

VISUALDSP++™ 3.0

User's Guide

for SHARC® DSPs

Revision 4.0, January 2003

Part Number
82-001962-01

Analog Devices, Inc.
Digital Signal Processor Division
One Technology Way
Norwood, Mass. 02062-9106



Copyright Information

©2003 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, VisualDSP, the VisualDSP logo, SHARC, the SHARC logo, TigerSHARC, the TigerSHARC logo, and EZ-ICE are registered trademarks of Analog Devices, Inc.

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

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

CONTENTS

PREFACE

Purpose of This Manual	xxi
Intended Audience	xxi
Manual Contents	xxii
What's New in This Manual	xxiii
Technical or Customer Support	xxiii
Supported Processors	xxiv
Product Information	xxiv
MyAnalog.com	xxv
DSP Product Information	xxv
Related Documents	xxvi
Online Documentation	xxvi
From VisualDSP++	xxvii
From Windows	xxvii
From the Web	xxviii
Printed Manuals	xxviii
VisualDSP++ Documentation Set	xxviii
Hardware Manuals	xxix
Data Sheets	xxix

CONTENTS

How to Contact DSP Publications	xxix
Notation Conventions	xxx

INTRODUCTION TO VISUALDSP++

VisualDSP++ Features	1-2
Integrated Development and Debugging Environment	1-2
Code Development Tools	1-2
Source File Editing Features	1-3
Project Management Features	1-4
Debugging Features	1-5
VDK Features	1-6
VisualDSP++ 3.0 Features	1-8
Project Development	1-12
DSP Project Development Stages	1-12
Simulation	1-12
Evaluation	1-13
Emulation	1-13
Simulation and Emulation	1-13
Targets	1-13
Simulation Targets	1-14
EZ-KIT Lite Targets	1-14
Emulation Targets	1-14
Platforms	1-14
Hardware Simulation	1-15
Debugging Overview	1-15

VisualDSP++ Kernel	1-17
Program Development Steps	1-17
Step 1: Create a Project	1-18
Step 2: Configure Project Options	1-18
Step 3: Add and Edit Project Source Files	1-18
Adding Files to Your Project	1-18
Creating Files to Add to Your Project	1-19
Editing Files	1-19
Managing Project Dependencies	1-19
Step 4: Define Project Build Options	1-19
Configuration	1-20
Project-Wide File and Tool Options	1-20
Individual File and Tool Options	1-20
Step 5: Build a Debug Version of the Project	1-21
Step 6: Create a Debug Session and Load the Executable	1-21
Step 7: Run and Debug the Program	1-21
Step 8: Build a Release Version of the Project	1-21
Code Development Tools	1-22
Compiler	1-23
C++ Runtime Libraries	1-24
Assembler	1-25
Linker	1-26
Linker Description Files	1-27
Expert Linker	1-28

CONTENTS

Expert Linker Window	1-29
Stack and Heap Usage	1-30
Archiver	1-32
Splitter	1-32
Loader	1-33
VCSE	1-35
VCSE Overview	1-35
VCSE Components	1-35
VCSE Component Model Specification	1-36
VCSE Component Model	1-36
VCSE Tools	1-37
Use of VCSE Components with VisualDSP++	1-37
VCSE User Interface	1-38
Tool Chain Integration	1-38
Wizards	1-39
Component Manager	1-39
Structure of VCSE	1-40
Interface Definition Language (IDL) and Compiler	1-42
DSP Projects	1-45
What is a Project?	1-45
Project Options	1-46
Makefiles	1-47
Rules	1-48
Output Window	1-48

Example Makefile	1-49
Project Configurations	1-52
Customized Project Configurations	1-53
Project Build	1-53
Build Options	1-54
File Building	1-55
Post-Build Options	1-55
Command Syntax	1-56
Project Dependencies	1-56
Project Rules	1-57
VisualDSP++ Help System	1-58
Using the Help Window	1-58
Invoking Online Help	1-59
Viewing Context-Sensitive Help	1-60
Viewing Menu, Toolbar, or Window Help	1-61
Viewing Dialog Box Button or Field Help	1-61
Viewing Window Help	1-62
Using Help Window Navigation Buttons	1-62
Copying Example Code from Help	1-63
Printing Help	1-64
Bookmarking Frequently Used Help Topics	1-64
Placing a Bookmark at a Topic	1-65
Opening a Bookmarked Topic	1-65
Navigating in Online Help	1-65

CONTENTS

Using the Search Features	1-67
Help System Search Rules	1-67
Rules for Full-Text Searches	1-67
Rules for Advanced Searches	1-68
Full-Text Searches	1-68
Advanced Search Techniques	1-70
Using Wildcard Expressions	1-70
Using Boolean Operators	1-71
Using Nested Expressions	1-71
Viewing Online Documents	1-72
Printing Online Documents	1-73

ENVIRONMENT

Parts of the User Interface	2-1
Title Bar	2-3
Additional Information in Title Bars	2-4
Title Bar Right-Click Menus	2-4
Control Menu	2-5
Program Icons	2-5
Editor Windows	2-5
Debugging Windows	2-6
Menu Bar	2-6
Command Information	2-7
Toolbars and User Tools	2-7
Built-In Toolbars	2-8

Toolbar Customization	2-9
Toolbars: Docked vs. Floating	2-10
Toolbar Button Appearance	2-10
Toolbar Shape	2-12
Toolbar Rules	2-12
User Tools	2-13
Status Bar	2-13
VisualDSP++ Windows	2-15
Project Window	2-15
Project Tab	2-16
Project Nodes	2-17
Project View Right-Click Menus	2-18
Project Folders	2-18
Project Files	2-19
File Associations	2-19
Automatic File Placement	2-20
File Placement Rules	2-21
Example	2-21
Kernel Tab	2-22
Project Window Right-Click Menus	2-24
Project Window Menu	2-24
Project Icon Menu	2-25
Folder Icon Menu	2-26
File Icon Menu	2-27

CONTENTS

Editor Windows	2-28
Right-Click Menu	2-29
Editor Tab Mode	2-30
Output Window	2-31
Output Window Tabs	2-31
Build Tab	2-32
Console Tab	2-32
Output Window Error Messages	2-33
Error Message Severity Hierarchy	2-34
Syntax of Help for Error Messages	2-34
How to Promote, Demote, and Suppress Compiler Messages	2-36
Log File	2-40
Output Window Customization	2-41
Right-Click Menu	2-42
Window Operations	2-43
Window Manipulation	2-43
Right-Click Menu Options	2-43
Scroll Bars and Resize Pull-Tab	2-44
Windows: Docked vs. Floating	2-44
Example of a Docked Window	2-45
Examples of Floating Windows	2-46
Window Position Rules	2-47
Standard Windows Buttons	2-48
Debugging Windows	2-49

Disassembly Windows	2-51
Other Disassembly Window Features	2-53
Right-Click Menu	2-53
Disassembly Window Symbols	2-54
Expressions Window	2-55
Trace Windows	2-56
Locals Window	2-57
Statistical/Linear Profiling Results Window	2-58
Window Components	2-58
Left Pane	2-58
Right Pane	2-60
Status Bar	2-60
Right-Click Menu	2-60
Window Operations	2-62
Changing the Window View	2-62
Displaying a Source File	2-63
Working with Ranges	2-63
Switching Display Modes	2-64
Filtering PC Samples with No Debug Information	2-66
Call Stack Window	2-67
Memory Windows	2-67
Memory Number Formats	2-68
Right-Click Menu	2-70
Expression Tracking in a Memory Window	2-70

CONTENTS

Memory Window Display Customization	2-73
Memory Map Window	2-74
Register Windows	2-75
Stack Windows	2-77
Custom Register Windows	2-77
Multiprocessor Window	2-78
Multiprocessor Groups	2-78
Focus	2-79
Right-Click Menu	2-79
Multiprocessor Window Tabs	2-80
Status Tab	2-80
Groups Tab	2-80
VDK Status Window	2-82
VDK State History Window	2-84
Thread Status and Event Colors	2-85
Window Operations	2-85
Right-Click Menu	2-86
Target Load Window	2-86
About Debugging Windows	2-87
Editor Window Features	2-87
Syntax Coloring	2-87
Right-Click Menu	2-88
Editor Window Symbols	2-89
Bookmarks	2-89

Context-Sensitive Expression Evaluation	2-89
Viewing an Expression	2-90
Highlighting an Expression	2-90
Source Mode vs. Mixed Mode	2-90
Source Mode	2-90
Mixed Mode	2-91
Expressions in an Expressions Window	2-92
About Expressions	2-93
Plot Windows	2-94
Plot Window Features	2-95
Status Bar	2-95
Right-Click Menu	2-96
Plot Window Statistics	2-98
Plot Configuration	2-99
Plot Window Presentation	2-99
Plot Presentation Options	2-102
Image Viewer	2-103
Right-Click Menu	2-105
Image Configuration Dialog Box	2-106
Gamma Correction Dialog Box	2-107
Export Image Dialog Box	2-108

DEBUGGING

Debug Sessions	3-2
Debug Session Management	3-3

CONTENTS

Breakpoints	3-3
Watchpoints	3-3
Multiprocessor (MP) Debugging	3-3
Setting Up a Multiprocessor Debug Session	3-4
Debugging a Multiprocessor System	3-4
Focus and Pinning	3-5
Window Title Bar Information	3-6
Additional Focus Indication	3-6
Code Analysis Tools	3-7
Statistical Profiling and Linear Profiling	3-7
Simulation	3-7
Emulation	3-8
Traces	3-8
DSP Memory Plots	3-9
Plot Types	3-9
Line Plots	3-10
X-Y Plots	3-11
Constellation Plots	3-12
Eye Diagrams	3-13
Waterfall Plots	3-14
Spectrogram Plots	3-16
Program Execution Operations	3-17
Selecting a New Debug Session at Startup	3-17
Loading the DSP Executable Program	3-18

Using Program Execution Commands	3-18
Restarting the Program	3-19
Performing a Restart during Simulation	3-19
Performing a Restart during Emulation	3-20
Using Breakpoints	3-20
Using Unconditional and Conditional Breakpoints	3-21
Using Watchpoints	3-21
Simulation Tools	3-22
Interrupts	3-22
Input/Output Simulation (Data Streams)	3-22
Simulation Options	3-23
ADSP-21x6x Anomalies	3-24
Short Word Anomaly (ADSP-2106x DSPs Except for the ADSP-21065L)	3-24
Shadow Write FIFO Anomaly (ADSP-2116x Only)	3-27
SIMD Read from Internal Memory with Shadow Write FIFO Hit Anomaly (ADSP-2116x Only)	3-28
How to Record a Simulator Anomaly or Event	3-28
Clock Doubling (ADSP-21161 DSPs Only)	3-31
Crystal Double Mode Enable Pin	3-31
Clock Rate Ratios	3-31
How to Configure the CLKOUT Pin	3-32
SPI Simulation in Slave Mode	3-32
Boot Options	3-33
Image Viewer	3-35

