- D__ADSP21000__ macro, [1-15](#), [2-9](#)
- D__ADSP21020__ macro, [1-15](#), [2-9](#)
- D__ADSP21060__ macro, [2-9](#)
- D__ADSP21061__ macro, [2-9](#)
- D__ADSP21062__ macro, [2-9](#)
- D__ADSP21065L__ macro, [2-9](#)
- D__ADSP2106x__ macro, [2-10](#)
- D__ADSP21160__ macro, [2-9](#)
- D__ADSP21161__ macro, [2-10](#)
- D__ADSP2116x__ macro, [1-15](#), [2-10](#)
- D__ADSP21261__ macro, [2-10](#)
- D__ADSP21262__ macro, [2-10](#)
- D__ADSP21266__ macro, [2-10](#)
- D__ADSP21267__ macro, [2-10](#)
- D__ADSP2126x__ macro, [1-16](#)
- D__ADSP21363__ macro, [2-10](#)
- D__ADSP21364__ macro, [2-10](#)
- D__ADSP21365__ macro, [2-10](#)
- D_LANGUAGE_ASM macro, [1-15](#), [2-10](#)
- D_LANGUAGE_C macro, [1-16](#), [2-10](#)
- DATA64, 64-bit word section type, [1-65](#)
- debugging information), [1-88](#)
- default
 - defines, [1-87](#)
 - symbol type, [1-74](#)
 - tab width, [1-54](#), [1-55](#)
- defines options, [1-87](#)

INDEX

defining

 a macro, [2-14](#)

directives

 assembler, [1-38](#)

DM (data), 40-bit word section

 type, [1-65](#)

E

easm21k assembler, [1-2](#)

ELF.h header file, [1-66](#)

expressions

 address, [1-26](#)

 constant, [1-25](#)

F

feature assembler macros

 -D__ADSP21000__, [1-15](#)

 -D__ADSP21020__, [1-15](#)

 -D__ADSP21060__, [1-15](#)

 -D__ADSP21061__, [1-15](#)

 -D__ADSP21062__, [1-15](#)

 -D__ADSP21065L__, [1-15](#)

 -D__ADSP21160__, [1-15](#)

 -D__ADSP21161__, [1-15](#)

 -D__ADSP21261__, [1-16](#)

 -D__ADSP21262__, [1-16](#)

 -D__ADSP21266__, [1-16](#)

 -D__ADSP21267__, [1-16](#)

 -D__ADSP21363__, [1-16](#)

 -D__ADSP21364__, [1-16](#)

 -D__ADSP21365__, [1-16](#)

 -D_LANGUAGE_ASM, [1-15](#)

feature preprocessor macros

 -D__ADSP21020__, [2-9](#)

 -D__ADSP21060__, [2-9](#)

 -D__ADSP21061__, [2-9](#)

 -D__ADSP21062__, [2-9](#)

 -D__ADSP21065L__, [2-9](#)

 -D__ADSP2106x__, [2-10](#)

 -D__ADSP21160__, [2-9](#)

 -D__ADSP21161__, [2-10](#)

 -D__ADSP2116x__, [2-10](#)

 -D__ADSP21261__, [2-10](#)

 -D__ADSP21262__, [2-10](#)

 -D__ADSP21266__, [2-10](#)

 -D__ADSP21267__, [2-10](#)

 -D__ADSP21363__, [2-10](#)

 -D__ADSP21364__, [2-10](#)

 -D__ADSP21365__, [2-10](#)

 -D__ADSPTS__, [2-9](#)

 -D_LANGUAGE_ASM, [2-10](#)

 -D_LANGUAGE_C, [2-10](#)

file extensions

 .ASM (assembly source), [1-3](#)

 .DAT (data file), [1-3](#)

 .DLB (library file), [1-4](#)

 .DOJ (object file), [1-3](#)

 .H (header file), [1-3](#)

 .IS (preprocessed assembly file),
 [1-92](#)

file formats

 ELF (Executable and Linkable
 Format), [1-3](#)

files

 extensions, [1-83](#)

 naming conventions, [1-82](#)