CONTENTS

Flash Programmer	3-35
Flash Devices	3-36
Flash Programmer Functions	3-36
Flash Driver	3-37
Flash Programmer Window	3-37

REFERENCE INFORMATION

Glossary	A-2
File Types	A-21
Keyboard Shortcuts	A-23
Working with Files	A-23
Moving within a File	A-24
Cutting, Copying, Pasting, Moving Text	A-25
Selecting Text within a File	A-25
Working with Bookmarks in an Editor Window	A-26
Building Projects	A-27
Using Keyboard Shortcuts for Program Execution	A-27
Working with Breakpoints	A-28
Working with Multiprocessor Systems	A-28
Obtaining Online Help	A-29
Miscellaneous	A-29
IDDE Command Line Parameters	A-29
Toolbar Buttons	A-30
Text Operations	A-36
Regular Expressions vs. Normal Searches	A-36

Specific Special Characters	A-37
Special Rules for Sequences	A-38
Repetition and Combination Characters	A-38
Match Rules	A-39
Tagged Expressions in Replace Operations	A-39
Comment Start and Stop Strings	A-41

TCL SCRIPTING

Overview of Tcl Scripting	B-1
Analog Devices Tcl Commands	B-1
Additional Tcl Resources	B-2
Tcl Output	B-2
Tcl Command Issuance	B-3
Issuing Commands from a Command Line	B-3
Issuing Commands from the Output Window	B-3
Issuing Commands from the File Menu	B-4
Issuing Commands from an Editor Window	B-5
Issuing Commands from a User Tool	B-5
Examples of Tcl Scripts	B-6
Step and Print Example	B-6
Creating the Tcl Script	B-6
Running the New Tcl Script	B-7
Regression Test Example	B-8
Types of Tcl Commands	B-14
GUI Manipulation Commands	B-15

CONTENTS

Target Query and Manipulation Commands	B-16
Project Build and Maintenance Commands	B-18
dspaddmenuitem	B-20
dspaddstream	B-22
dspcancelbreak	B-24
dspcheckmenuitem	B-25
dspclickmenuitem	B-26
dspdeleteallstream	B-27
dspdeletemenuitem	B-28
dspdeletestream	B-29
dspenablemenuitem	B-30
dspeval	B-31
dspgetbreak	B-33
dspgetmemblock	B-35
dspgetmeminfo	B-37
dspgetprocessors	B-38
dspgetstate	B-39
dspgetswstack	B-40
dsphalt	B-41
dspliststream	B-42
dspload	B-43
dsplookupline	B-44
dsplookupsymbol	B-45
dspmemorywin	B-46

dspplotrotate	B-48
dspplotwin	B-49
dspprojectaddfile	B-53
dspprojectaddfolder	B-54
dspprojectbuild	B-55
dspprojectclose	B-56
dspprojectinfo	B-57
dspprojectload	B-58
dspprojectremovefile	B-59
dspprojectremovefolder	B-60
dspregisterwin	B-61
dspreset	B-62
dsprestart	B-63
dsprun	B-64
dspset	B-65
dspsetbreak	B-66
dspsetmemblock	B-68
dspsetswstack	B-70
dspstepasm	B-71
dspstepin	B-72
dspstepout	B-73
dspstepover	B-74
dspwaitforhalt	B-75

INDEX

CONTENTS

PREFACE

Thank you for purchasing VisualDSP++[®], Analog Devices development software for digital signal processors (DSPs).

Purpose of This Manual

The *VisualDSP++ 3.0 User's Guide for SHARC DSPs* describes the features, components, and functions of VisualDSP++. Use this guide as a reference for developing programs for SHARC processors.

The User's Guide does not include detailed procedures for building and debugging projects. For how-to information, refer to the VisualDSP++ on-line Help and the *VisualDSP++ 3.0 Getting Started Guide for SHARC DSPs*.

Intended Audience

This manual is primarily intended for DSP programmers who are familiar with Analog Devices DSPs, but are unfamiliar with the VisualDSP environment. This manual assumes that the audience has a working knowledge of the appropriate DSP architecture and instruction set.

Programmers who are unfamiliar with Analog Devices DSPs can use this manual, but they should supplement it with other texts, such as *Hardware Reference* and *Programming Reference* manuals that describe the target architecture.

Manual Contents

This manual consists of:

- Chapter 1, “Introduction”
Describes VisualDSP++ features, new Release 3.0 features, project development, code development tools, VCSE, and DSP projects
- Chapter 2, “Environment”
Describes the VisualDSP++ user interface, windows, environment customization, window operations, and the debugging windows
- Chapter 3, “Debugging”
Describes debug sessions, code analysis tools, program execution operations, simulation tools, Image Viewer, and plots
- Appendix A, “Reference Information”
Provides a glossary and information about file types, keyboard shortcuts, command line parameters, scripting, toolbar buttons, and text operations
- Appendix B, “Simulation of SHARC DSPs”
Describes simulator instruction timing analysis, pipeline stages, the Pipeline Viewer, stalls, aborts, the current program counter value, and stepping
- Appendix C, “Tcl Scripting”
Describes the Tool Command Language (Tcl) and command syntax, and shows examples of Tcl scripting used to test DSP systems

What's New in This Manual

The *VisualDSP++ 3.0 User's Guide for SHARC DSPs* supports all SHARC processors. This edition also documents new features that affect plotting, profiling, and other VisualDSP++ functions.

Technical or Customer Support

You can reach DSP Tools Support in the following ways.

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

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

Supported Processors

VisualDSP++ currently supports the following SHARC processors.

- ADSP-21020
- ADSP-21060
- ADSP-21060L
- ADSP-21061
- ADSP-21061L
- ADSP-21062
- ADSP-21062L
- ADSP-21065L
- ADSP-21160M, ADSP-21160N
- ADSP-21161N

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 website 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 notification 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 for you to select the information you want to receive.

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

DSP Product Information

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

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

- 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)
+49 (0) 089 76 903 157 (Europe)
- Access the Digital Signal Processor Division's FTP website at
[ftp ftp.analog.com](ftp://ftp.analog.com) or **ftp 137.71.23.21**
<ftp://ftp.analog.com>

Related Documents

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

VisualDSP++ 3.0 Getting Started Guide for SHARC DSPs

VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs

VisualDSP++ 3.0 Assembler and Preprocessor Manual for SHARC DSPs

VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs

VisualDSP++ 3.0 Product Bulletin for SHARC DSPs

VisualDSP++ 3.0 Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Card

For hardware information, refer to your DSP's *Hardware Reference*, *Programming Reference*, and data sheet.

All documentation is available online. Most documentation is available in printed form.

Online Documentation

Online documentation comprises Microsoft HTML Help (.CHM), Adobe Portable Documentation Format (.PDF), and HTML (.HTM and .HTML) files. A description of each file type is as follows.

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

File	Description
.CHM	VisualDSP++ online Help system files and VisualDSP++ manuals are provided in Microsoft HTML Help format. Installing VisualDSP++ automatically copies these files to the <code>VisualDSP\Help</code> folder. Online Help is ideal for searching the entire tools manual set. Invoke Help from the VisualDSP++ Help menu or via the Windows Start button.
.PDF	Manuals and data sheets in Portable Documentation Format are located in the installation CD's <code>Docs</code> folder. Viewing and printing a .PDF file requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). Running <code>setup.exe</code> on the installation CD provides easy access to these documents. You can also copy .PDF files from the installation CD onto another disk.
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation is located on the installation CD in the <code>Docs\Reference</code> folder. Viewing or printing these files requires a browser, such as Internet Explorer 4.0 (or higher). You can copy these files from the installation CD onto another disk.

From VisualDSP++

VisualDSP++ provides access to online Help. It does not provide access to .PDF files or the supplemental reference documentation (Dinkum Abridged C++ library and FlexLM network licence). Access Help by:

- Choosing **C**ontents, **S**earch, or **I**ndex from the VisualDSP++ **H**elp menu
- Invoking context-sensitive Help on a user interface item (toolbar button, menu command, or window)

From Windows

In addition to shortcuts you may construct, Windows provides many ways to open VisualDSP++ online Help or the supplementary documentation.

Product Information

Help system files (.CHM) are located in the `VisualDSP\Help` folder. Manuals and data sheets in PDF format are located in the `Docs` folder of the installation CD. The installation CD also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation in the `\Reference` folder.

To use Windows Explorer:

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

From the Web

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

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

Printed Manuals

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

VisualDSP++ Documentation Set

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

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

Hardware Manuals

Printed copies of hardware manuals can be ordered through the Literature Center or downloaded from the Analog Devices Web site. The phone number is **1-800-ANALOGD (1-800-262-5643)**. The manuals can be ordered by title or by product number (located on the back cover of each manual).

Data Sheets

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

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

How to Contact DSP Publications

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

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

Notation Conventions

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

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

 Code has been formatted to fit this manual's page width.

Example	Description
Close command (File menu) or OK	Text in bold style indicates the location of an item within the VisualDSP++ environment's menu system and user interface items.
{this that}	Alternative required items in syntax descriptions appear within curly brackets separated by vertical bars; read the example as <i>this</i> or <i>that</i> .
[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 ellipsis; read the example as an optional comma-separated list of <i>this</i> .
.SECTION	Commands, directives, keywords, code examples, and feature names are in text with <code>letter gothic</code> font.
<i>filename</i>	Non-keyword placeholders appear in text with italic style format.
 Note:	A note providing information of special interest or identifying a related topic. In the online version of this book, the word Note appears instead of this symbol.
 Caution:	A caution providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word Caution appears instead of this symbol.

1 INTRODUCTION TO VISUALDSP++

This chapter describes VisualDSP++, a flexible management system that provides a suite of tools for developing DSP applications and for DSP projects.

VisualDSP++ includes:

- Integrated Development and Debugging Environment (IDDE) with VisualDSP++ Kernel (VDK) integration
- C/C++ optimizing compiler with runtime library
- Assembler and linker
- Simulator software and example programs

This chapter contains the following topics.

- [“VisualDSP++ Features” on page 1-2](#)
- [“Project Development” on page 1-12](#)
- [“Code Development Tools” on page 1-22](#)
- [“VCSE” on page 1-35](#)
- [“DSP Projects” on page 1-45](#)
- [“VisualDSP++ Help System” on page 1-58](#)

VisualDSP++ Features

VisualDSP++ includes all the tools you need to build and manage your DSP projects.

Integrated Development and Debugging Environment

The VisualDSP++ single, integrated project management and debugging environment provides complete graphical control of the edit, build, and debug process. In this integrated environment, you can move easily between editing, building, and debugging activities.

Code Development Tools

Depending on the DSP development tools you purchased, VisualDSP++ includes one or more of the following components.

- C/C++ compiler with runtime library
- VisualDSP++ Component Software Engineering (VCSE) Interface Definition Language (VIDL) compiler
- Assembler, linker, preprocessor, and archiver
- Splitter
- Loader

VisualDSP++ supports ELF/DWARF-2 (Executable Linkable Format) executable files. VisualDSP++ supports all executable file formats produced by the linker.



If your system is configured with third-party development tools, you can select the compiler, assembler, linker, or loader to use for a particular target build.

Source File Editing Features

VisualDSP++ simplifies tasks involving source files. You can easily perform all the activities necessary to create, view, print, move within, and locate information.

- **Edit text files.** Create and modify source files and view listing or map files generated by the DSP code development tools.

Source files are the C/C++ language or assembly language files that make up your project.

DSP projects can include additional files such as data files and a Linker Description File (.LDF), which contains command input for the linker. For more information about .LDF files, see [“Linker” on page 1-26](#).

- **Editor windows.** Open multiple editor windows to view and edit related files, or open multiple editor windows for a single file. The VisualDSP++ editor is an integrated code-writing tool that enables you to focus on code development.
- **Specify syntax coloring.** Configure options that specify the color of text objects viewed in an editor window.

This feature enhances the view and helps you to locate portions of the text, because keywords, quotes, and comments appear in distinct colors.

- **Context-sensitive expression evaluation.** Move the mouse pointer over a variable that is in the scope and view the variable's value.
- **Status icons.** View icons that indicate breakpoints, bookmarks, and the current PC position.

VisualDSP++ Features

- **Editor display format.** Specify an editor window's display format: source mode or mixed mode.
- **View offending code.** Double-click on an error from the **Output** window's **Build** view to jump to the offending code in an editor window.

Project Management Features

VisualDSP++ provides flexible project management for the development of DSP applications, including access to all the activities necessary to create, define, and build DSP projects.

- **Define and manage projects.** Identify files that the code development tools process to build your project. Create this project definition once, or modify it to meet changing development needs.
- **Access and manage code development tools.** Configure options to specify how the DSP code development tools process inputs and generate outputs.

Tool settings correspond to command line switches for code development tools. Define these options once, or modify them to meet your needs.

- **View and respond to project build results.** View project status while a build progresses and, if necessary, halt the build.

Double-click on an error message in the **Output** window to view the source file causing the error, or iterate through the error messages.

- **Manage source files.** Manage source files and track file dependencies in your project from the **Project** window to provide a display of software file relationships. VisualDSP++ uses code development tools to process your project and to produce a DSP program.

Debugging Features

While debugging your project, you can:

- **View and debug mixed C/C++ and assembly code.** View C/C++ source code interspersed with assembly code. Line number and symbol information help you to source-level debug assembly files.
- **Run Tcl command-line scripts.** Use Tcl and its ADI extensions to customize key debugging features. VisualDSP++ supports Tool Command Language (Tcl) version 8.3. For more information, see Appendix B, “Tcl Scripting.”
- **Use memory expressions.** Use expressions that refer to memory.
- **Use breakpoints to view registers and memory.** Quickly add and remove, and enable and disable breakpoints.
- **Set simulated watchpoints.** Set watchpoints on stacks, registers, memory, or symbols to halt program execution.
- **Trace program execution history.** Trace results show how your program arrives at a certain point, and show program reads, writes, and symbolic names.
- **Statistically profile the target processor’s PC** (JTAG emulator debug targets only). Take random samples and display them graphically to see where the program uses most of its time.
- **Linearly profile the target processor’s PC** (simulation only). Sample every executed PC and provide an accurate and complete graphical display of what was executed in your program.
- **Create interrupts generated by peripherals.** Use interrupts with serial port (SPORT) I/O data streams to simulate DSP operation.
- **Transfer data through memory-mapped I/O.** Transfer data without going through peripherals.

VisualDSP++ Features

- **Create customized register windows.** Configure a custom register window to display a specified set of registers.
- **Plot values from DSP memory.** Choose from multiple plot styles, data processing options, and presentation options.
- **View pipeline depth of assembly instructions.** Display pipeline stages (fetch, decode, and execute) in the Disassembly window when the selected session is connected to a simulator target.

For details, see the *VisualDSP++ 3.0 Getting Started Guide for SHARC DSPs*.

VDK Features

The VisualDSP++ Kernel (VDK) is a scalable software executive specially developed for effective operations on Analog Devices SHARC DSPs. Although the kernel is tightly integrated with VisualDSP++, you can also use it via standard command line development tools.

The kernel enables you to abstract the details of the hardware implementation from the software design. As a result, you can concentrate on processing algorithms.

The kernel provides all the basic building blocks required for application development. Integration of the kernel is characterized as follows.

- **Automatic.** VisualDSP++ automatically generates source code framework for each user-requested object in the user-specified language.
- **Deterministic.** You can specify a function's execution time.
- **Multitasking.** Kernel tasks (threads) are independent of one another. Each thread has its own stack.

- **Modular.** The kernel comprises several components, including message queues, a memory pool manager, semaphores, and device flags.
- **Portable.** Most of the kernel components can be written in ANSI Standard C or C++ and are portable to other Analog Devices DSPs.
- **Pre-emptive.** The kernel's priority-based scheduler enables the highest priority thread not waiting for a signal to be run at any time.
- **Prototypical.** The kernel and VisualDSP++ create an initial file set based on a series of template files. The entire application is prototyped and ready to be tested.
- **Reliable.** Besides detecting as many errors as possible at build time, the kernel supports multiple models for error handling.
- **Scalable.** If a project does not include a kernel feature, the support code is not included in the target system.

VisualDSP++ 3.0 Features

VisualDSP++ 3.0 includes the following new features and enhancements.

- **Expert Linker.** This graphical tool makes it easier for you to produce a Linker Description File (.LDF). You can display and modify your DSP's memory map simply by dragging and dropping object sections into the map. You can then drag those object sections from slower external memory to faster internal memory to improve performance. You can also see how much space is allocated for your program's heap and stack and reduce their size if they are using too much memory.
- **New profiler.** A new linear profiler replaces the legacy profiler. A new menu option (**New Profile**) under **Tools** and **Linear Profiling** opens the **Linear Profiling Results** window. Selecting **Properties** from the window's right-click menu enables you to specify which part of the program to profile and what profiling data to collect for viewing in the profiling window. You can also specify the profiling data to sort and a memory range filter (entire memory, C/C++ functions, or memory range).
- **Support for VisualDSP++ Component Software Engineering (VCSE).** VCSE tools simplify the process of developing, documenting, and validating reusable software components. VisualDSP++ support includes the means to display information about available components and to incorporate them into a project. You use the VCSE Interface Definition Language (VIDL) to define interfaces and the components that implement them. VisualDSP++ provides a VIDL compiler that enables you to create and use components without becoming familiar with the detail of the model and mechanism involved. VisualDSP++ also provides wizards for creating interfaces and components.

- **Image Viewer.** The new **Image Viewer** plugin window enables you to view image and video data (BMP, JPEG, PPM, or MPEG) from DSP memory or from a file on your PC. You can adjust the gamma attributes of an image, copy an image, save an image, print an image, and export an image.
- **Editor enhancements.** New features added to VisualDSP++ editor windows include a right-click menu command that toggles the display of line numbers, brace matching based on the position of the text cursor, and a right-click command to open header files from the filename in an `#include` statement.
- **Streams interface enhancements.** The new **Streams** dialog box, accessed from the **Settings** menu, enables you to add and modify streams. Settings for the streams are automatically saved and restored, so you do not have to set up the streams every time you start VisualDSP++. A new check box, **Rewind at Start**, provides loop back support by allowing you to use one file to continue testing. A new **Processor** field enables you to set up streams between processors in a multiprocessor session (emulator sessions only). New interfaces are supported by the targets for compiled simulation, and new Tcl commands are available for creating and deleting streams.
- **Plot enhancements.** Scroll bars, slider controls, and zoom buttons added to plot windows improve usability. Other new features include options for displaying plot statistics to the right of the plot and for playing audio data through a PC sound card.

- **VDK project enhancements.** A memory pool element added to the **Kernel** view in the **Project** window has the following properties associated with it: block size, number of blocks, and initialize on pool creation. Right-clicking on the memory pool element enables you to create a new memory pool. Release 3.0 provides messaging between threads, dynamic (runtime) creation of semaphores and device flags, and support for measuring thread stack usage. Device drivers are now part of the I/O interface.

The **VDK Status** window now displays values for additional thread items (last error, stack, and message queue), semaphores, events, event bits, device flags, and memory pools.

The **VDK State History** window now displays new log events, including user-defined events, and user-configurable colors, patterns, and fonts. It also provides options for data filtering and searching.

- **External makefile support.** You can perform common command line operations such as view, edit, and save on a makefile without leaving VisualDSP++. You can open a makefile (`.MAK` or `.MK` extension) as a text file or a project file. Double-clicking the makefile in the **Project** window opens it as a text file for viewing and editing in an editor window.

If you open the makefile as a project, VisualDSP++ looks for a project file with the same name in the makefile folder. If it finds the project, it adds the makefile to the project. Otherwise, it prompts you to create a new project and automatically adds the makefile to the new project.

Right-clicking a makefile in the **Project** window opens the makefile's **File Options** property page, where you specify the command for building the project. You use the active makefile with the `gmake` command to build projects. Errors from invoking the `gmake` command are displayed in the **Output** window. Double-clicking an error message positions the cursor on the line with the error in an editor window. An active makefile in a project disables internal project building in VisualDSP++. Only one makefile can be active when you build a project.

- **Flash Programmer.** The VisualDSP++ Flash Programmer provides a convenient graphical interface to on-board flash memory devices. Use the Flash Programmer to view various flash memory information, fill or erase portions of flash memory, or load an Intel hex formatted file into flash memory. Analog Devices supplies flash drivers (algorithms) for many EZ-KIT Lite™ evaluation systems. Additionally, the Flash Programmer enables you to create custom flash drivers or modify existing drivers for use on custom boards with a variety of processor and flash device combinations.

Project Development

During project development, VisualDSP++ helps you interactively observe and alter the data in the processor and in memory.

DSP Project Development Stages

The typical project includes three phases: *simulation*, *evaluation*, and *emulation*.