 -flags-compiler assembler switch,
 [1-86](#)

-flags-pp assembler switch, [1-88](#)

floating-point

precision, [1-61](#)

rounding, [1-63](#)

formats

numeric, [1-28](#)

fractional

arithmetic, [1-30](#)

constants, [1-30](#)

fracts

1.0r special case, [1-30](#)

1.31 format, [1-30](#)

constants, [1-29](#)

mixed type arithmetic, [1-31](#)

signed values, [1-29](#)

G

generating

unique labels, [2-32](#)

global substitutions, [2-4](#)

H

-h (help) assembler switch, [1-88](#),
[2-39](#)

header files

system, [2-4](#)

tokens, [2-5](#)

user, [2-5](#)

I

-i (include directory path) assembler
switch, [1-89](#)

-i (include directory) preprocessor
switch, [2-40](#)

-I (include search-path)) assembler
options, [1-87](#)

-i (less includes) preprocessor
switch, [2-39](#)

-I- (search system include files)
preprocessor switch, [2-41](#)

-I- (system include files)
preprocessor switch, [2-41](#)

-I assembler switch, see
-flags-compiler switch, [1-87](#)

IMPORT headers

make dependencies, [1-17](#)

include files

system header files, [2-4](#)

user header files, [2-4](#)

include path search, [2-6](#)

initialization section qualifiers, [1-67](#)

input section alignment instruction,
[1-41](#)

instruction set, [1-5](#)

intermediate source file (.IS), [1-5](#)

K

keywords

assembler, [1-20](#)

L

-l (listing file) assembler switch,
[1-89](#)

legacy directives

.PORT, [1-60](#)

.SEGMENT/.ENDSEG, [1-69](#)

-li (listing with include) assembler
switch, [1-90](#)

INDEX

- linker
 - object file input to, [1-5](#)
 - Linker Description File, [1-7](#)
 - LIST_DEFTAB assembler
 - directive, [1-54](#)
 - LIST_LOCTAB assembler
 - directive, [1-55](#)
 - listing files
 - .LST extension, [1-5](#), [1-18](#)
 - address, [1-18](#)
 - assembly process information, [1-5](#)
 - assembly source code, [1-18](#)
 - C data structure information, [1-5](#)
 - data initialization, [1-53](#)
 - data opcodes, [1-52](#)
 - large opcodes, [1-56](#)
 - line number, [1-18](#)
 - named, [1-89](#)
 - opcode, [1-18](#)
 - producing, [1-4](#)
 - local tab width, [1-54](#), [1-55](#)
 - long-form initialization, [1-70](#), [1-71](#)
 - M
 - M (make rule only) assembler switch, [1-90](#)
 - M (make rule only) preprocessor switch, [2-41](#)
 - macro expansion
 - tokens, [2-5](#)
 - macros
 - __SILICON_REVISION__, [1-93](#)
 - assembler feature, [1-15](#)
 - defining with variable length
 - argument list, [2-15](#)
 - predefined preprocessor, [2-8](#)
 - preprocessor feature, [2-8](#)
 - writing, [2-6](#)
 - macroSymbol, [2-14](#)
 - make dependencies, [1-48](#)
 - meminit linker switch, [1-67](#)
 - memory
 - initializer, [1-67](#)
 - types, [1-7](#)
 - memory sections
 - declaring, [1-65](#)
 - memory type
 - PM (code and data), [1-65](#)
 - RAM (random access memory), [1-65](#)
 - MM (make rule and assemble)
 - assembler switch, [1-90](#)
 - MM (make rule and assemble)
 - preprocessor switch, [2-41](#), [2-42](#)
 - Mo (output make rule) assembler switch, [1-91](#)
 - Mo (output make rule)
 - preprocessor switch, [2-42](#)
 - Mt (output make rule for named file) assembler switch, [1-91](#)
 - Mt preprocessor switch, [2-42](#)
- N
- nested struct references, [1-36](#)
 - NO_INIT
 - memory section, [1-68](#)
 - section qualifier, [1-67](#)