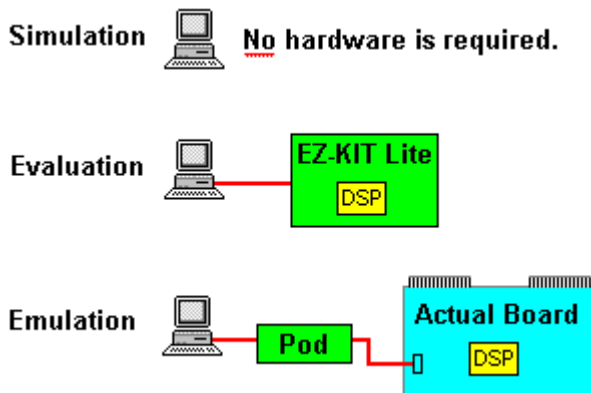


Figure 1-1. Project Development Stages

Simulation

You typically begin project development in a simulation environment while hardware engineers are developing the new hardware (cell phone, computer, and so on). Simulation mimics system memory and I/O, which enables you to view portions of the target system hardware. You run VisualDSP++ with a simulation target without a physical DSP. You can build, edit, and debug your DSP program.

Evaluation

Use an EZ-KIT Lite™ evaluation system in your project's early planning stage to determine the DSP that best fits your needs. Your PC connects to the EZ-KIT Lite board via cable, which enables you to monitor DSP behavior.

Emulation

Once the hardware is ready, you move directly to a JTAG emulator that connects your PC to the actual DSP board.

Simulation and Emulation

VisualDSP++ is the front end for all the available targets and platforms. Use VisualDSP++ during both simulation and emulation.

A *simulator* is software that mimics the behavior of a DSP chip. Use simulators to test and debug DSP code before a DSP chip is manufactured.

An *emulator* is hardware used to connect a PC to a DSP target board to allow application software to be downloaded and debugged from within the VisualDSP++. Emulator software performs the communications that enable you to see how your DSP code affects DSP performance.

Targets

A *target* (or debug target) refers to the communication channel between VisualDSP++ and a DSP (or group of DSPs). A target can be a simulator, EZ-KIT Lite evaluation board, or an emulator. Your system may include multiple targets.

For example, the JTAG emulator communicates with one or more physical devices over the host PC's PCI bus, and the Trek-ICE™ emulator communicates with a device over a TCP/IP network.

Simulation Targets

A *simulation target*, such as the ADSP-2106x Family Simulator, is a pure software module and does **not** require the presence of a DSP for debugging.

During simulation, VisualDSP++ reads an executable file (.EXE) and executes it in software, similar to the way a DSP executes a DSP image in hardware. VisualDSP++ simulates the memory and I/O devices that you specify in an .LDF file.

EZ-KIT Lite Targets

An *EZ-KIT Lite target* is a development board that enables you to evaluate a particular DSP. Analog Devices provides several EZ-KIT Lite evaluation systems, which include demonstration programs.

Emulation Targets

An *emulation target* is a module that controls a physical DSP connected to a JTAG emulator system. For example, the Summit-ICE™ emulator communicates with one or more physical devices through the host PC's PCI bus, and the Apex-ICE emulator communicates with a device through the PC's USB port.

Platforms

A *platform* refers to the configuration of DSPs with which a target communicates.

Several platforms may exist for a given debug target. For example, if three emulators are installed on your system, you might select emulator 2 as the platform that you want to use. The platform that you use depends on your project development stage.

Table 1-1. Development Stages and Supported Platforms

Stage	Platform
Simulation	Typically one or more DSPs of the same type. By default, the platform name is the identical DSP simulator.
Evaluation	An EZ-KIT Lite evaluation system
Emulation	Any combination of devices. You must configure the platform for a particular target with the JTAG ICE Configurator. When the debug target is a JTAG emulator, a platform refers to a JTAG chain.

Hardware Simulation

VisualDSP++ enables you to simulate a hardware environment when connected to a simulation target. You can simulate the following.

- Random interrupts that can occur during program execution
- Processor booting from a PROM or host processor

Setting up VisualDSP++ to generate random interrupts during program execution enables you to exercise interrupt service routines in your code.

Debugging Overview

Once you have successfully built your DSP project and have generated a DSP executable file, you can debug the project. Projects developed in VisualDSP++ are run as hardware and software *debug sessions*.

In [Table 1-2 on page 1-16](#), the check mark ✓ indicates the various debugging tools that you can use while building and debugging your DSP program.

You can attach to any Analog Devices DSP or DSP simulator and control its operation. Download your application code to the processor and use VisualDSP++'s debugging facilities to ensure that your application functions as desired.

Table 1-2. Tools Used for Simulation and Emulation

Tool	Simulation	Evaluation	Emulation
Linear profiles	✓		
Interrupts	✓		
Streams	✓		
Traces	✓		
Breakpoints	✓	✓	✓
Watchpoints	✓		
Hardware breakpoints			✓
Plotting	✓	✓	✓
Statistical profiles			✓

VisualDSP++ is your window into the inner workings of the target processor or simulator. From this user interface, you can:

- Run, step, and halt the program and set breakpoints
- View the state of the processor's memory, registers, and stacks
- Perform a cycle-accurate statistical profile or linear profile
- Perform integrated multiprocessor debugging (emulator sessions only)

VisualDSP++ Kernel

A SHARC project can include the VisualDSP++ Kernel (VDK), which is a software executive between DSP algorithms, peripherals, and control logic.

The **Project** window's **Kernel** tab enables you to access a tree control for structuring and scaling application development. Using this tree control, you can add, modify, and delete Kernel elements such as thread types, boot threads, round-robin priorities, semaphores, events, event bits, interrupts, and device drivers.

Two VDK-specific windows, **VDK State History** and **Target Load**, provide views of VDK information.

Another VDK window, **VDK Status**, provides thread status data when a VDK-enabled program is halted.

Refer to the *VisualDSP++ Kernel (VDK) User's Guide* for complete details.

Program Development Steps

In the VisualDSP++ environment, program development consists of the following steps.

1. Create a project
2. Configure project options
3. Add and edit project source files
4. Define project build options
5. Build a debug version (executable file) of the project
6. Create a debug session and load the executable

Project Development

7. Run and debug the program
8. Build a release version of the project

By following these steps, you can build DSP projects consistently and accurately with minimal project management. This process reduces development time and lets you concentrate on code development.

These steps, described below, are covered in detail in the online Help and in the “Tutorial” chapter of the *VisualDSP++ 3.0 Getting Started Guide for SHARC DSPs*.

Step 1: Create a Project

All development in VisualDSP++ occurs within a project. The project file (.DPJ) stores your program’s build information: source files list and development tools option settings.

Step 2: Configure Project Options

Define the target processor and set up your project options (or accept default settings) before adding files to the project. The **Project Options** dialog box provides access to project options, which enable the corresponding build tools to process the project’s files correctly.

Step 3: Add and Edit Project Source Files

A project normally contains one or more C, C++, or assembly language source files. After you create a project and define its target processor, you add new or existing files to the project by importing or writing them. Use the VisualDSP++ Editor to create new files or edit any existing text file.

Adding Files to Your Project

You can add any type of file to the project. The DSP development tools selectively process only recognized file types when building the project.

Creating Files to Add to Your Project

You can create new text files. The Editor can read or write text files with arbitrary names. When you add files to your project, VisualDSP++ updates the project's file tree in the **Project** window.

Editing Files

You can edit the file(s) that you add to the project. To open a file for editing, double-click on the file icon in the **Project** window.

The editor has a standard Windows-style user interface and can handle normal editing operations and multiple open windows. Additional features include customizable language- and DSP-specific syntax coloring, and bookmark capabilities (creation and search).

Managing Project Dependencies

Project dependencies control how source files use information in other files, and consequently determine the build order. VisualDSP++ maintains a makefile, which stores dependency information for each file in the project. VisualDSP++ updates dependency information when you change the project's build options, when you add a file to the project, or when you choose **Update Dependencies** from the **Project** menu.

Step 4: Define Project Build Options

After you create a project, set the target processor, and add or edit the project's source files, you configure your project's build options. You must specify options or accept the default options in VisualDSP++ before using the development tools that create your executable file. You can specify options for a whole project or for individual files, or you can specify a custom build.



VisualDSP++ retains your changes to the build options. Settings reflect your last changes, not necessarily the original defaults.

Project Development

Configuration

A project's configuration setting controls its build. By default, the choices are **Debug** or **Release**.

- Selecting **Debug** and leaving all other options at their default settings builds a project that can be debugged. The compiler generates debug information.
- Selecting **Release** and leaving all other options at their default settings builds a project with limited or no debug capabilities. Release builds are usually optimized for performance. Your test suite should verify that the Release build operates correctly without introducing significant bugs.

You can modify VisualDSP++'s default operation for either configuration by changing the appropriate entries in the compile, assemble, and link property pages. You can create custom configurations that include the build options and source files that you want.

Project-Wide File and Tool Options

Next, you must decide whether to use project-wide option settings or to use individual file settings.

For projects built entirely within VisualDSP++ with no pre-existing object or archive (library) files, you typically use project-wide options. New files added to the project inherit these settings.

Individual File and Tool Options

Occasionally, you may want to specify tool settings for individual files. Each file is associated with two property pages: a **General** page, which lets you choose output directories for intermediate and output files, and a tool-specific property page (**Compile**, **Assemble**, **Link**, and so on), which lets you choose options. For information about each tool's options, see the online Help or the manual for each tool.

Step 5: Build a Debug Version of the Project

Now you must build a debug version of the project.

Status messages from each code development tool appear in the **Output window** as the build progresses.

 The output file type *must* be an executable (.EXE) file to produce debugger-compatible output.

Step 6: Create a Debug Session and Load the Executable

After you successfully build an executable file, you set up a debug *session*. You run DSP projects that you develop as either hardware or software sessions. After you specify target and processor information, you must load your project's executable file. On the **General** page in the **Preferences** dialog box, you can configure VisualDSP++ to load the file automatically and advance to the `main` function of your code.

Step 7: Run and Debug the Program

After you successfully create a debug session and build and load your executable program, you run and debug the program.

If the project is not current (has outdated source files or dependency information), VisualDSP++ prompts you to build the project before loading and debugging the executable file.

Step 8: Build a Release Version of the Project

After you finish debugging your application, you build a Release version of your project to run on the product's DSP.

Code Development Tools

Depending on the DSP development tools that you purchased, VisualDSP++ includes one or more of the following components.

- C/C++ compiler with runtime libraries
- VIDL compiler
- Assembler
- Linker
- Expert Linker
- Preprocessor
- Archiver
- Splitter
- Loader

Available code development tools differ, depending on your DSP. The options available on the tabbed pages of the **Project Options** dialog box enable you to specify tool preference.