- no-source-dependency assembler switch, [1-91](#)
- nowarn preprocessor switch, [2-44](#)
- numeric formats, [1-28](#)

O

- o (output) assembler switch, [1-92](#)
- o (output) preprocessor switch, [2-42](#)
- object files
 - .DOJ extension, [1-5](#)
 - producing, [1-4](#)
- OFFSETOF() built-in function, [1-34](#)

P

- PM, 48-bit word section type, [1-65](#)
- pp (proceed with preprocessing) assembler switch, [1-92](#)
- predefined preprocessor macros
 - __DATE__, [2-9](#)
 - __FILE__, [2-9](#)
 - __LASTSUFFIX__, [2-9](#)
 - __LINE__, [2-9](#)
 - __STDC__, [2-9](#)
 - __TIME__, [2-9](#)
 - ADI, [2-9](#)
- preprocessed
 - assembly files, [2-12](#)
 - source file, [2-12](#)
- preprocessing
 - a program, [1-14](#)
- preprocessor
 - command syntax, [2-3](#), [2-12](#)

- command-line switches, [2-35](#)
 - cpredef, [2-37](#)
 - cs!, [2-38](#)
 - cs/* ("/* */" comment style), [2-38](#)
 - cs// ("//" comment style), [2-38](#)
 - cs{ ("{" comment style), [2-38](#)
 - csall (all comment styles), [2-39](#)
 - cstring (C style), [2-37](#)
 - D (define macro), [2-39](#)
 - h (help), [2-39](#)
 - i (include path), [2-40](#)
 - i (less includes), [2-39](#)
 - I- (search system include files), [2-41](#)
 - M (make rule only), [2-41](#)
 - MM (make rule and assemble), [2-42](#)
 - Mo (output make rule), [2-42](#)
 - Mt (output make rule for named file), [2-42](#)
 - notokenize-dot, [2-43](#)
 - nowarn, [2-44](#)
 - o (output), [2-42](#)
 - stringize, [2-43](#)
 - tokenize-dot, [2-43](#)
 - Uname, [2-43](#)
 - v (verbose), [2-43](#)
 - version (display version), [2-44](#)
 - w (skip warning messages), [2-44](#)
 - warn (print warnings), [2-44](#)
 - Wnumber (warning suppression), [2-44](#)

INDEX

- command-line syntax, [2-34](#)
- commands, [1-6](#)
 - list of, [2-12](#)
- compiler, [2-2](#)
- feature macros, [2-8](#)
- global substitutions, [2-4](#)
- guide, [2-2](#)
- option settings, [2-11](#)
- output file (.IS extension), [1-5](#)
- overview, [2-1](#)
- predefined macros, [2-8](#)
- running from command line, [2-34](#)
- system header file, [2-24](#)
- user header file, [2-24](#)
- preprocessor commands, [2-12](#)
 - `#define`, [2-14](#)
 - `#elif`, [2-17](#)
 - `#else`, [2-18](#)
 - `#endif`, [2-19](#)
 - `#error`, [2-20](#)
 - `#if`, [2-21](#)
 - `#ifdef`, [2-22](#)
 - `#ifndef`, [2-23](#)
 - `#include`, [2-24](#)
 - `#line` (counter), [2-26](#)
 - `#pragma`, [2-27](#)
 - `#undef`, [2-28](#)
 - `#warning`, [2-29](#)
- preprocessor operators
 - `#` (stringization), [2-30](#)
 - `##` (concatenate), [2-31](#)
 - `?` (generate unique label), [2-32](#)

- `-proc` (target processor) assembler switch, [1-92](#)
- programs
 - assembling, [1-3](#)
 - content, [1-5](#)
 - listing files, [1-18](#)
 - preprocessing, [1-14](#)
 - structure, [1-7](#)
 - writing assembly, [1-3](#)
- project settings
 - assembler, [1-95](#)
 - preprocessor, [1-14](#), [2-11](#)