VisualDSP++ supports ELF/DWARF-2 (Executable Linkable Format, Debugging Information Format) executable files. VisualDSP++ supports all executable file formats produced by the linker.

If your system is configured with third-party development tools, you can select the compiler, assembler, or linker to use for a particular target build.

Compiler

The compiler processes C/C++ programs into assembly code. The term *compiler* refers to the compiler utility shipped with the VisualDSP++ software.

The term *compiler* refers to the compiler program (such as ccts) shipped with the VisualDSP++ software.

The compiler generates a linkable object file by compiling one or more C/C++ source files. The compiler's primary output is a linkable object file with a .`OBJ` extension.

You specify the compilation options that you need for your build. VisualDSP++ groups compiler options into categories (see [Table 1-3](#)).

Table 1-3. Groups of Compiler Options

Category	Purpose
General	Optimization, compilation, and termination options
Preprocessor	Macro and directory search options
Warning	Warning and error reporting options

You access each category of options from the **Compile** page of the **Project Options** dialog box.



Compilation options depend on your target DSP and your code development tools.

For more information, refer to the *VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs*.

C++ Runtime Libraries

 You must run VisualDSP++ to use the C++ runtime libraries.

The C and C++ runtime libraries are collections of functions, macros, and class templates that can be called from source programs. Many functions are implemented in the DSP assembly language.

C and C++ programs depend on library functions to perform operations that are basic to the C and C++ programming languages. These operations include memory allocations, character and string conversions, and math calculations. The libraries also include multiple signal processing functions that ease DSP code development. Using the runtime library simplifies software development by providing code for a variety of common needs.

The compiler provides a broad collection of C functions including those required by the ANSI standard and additional Analog Devices-supplied functions of value for DSP programming. In addition to the Standard C Library, this release of the compiler software includes the Abridged Library, a conforming subset of the Standard C++ Library.

For more information about the algorithms on which many of the C library's math functions are based, refer to the Cody and Waite text *Software Manual for the Elementary Functions* from Prentice Hall (1980).

For more information about the C++ library portion of the ANSI/ ISO Standard for C++, refer to the Plauger text *Draft Standard C++ Library* from Prentice Hall (1994) (ISBN: 0131170031).

Assembler

The assembler generates an object file by assembling source, header, and data files. The assembler's primary output is an object file with a `.DOJ` extension.

Assembler terms are defined as follows.

Instruction set

The set of assembly instructions that pertain to a specific DSP. For information on the instruction set, refer to your DSP's *Hardware Reference*.

Preprocessor commands

Commands that direct the preprocessor to include files, perform macro substitutions, and control conditional assembly.

Assembler directives

Directives that tell the assembler how to process your source code and set up DSP features. You use directives to structure your program into logical *segments* or sections that support the use of a Linker Description File (`.LDF`) to construct an image suited to the target system.

For more information, refer to the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for SHARC DSPs*.

Linker

The linker links separately assembled files (object files and library files) to produce executable files (.EXE), shared memory files (.SM), and overlay files (.OVL), which can be loaded onto the target.

The linker's primary output is an executable program file with a .EXE extension. To make an executable file, the linker processes data from a Linker Description File (.LDF) and one or more object files (.OBJ).

Linker terms are defined as follows.

Link against

Functionality that enables the linker to resolve symbols to which multiple executables refer. For instance, shared memory executable files (.SM) contain sections of code that other processor executables (.EXE) link against. Through this process, a shared item is available to multiple executables without being duplicated.

Link objects

Object files (.OBJ) that become linked and other items, such as executables (.EXE, .SM, .OVL), that are linked against

.LDF file

File that contains the commands, macros, and expressions that control how the linker arranges your program in memory

Memory

Definitions that provide the linker with a description of your target DSP system

Overlays

Files that your overlay manager swaps in and out of runtime memory, depending on code operations. The linker produces overlay files (.OVL).

Sections

Declarations that identify the content for each executable that the linker produces

For more information, refer to the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*.

Linker Description Files

A Linker Description File (.LDF) describes the target system and maps your program code with the system memory and processors.

The .LDF file creates an executable file by using:

- The target system memory map
- Specified segments in your source files

The parts of an .LDF file from the beginning to the end of the file, are described as follows.

- Memory map – describes the processor’s physical memory, at the beginning of the .LDF file
- SEARCH_DIR(), \$LIBRARIES, and \$OBJECTS – define the path names that the linker uses to search and resolve references in the input files
- MEMORY{} command – defines the systems’ physical memory and assigns labels to logical segments within it. These logical segments define program, memory, and stack memory types.

- `SECTIONS{ }` command – defines the placement of code in physical memory by mapping the sections specified in program files to the sections declared in the `MEMORY{ }` command. The `INPUT_SECTIONS( )` command specifies the object file that the linker uses to resolve the mapping.

For details, refer to the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*.

Expert Linker

Expert Linker is a graphical tool that enables you to:

- Define a DSP target's memory map
- Place a project's object sections into that memory map
- Determines how much stack or heap has been used after you run a DSP program



The Expert Linker works with the linker. For more information about linking, refer to the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*.

This interactive tool speeds up the configuration of system memory. It uses your application's target memory description, object files, and libraries to create a memory map that you can manipulate to optimize your system's use of memory.

Expert Linker generates a new Linker Description File (`.LDF`) or modifies an existing `.LDF` file. Use the Expert Linker wizard to create and customize a new `.LDF` file.

When you open Expert Linker in a project that already includes an `.LDF` file, Expert Linker parses the `.LDF` file and graphically displays the DSP target's memory map and the object mappings. The memory map appears in the **Expert Linker** window (Figure 1-2).

Expert Linker can graphically display space allocated to program heap and stack. After you load and run your program, Expert Linker indicates the portion of the heap and stack that has been used. You can then reduce the size of the heap or stack to minimize the memory allocated for the heap and stack. Freeing up memory in this way enables you to use it for storing other things like DSP code or data.

Expert Linker Window

The Expert Linker window (Figure 1-2) enables you to modify the memory map or the object mappings. When the project is about to be built, Expert Linker saves the changes to the `.LDF` file.

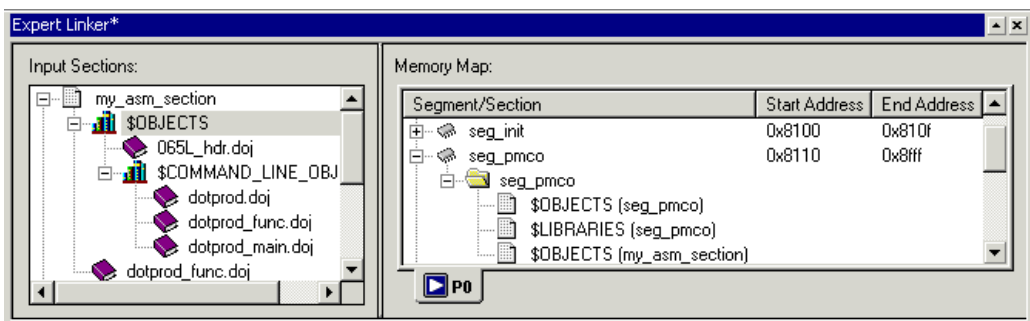


Figure 1-2. Expert Linker Window

The **Expert Linker** window contains two main panes:

- The Input Sections pane displays a tree structure of the input sections.
- The Memory Map pane displays each memory map in a tree or graphical representation.

You can dock or float the **Expert Linker** window in the VisualDSP++ main window.

Stack and Heap Usage

Expert Linker enables you to adjust the size of the stack and heap, and make better use of memory.

Expert Linker can:

- Locate stacks and heaps and fill them with a marker value.
This operation occurs after you load the program into a DSP target. The stacks and heaps are located by their memory segment names, which may vary across processor families
- Search the heap and stack for the highest memory locations written to by the DSP program
This operation occurs when the target halts after you run the program. Assume that these values are the start of the unused portion of the stack or heap. The Expert Linker updates the memory map to show how much of the stack and heap are unused.

[Figure 1-3 on page 1-31](#) shows an example memory map after you run a SHARC program.

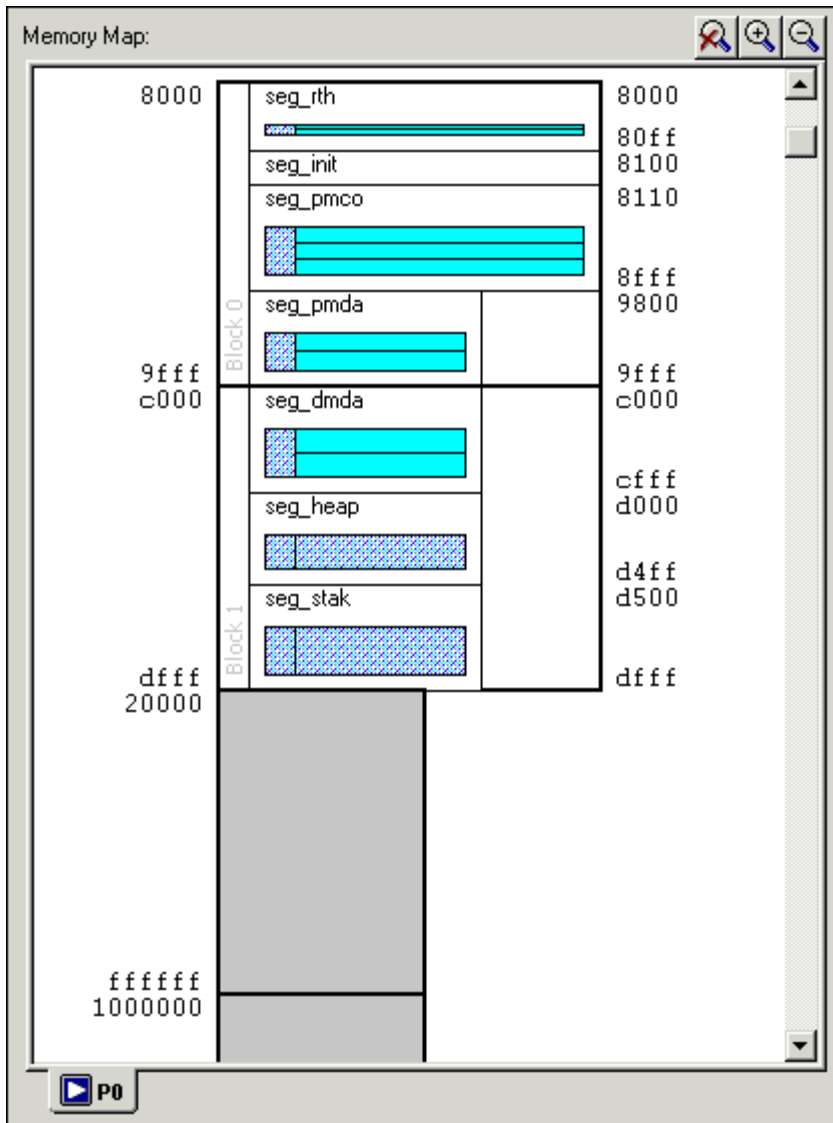


Figure 1-3. Memory Map Example after You Run a SHARC Program

Archiver

The VisualDSP++ archiver, `elfar.exe`, combines object files (`.DOJ`) into library files (`.DLB`), which serve as a reusable resource for project development. The linker searches library files for routines (library members) that are referred to by other objects and links them in your executable program.

You can run the archiver from within VisualDSP++ or from the command line. From VisualDSP++, you can create a library file as your project's output.

To modify or list the contents of a library file (or perform other operations on it), you must run the archiver from a command line. Refer to your DSP's Linker and Utilities Manual for details.

Splitter

The splitter builds boot-loadable files from DSP executables.

The splitter processes a DSP executable file to generate a series of PROM files. The splitter's primary output is a PROM file with these extensions:

- `.S_#`
- `.H_#`
- `.STK`

The splitter is typically used only for programs that execute from external memory. For programs that execute from internal memory, use the loader, which produces boot-loadable files.

Splitter terms are defined as follows.

Non-bootable PROM-image files

The splitter's output, which consists of PROM files that cannot be used to boot-load a system

Splitter

The splitter application, such as `elfspl21k.exe`, contained in the software release

For more information about the splitter, refer to the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*.

Loader

The SHARC loader (`elfloader.exe`) generates a boot-loadable file for SHARC DSPs by processing executable files. To generate a loadable file, the loader processes data from a boot kernel file (`.DXX`), the linker's executable file (`.DXX`), and in some cases overlay files (`.OVL`). The ADSP-21xxx DSPs use one on-chip ROM bootstrap kernel for automatic booting from an external memory device. The ADSP-21xxx DSPs are supported by two booting kernels. Boot loading through the boot kernel is currently supported only on ADSP-21xxx DSPs.

Once you have fully debugged your program, use the loader to generate a set of boot-loadable files for your target system. The loader's primary output is a load file with an `.LDR` extension (`.BNM` for SHARC DSPs) or two output files (`.LDR` boot kernel file and `.KLN` application code file), depending on your loader setup selections.

Loading the loader output into a simulator session in the VisualDSP++ debugger enables you to simulate the boot process and the boot loaded application.

Code Development Tools

Loader terms are defined as follows.

Boot kernel

The executable file that performs the memory initialization on the target

Boot-loadable file

The loader's output, which contains the boot loader and the formatted system configurations. This file is a bootable image file.

Boot loading

The process of loading the boot loader, initializing system memory, and starting the application on the target

Loader

The loader application, such as elfloader.exe, contained in the software release

For more information about the loader, refer to the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*.

VCSE

VCSE Overview

VCSE consists of a combination of tools and guidelines that simplify the process of developing components and help to document and validate such components. These tools and guidelines:

- Enable applications to incorporate and use software algorithm components from other developers easily and with confidence
- Ensure that components from multiple vendors will not interact with each other in unpredictable ways or have resource clashes
- Allow components to be developed in assembler, C, or C++ and be used from applications developed in any of these languages
- Allow components to be reused easily
- Allow comparison of algorithms that offer the same functionality
- Encourage third party developers to provide the implementation of algorithms as easily used components

For more information, refer to the *VisualDSP++ Component Software Engineering User's Guide*.

VCSE Components

VCSE provides support for creating and using software components that are specifically targeted at the embedded space.

VCSE Component Model Specification

Components that adhere to the VCSE Component Model specification enable you to achieve these objectives:

- Create software algorithms as reusable components
- Use software algorithms as components from other developers
- Use components from an assembler, C, or C++ program irrespective of the language that they were implemented in
- Use components from multiple sources predictably in any application without resource conflicts

VCSE Component Model

The VCSE component model does the following.

- Defines a binary standard to allow component interoperability
- Specifies a mechanism that is language independent and usable by assembly, C, and C++ components
- Provides a robust mechanism to cope with the evolution of components over time
- Defines a naming standard to ensure the uniqueness of component names

The binary standard defining the mechanism allows function calls between components, and supports the grouping of available functions or methods into interfaces that are accessible as a unit. Each component can support more than one interface. Each component must support a base interface that can be used to access any other interface that is supported by the component.

VCSE Tools

VCSE tools enable you to create and use components without having to become familiar with the detail of the model and the mechanisms it involves. As a result, you can concentrate on the application itself.

VCSE consists of tools and guidelines that simplify the process of developing components and automate the conformance testing of such components. VCSE components are integrated with VisualDSP++. They simplify the process of incorporating and utilizing components from a variety of developers.

Use of VCSE Components with VisualDSP++

VCSE automatically generates an interface header file that defines the services it offers and provides access to those services from assembly, C, and C++ files.

VCSE components can be integrated with VisualDSP++, so you can view information on all the components that are available. From VisualDSP++, you can view a list of all the registered components that shows all the components that have been registered within VisualDSP++. Some components may not have a complete implementation, but they are available for purchase. You can access the information for each registered component and view the list of interfaces it supports.

When you add a component to a project, VisualDSP++ adds the relevant object and header files to the project, and the supported interfaces. You can use the New Interface Wizard to create an interface and add methods and parameters to it. This process generates the necessary Interface Definition Language (IDL) source code.

VCSE User Interface

Integration of VCSE with VisualDSP++ simplifies the process of creating components and of incorporating and utilizing components from a variety of developers. VCSE menu options accessed from the **Tools** menu enable you to:

- Use a wizard to create an new interface specification
- Use a wizard to create a VCSE component and a project to build the component
- Create a project to support the creation of a VCSE component
- Install and a view components on the local system
- Download and install new or updated components from the ADI web site
- Add an installed component to a project

Tool Chain Integration

You tell VisualDSP++ to produce a VCSE component library from the **Project** page of the **Project Options** dialog box. Under **Type**, select **VCSE component library**. Specify VCSE compiler options from the **VIDL** page of the **Project Options** dialog box. Specify IDL font and color preferences for editing on the **Editor** page of the **Preferences** dialog box.

Wizards

VCSE wizards lead you through various tasks.

- New Component Project Wizard

Creating a new VCSE component project is simple with the New Component Project wizard. After making the required decisions in the wizard, a new VCSE Component Library project is created and added to the current workspace. In addition, the IDL source code describing your component is generated and added to the project.

- New Component Package Wizard

You create a new component package with the New Component Package wizard. Select an XML component manifest, review component information, and then include files in the component package.

Creating a new interface is easy with the New Interface wizard. You define methods and parameters for the new interface, and the wizard generates the IDL source code for that interface and adds it to the current project.

Once you have downloaded and installed a component onto your system, you can easily add the component to a project.

Component Manager

You can view the list of currently installed components, add installed components to a project, and interactively view and download new components from the Analog Devices Web site. Use the Component Manager to browse components at various locations.

From the Component Manager dialog box, you can:

- Browse components

View the list of installed components on your system or the new and updated components on the ADI Web site. Each component includes a description.

- Filter your view of the components

Sort the component list by name, category, company, status, supported processor, or implemented interface.

- Install components

Install them directly from the Analog Devices Web site or from a third party. You must download the component to your PC from a Web site, or copy the component to your PC by any means.

- Uninstall components from your PC

Structure of VCSE

VCSE specifies the requirements that a component must meet, and provides a set of tools to help ensure that a component conforms to the ADI component standard for DSP algorithms and other objects. VCSE greatly simplifies the task of enabling components within a system to communicate, and provides a degree of abstraction that offers greater flexibility in the choice and use of components.

VCSE support for DSP algorithms makes it much easier for integrators to exploit algorithms from one or more vendors. This support also provides the algorithm developer with a standardized framework to make algorithms more interoperable and usable at a minimal cost.

VCSE provides a set of specifications and tools that comprise:

- A software architecture that is designed to be efficient and effective for DSP applications and processors. The language-neutral architecture provides support for inter-working between software written in assembly, C, and C++. The architecture is designed to operate within multiple environments such as single or multi-threaded applications.
- A component model that provides encapsulation (hiding of the implementation and other information) of the algorithms and objects, and supports the idea of abstract interfaces and a single inheritance model. The VCSE component model is specifically designed for use within DSP and embedded applications.
- An Interface Definition Language (IDL) that supports simple bindings for C, C++, and assembly. The IDL is supported by the VCSE IDL (VIDL) compiler, which processes the IDL specified component and interface definitions and generates interface headers, component shells, and HTML-based documentation for the component.

IDL incorporates support for documenting the interfaces, and enables standardized documentation to be generated, which allows other statements about the interface to be automatically validated or even generated.

- A set of rules and guidelines to which each component must adhere, if it is to be a conforming component or algorithm. Validation of conformance to some of these rules is effected automatically by VisualDSP++.
- An interface wizard that provides a visual user interface to allow the object or algorithm provider to define interfaces. The interface wizard generates an IDL specification of the interface, which you can then compile by using the VIDL compiler.

- Support for the inclusion of VCSE components in a project and the display of associated component documentation

Each VCSE component can be packaged as a compressed package, called a component package file (`.VCP`). You can install this file into VisualDSP++ by using the same techniques used to install and identify debug-targets from third-party suppliers. Each such package contains the component interface headers, documentation, and (optionally) the implementation and any necessary license information. Packages that do not contain the implementation provide a means of promoting a component or algorithm in a form that is convenient for VisualDSP++ developers.

Interface Definition Language (IDL) and Compiler

VCSE supports an Interface Definition Language (IDL) and compiler that enable developers to specify and then create and use components without having to become familiar with the detail of the model and its mechanisms. The VIDL compiler processes the specification of the interfaces that a component supports and generates the framework code needed to implement the component. The developer of the component can then concentrate on providing the implementation of the methods that the component is to provide. In addition, the VIDL compiler can generate a simple test harness to help in testing the component.

The VIDL compiler compiles the supplied IDL definitions of interfaces and components and generates up to four possible output items. These items can be emitted for each interface:

- An interface header file that can be used by a client of the interface to request services from the interface. Each interface header file can be used within assembly, C, and C++ source files.
- An interface component shell, which provides a standard framework for providing the implementation of each interface supported by the component in the chosen implementation language. This shell supports the interface but leaves the implementation of each method to the developer. Consequently, the developer is free to concentrate on the implementation of interface services without having to know the details of the VCSE binary standard.

The binary standard requires all functions within the interface to obey the C runtime model. The generated assembler shell for an interface ensures that the requirement is met by using the appropriate macros in the generated code. The component shell can be generated in the assembly, C, or C++ language.