R

- relational
 - expressions, [1-33](#)
 - operators, [1-26](#)
- rounding modes, [1-63](#)
- `RUNTIME_INIT`
 - section qualifier, [1-67](#)

S

- `-save-temps` (save intermediate files) assembler switch, [1-93](#)
- searching
 - system include files, [2-41](#)
- section
 - name symbol, [1-65](#)
 - type identifier, [1-66](#)
 - type qualifier, [1-65](#)
- section qualifier
 - DM (data memory), [1-65](#)
 - PM (code and data), [1-65](#)
 - RAM (random access memory),

INDEX

- 1-65
 - sectionTypes identifier, 1-66
 - settings
 - assembler options, 1-95
 - from command line, 1-81
 - from VisualDSP++ IDDE, 1-95
 - default tab width, 1-54
 - local tab width, 1-55
 - preprocessor options
 - from command line, 2-11
 - from VisualDSP++ IDDE, 2-11
 - through build tools, 2-11
 - SHF_ALLOC flag, 1-68
 - SHF_INIT flag, 1-68
 - short-form initialization, 1-70
 - SHT_PROGBITS
 - identifier, 1-66
 - memory section, 1-68
 - si-revision (silicon revision)
 - assembler switch, 1-93
 - SIZEOF() built-in function, 1-35
 - source files
 - (.ASM), 1-3
 - sp (skip preprocessing) assembler
 - switch, 1-94
 - special characters
 - dot, 1-24
 - stringization operator, 2-30, 2-43
 - stringize (double quotes)
 - preprocessor switch, 2-43
 - struct
 - compiler keyword, 1-70
 - layout, 1-48, 1-70
 - member initializers, 1-70
 - variable, 1-70
 - struct references, 1-36
 - nested, 1-36
 - STT_* symbol type, 1-74
 - switches (see assembler
 - command-line switches)
 - switches (see preprocessor
 - command-line switches)
 - symbol
 - conventions, 1-24
 - types, 1-74
 - symbolic expressions, 1-26
 - symbols (see assembler symbols)
 - syntax
 - assembler command line, 1-82
 - assembler directives, 1-38
 - constants, 1-25
 - instruction set, 1-5
 - macro, 2-6
 - preprocessor command, 2-12
 - system
 - header files, 2-4
 - include files, searching, 2-41
- ## T
- tab
 - characters, 1-54
 - characters in source file, 1-55
 - tokenize-dot (identifier parsing)
 - preprocessor switch, 2-43
 - tokens
 - macro expansion, 2-5

INDEX

U

- Uname preprocessor switch, [2-43](#)
- unique labels, [2-32](#)
- user header files, [2-4](#)

V

- v (verbose) preprocessor switch, [2-43](#)
- v, -verbose assembler switch, [1-94](#)
- variable length argument list, [2-15](#)
- VCSE_ optimization directives, [1-79](#)
- version (display version) assembler switch, [1-94](#)
- version (display version) preprocessor switch, [2-44](#)
- VisualDSP++
 - Assemble tab, [1-95](#), [2-11](#)
 - assembler settings, [1-95](#)
 - assembling from, [1-3](#)
 - preprocessor settings, [2-11](#)
 - Project Options dialog box, [1-19](#), [1-95](#), [2-11](#)
 - setting assembler options, [1-95](#)
 - setting preprocessor options, [2-11](#)

W

- w (skip warning messages) assembler switch, [1-94](#)
- w (skip warning messages) preprocessor switch, [2-44](#)
- warn (print warnings) preprocessor switch, [2-44](#)
- warning suppression, [2-44](#)
- warnings
 - printing, [2-44](#)
- weak symbol binding, [1-80](#)
- wnumber (warning suppression) assembler switch, [1-94](#)
- Wnumber (warning suppression) preprocessor switch, [2-44](#)
- wrapping
 - opcode listings, [1-56](#)
- writing
 - assembly programs, [1-3](#)

Z

- ZERO_INIT
 - memory section, [1-68](#)
 - section qualifier, [1-67](#)