- HTML-based documentation of the component and all of its supported interfaces in a standardized way. The documentation is derived from the IDL definition and the embedded auto-doc comments that the developer is encouraged to provide as part of the IDL definition.
- An .XML file, which can be used by the New Component Package Wizard when a component is packaged for distribution.

The IDL language supports all the standard C/C++ base types as well as arrays, structs, and enums, and the use of typedef. The only explicit pointer types that IDL supports are interface pointers. All *out* parameters are mapped to arrays or pointers. All *in* arrays and structs are also mapped to arrays and pointers.

The VIDL compiler inputs `.IDL` files and produces the files shown in Figure 1-4.

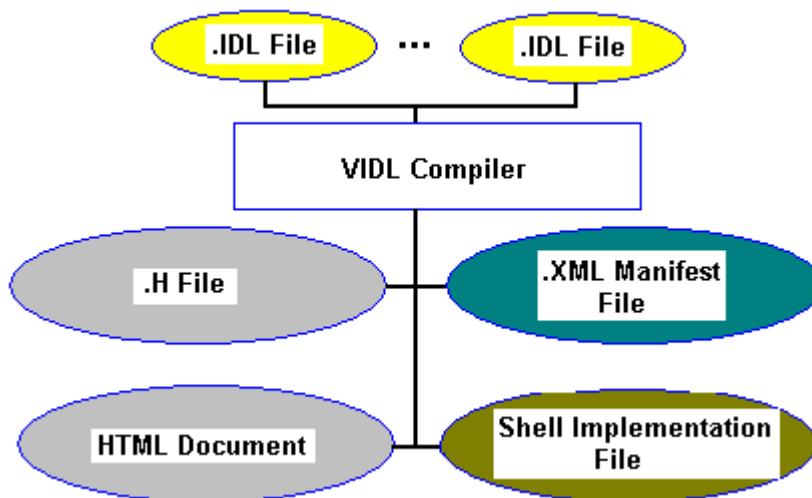


Figure 1-4. Files Produced by the VIDL Compiler

DSP Projects

Your goal is to create a program that runs on a single processor or a multi-processor system (emulator sessions only). All your development in VisualDSP++ occurs within a project.

What is a Project?

The project is the structure in which you build the DSP program.

The term *project* refers to the collection of source files and tool configurations used to create a DSP program. A project file (`.DPJ`) stores program build information.

Use the **Project** window to manage projects from start to finish. Within the context of a DSP project, you can:

- Specify DSP code development tools
- Specify project-wide and individual-file options for Debug or Release configurations of project builds
- Create source files

VisualDSP++ facilitates movement among editing, building, and debugging activities.

VisualDSP++ provides flexibility in how you set up projects. You configure settings for DSP code development tools and configurations, and specify build settings for the project and for individual files. You can set up folders that contain your source files. A project can include VDK support.

Project Options

You specify project options, which apply to the entire DSP project. [Figure 1-5](#) shows the top of the Project Options dialog box.

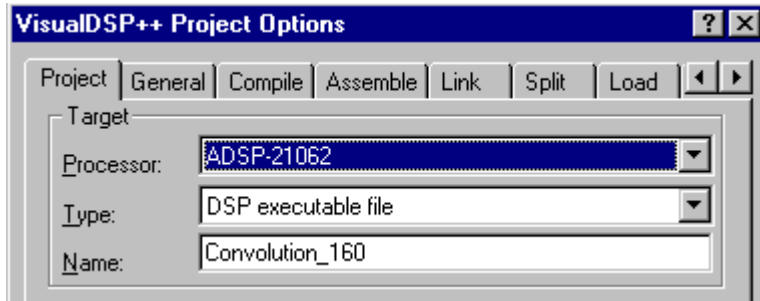


Figure 1-5. Top Portion of the Project Options Dialog Box

For each code development tool (compiler, assembler, linker, and loader), a tabbed page provides options that control how each tool processes inputs and generates outputs. The available pages depend on your target. Options correspond to an individual tool's command line switches. You can define these options once or modify them to meet changing development needs.

 You can also access the tools from the operating system's command line.

Project options also specify the following information.

- Project target
- Tool chain
- Output file directories
- Post-build options

Makefiles

You can use a makefile (.MAK) to automate builds within VisualDSP++. The output make rule is compatible with the `gmake` utility (GNU Make V3.77 or higher) or other make utilities. VisualDSP++ generates a project make file that controls the orderly sequence of code generation in the target. You can also export a makefile for use outside of VisualDSP++. For more information about makefiles, go to:

<http://www.gnu.org/manual/make/>

A project can have multiple makefiles, but only one makefile can be enabled (active).

The project in [Figure 1-6](#) includes an active makefile (indicated by ).

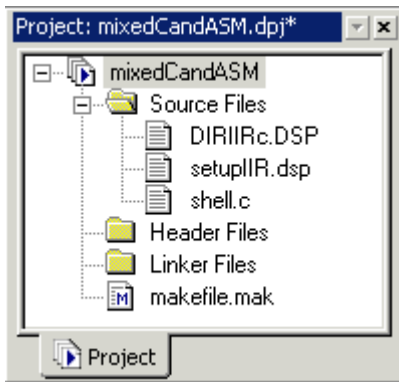


Figure 1-6. Enabled Makefile makefile.mak

The active makefile, with its explicit `gmake` command line, builds the project. When no makefile is enabled for a project, VisualDSP++ uses specifications configured in the **Project Options** dialog box.

DSP Projects

You can view a makefile's command line. Use the Configuration box, shown in [Figure 1-7](#), to change the makefile's target.

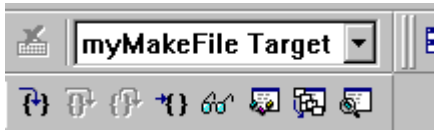


Figure 1-7. Configuration Box

When you close a project, the make commands and the target list associated with each makefile are serialized into the project file (.DPJ).

Rules

You can enable only one makefile when you build a project. If you enable more than one makefile, VisualDSP++ generates an error message. After you build your project with an external makefile, the executable is not automatically loaded (even when this option is configured).

Output Window

Make command error messages and standard output appear in the **Output** window. Double-clicking on an error-message position opens the makefile in an editor window to the line of code causing the error.

Keywords in the makefile are syntax colored.

Note: The error message format of gmake is parsed correctly when you double-click on an error message. If you use another make utility, the double-click feature does not function.

Example Makefile

An example of a makefile is shown below.

```
# Generated by the VisualDSP++ IDDE

# Note: Any changes made to this Makefile will be lost the next
# time the matching project file is loaded into the IDDE. If you
# wish to preserve changes, rename this file and run it
# externally to the IDDE.

# The syntax of this Makefile is such that GNU Make v3.77 or
# higher is required.

# The current working directory should be the directory in which
# this Makefile resides.

# Supported targets:
#   Debug
#   Debug_clean
#   Release
#   Release_clean

# Define ADI_DSP if it is not already defined. Define this
# variable if you wish to run this Makefile on a host other than
# the host that created it and VisualDSP++ may be installed in a
# different directory.

ifndef ADI_DSP

ADI_DSP=C:\Program Files\Analog Devices\VisualDSP

endif

# $VDSP is a gmake-friendly version of ADI_DIR

empty:=

space:= $(empty) $(empty)

VDSP_INTERMEDIATE=$(subst \,/,$(ADI_DSP))
```

DSP Projects

```
VDSP=$(subst $(space),\$(space),$(VDSP_INTERMEDIATE))

# Define the command to use to delete files (which is different
# on Win95/98 and Windows NT/2000)

ifeq ($(OS),Windows_NT)

RM=cmd /C del /F /Q

else

RM=command /C del

endif

#

# Begin "Debug" configuration

#

ifeq ($(MAKECMDGOALS),Debug)

Debug : ./debug/mean.dxe

./debug/mean.doj :./mean.c ../../../include/stdio.h

$(VDSP)/cc21k -c .\Mean.c -g -proc ADSP-21062 -o
.\Debug\Mean.doj

./debug/benchmark.doj :./benchmark.asm
../../../include/asm_sprt.h ../../../include/def21060.h

$(VDSP)/easm21k.exe -proc ADSP-21062 -o .\Debug\benchmark.doj -g
.\benchmark.asm

./debug/mean.dxe :./debug/mean.doj ./debug/benchmark.doj

$(VDSP)/cc21k.exe .\Debug\Mean.doj .\Debug\benchmark.doj -proc
ADSP-21062 -L .\Debug -flags-link -od,.\Debug -o .\Debug\Mean.dxe
```



```

endif

ifeq ($(MAKECMDGOALS),Debug_clean)
Debug_clean:$(RM) ".\Debug\Mean.doj"
    $(RM) ".\Debug\benchmark.doj"
    $(RM) ".\Debug\Mean.dxe"
    $(RM) ".\Debug\*.ipa"
    $(RM) ".\Debug\*.opa"
    $(RM) ".\Debug\*.ti"

endif

# Begin "Release" configuration

#

ifeq ($(MAKECMDGOALS),Release)

Release : ./release/mean.dxe

./release/mean.doj :./mean.c

$(VDSPP)/cc21k -c .\Mean.c -O1 -proc ADSP-21062 -o
.\Release\Mean.doj

./release/benchmark.doj :./benchmark.asm

$(VDSPP)/easm21k.exe -proc ADSP-21062 -o .\Release\benchmark.doj
.\benchmark.asm

./release/mean.dxe :./release/mean.doj ./release/benchmark.doj

$(VDSPP)/cc21k.exe .\Release\Mean.doj .\Release\benchmark.doj
-proc ADSP-21062 -L .\Release -flags-link -od,.\Release -o
.\Release\Mean.dxe

endif

ifeq ($(MAKECMDGOALS),Release_clean)

```

```
Release_clean:
    $(RM) ".\Release\Mean.doj"
    $(RM) ".\Release\benchmark.doj"
    $(RM) ".\Release\Mean.dxe"
    $(RM) ".\Release\*.ipa"
    $(RM) ".\Release\*.opa"
    $(RM) ".\Release\*.ti"

endif
```

Project Configurations

By default, a project includes two configurations, Debug and Release, described in the following table. In previous software releases, the term *configuration* was called “build type.”

Table 1-4. Default Project Configurations

Configuration	Description
Debug	Builds a project that enables you to use VisualDSP++ debugging capabilities
Release	Builds a project with optimization enabled

Available configurations appear in the configuration box, which is part of the **Project** toolbar, as shown in [Figure 1-8](#).



Figure 1-8. Configuration Box

 You cannot delete the Release or Debug configuration.

Customized Project Configurations

You can add a configuration to your project. A customized project configuration can include various project options and build options to help you develop your project. Figure 1-9 shows a customized configuration (**Version2**) listed in the configuration box.

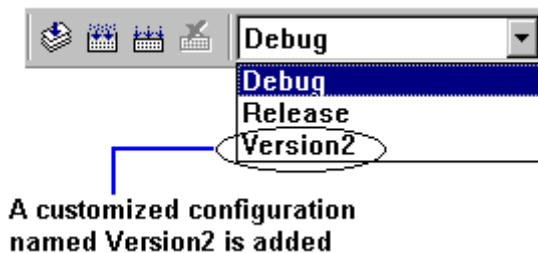


Figure 1-9. Customized Configuration Version2

Project Build

The term *build* refers to the performance of processing operations (such as preprocessing, assembling, and linking) on projects and files. During a build, VisualDSP++ processes project files that have been modified since the previous build as well as project files that include modified files.

A build differs from a *rebuild all*. When you run the **Rebuild All** command, VisualDSP++ processes all the files in the project, regardless of whether they have been modified.

Building a project builds all outdated files in the project and enables you to make your program. An *outdated file* is a file that has been modified since the last time it was built or a file that includes a modified file. For example, if a C file that has not been modified includes a header file that has been modified, the C file is out of date.

VisualDSP++ uses *dependency* information to determine which files, if any, must be updated during a build.



Note the following.

- A file with an unrecognized file extension is ignored at build time.
- If an included header file is modified, VisualDSP++ builds the source files that include (`#include`) the header file, regardless whether the source files have been modified since the previous build.
- File icons in the **Project** window indicate file status (such as excluded files or files with specific options that override project settings).

Build Options

You can specify options for the entire project and for individual files.

[Table 1-5](#) describes these build options.

Table 1-5. Build Options

Options	Description
Project-wide	You specify these options from a tabbed page (for example, Compile or Link) for each of the DSP code development tools.
Individual file	These settings override project-wide settings.
Custom build step	For maximal flexibility, you can edit the command line(s) issued to build a particular file. For example, you might call a third-party utility.

File Building

You build a single file to compile or assemble the file and to locate and remove errors. The build process updates the source file's output (.OBJ file), and updates the output file's debug information. Building a single file is very fast. Large projects, however, may require hours to build.

You can build multiple files that you select. Similar to building an individual file, this process enables you to update output files.

If you change a common header file that requires a full build, you can build only the current file to ensure that your change fixes the error in the current file.

Post-Build Options

Post-build options are typically DOS commands that execute after a project has been successfully built. These commands invoke external tools.

For example, you can use a post-build command to copy the final output file to another location on the hard drive or to invoke an application automatically.

Automatically copying files and cleaning up intermediate files after a successful build can be very useful.

Command Syntax

Depending on your operating system, you must place “cmd /C” or “command /C” at the beginning of each DOS command line.

For example, to execute `copy a.txt b.txt`, use one of the commands shown in the [Table 1-6](#). The “C” after the slash in the commands must be uppercase.

Table 1-6. Operating System and Required Command Syntax

Operating System	Command
Windows 98	command /C copy a.txt b.txt
Windows Me	command /C copy a.txt b.txt
Windows NT	cmd /C copy a.txt b.txt
Windows 2000	cmd /C copy a.txt b.txt
Windows XP	cmd /C copy a.txt b.txt

Project Dependencies

Dependency data determines which files must be updated during a build. The following data is an example of dependency information.

```
..\..\include\cplus\cstddef
..\..\include\cplus\exception
..\..\include\cplus\new
..\..\include\cplus\xstddef
..\..\include\def21060.h
..\..\include\limits.h
..\..\include\cplus\stddef
..\..\include\stdio.h
..\..\include\string.h
..\..\include\VDK_Internals.h
..\..\include\VDK_Public.h
..\..\include\yvals.h
```

Project Rules

The **Project** window displays a project's files, as shown in [Figure 1-10](#).

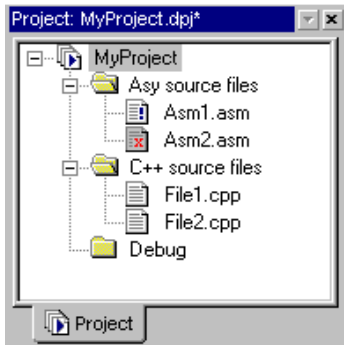


Figure 1-10. Example of Project Files

The following rules dictate how files and subfolders behave in the **Project** window's file tree.

- You can include any file in a project.
- Only one .LDF file is permitted.
- You cannot add the same file into the same project more than once.
- Only one project (project node) is permitted.
- A file with an unrecognized file extension is ignored at build time.
- When you add a file to a project, the file is placed in the first folder configured with the same extension. If no such folders are present, an added file goes to the project level.

VisualDSP++ Help System

The VisualDSP++ Help system is designed to help you obtain information quickly. You can use the Help system's table of contents, index, full-text search function, bookmark function, and extensive hyperlinks to jump to topics.

VisualDSP++ Help is a merge of several Help systems (.CHM files). Each is identified with a book icon  in your product installation's **Help** folder.

Most of the Help system comprises VisualDSP++ tools *manuals*, such as the Assembler and Preprocessor manuals. These manuals are also provided in PDF format (on installation disk) for printing and are available from Analog Devices as printed books.

Some .CHM files support *pop-up* messages for dialog box controls (buttons, fields, and so on). These messages, which appear in little yellow boxes, compose part of the context-sensitive Help in VisualDSP++.

The Help system describes the VisualDSP++ *user interface*. Help files include concepts, procedures, and reference information. Each toolbar button, menu-bar command, and debugging window in VisualDSP++ is linked to a topic in one of these files.

Using the Help Window

The Help window comprises three parts:

- The *Navigation pane* provides tabbed pages (**Contents**, **Index**, **Search**, and **Favorites**) that show different navigational views.
- The *Viewing pane* displays the selected object (topic, Web page, video, .PDF file, application).
- *Toolbar buttons* enable you to navigate or specify options.

Figure 1-11 shows the parts of the VisualDSP++ Help window.

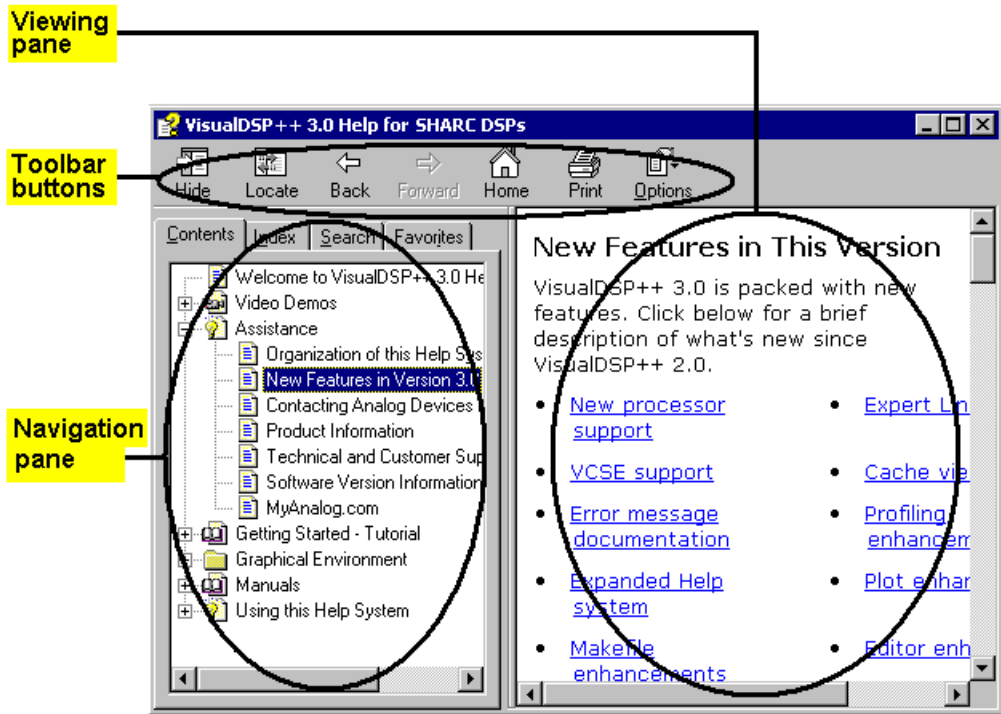


Figure 1-11. Parts of the VisualDSP++ Help Window

Invoking Online Help

You can invoke online Help from VisualDSP++ or from the Windows Start button. You can also access Help manually via Windows Explorer.

To access online Help from VisualDSP++, choose **C**ontents, **S**earch, or **I**ndex from the **H**elp menu.

VisualDSP++ Help System

To access online Help from the Windows **Start** button, click the **Start** button and choose **Programs, VisualDSP, and VisualDSP++ Documentation**.

The Help function is programmed to look for the Help system in the VisualDSP++ Help folder.

By default, the VisualDSP++ software installation procedure places the complete set of Help files (except the *Getting Started Guide*) in the folder VisualDSP\Help.

If you receive an error message after invoking Help, the Help system:

- May not have been loaded onto your PC
- May have been deleted
- May reside in a directory other than the default directory

To locate the help (.CHM) files manually, use the Windows **Search** function as follows.

1. Write down the Help file (.CHM) named in the error message.
2. From the Windows **Start** button, choose **Search and For Files or Folders**. Enter the name of the .CHM file from step 1.
3. After locating the file, launch it manually by clicking the file name from the **Search Results** window or from Windows Explorer.

Viewing Context-Sensitive Help

You can view context-sensitive Help (help pertinent to your current activity) for various items in VisualDSP++.

Context-sensitive Help in VisualDSP++ is linked to toolbar buttons, menu commands, windows, and dialog box items.

Viewing Menu, Toolbar, or Window Help

1. Click the toolbar's Help button  or press **Shift+F1**.
The mouse pointer becomes a Help pointer .
2. Move the Help pointer over a menu command, toolbar button, or window.
3. Click the mouse to open the Help window. A description of the object appears in the right panel.

Viewing Dialog Box Button or Field Help

Perform one of these actions:

- Select a field or button in a dialog box and press **F1** or **Shift+F1**.
- Click the Question-Mark button  in the top-right corner of the dialog box.

The mouse pointer becomes a Help pointer .

Move the Help pointer over a dialog box control (button or field) and click the mouse. A description of the object appears in a yellow pop-up window.

- Position the mouse pointer over a label or control (button or field) in a dialog box and right-click.

A **What's This** button  appears. Move the mouse pointer over the **What's This** button and click.



“What's This” Help is not configured for all items.

Viewing Window Help

1. Click the window to make it active.
2. Press the F1 key to open the Help window.

A description of the window appears in the right panel.

Using Help Window Navigation Buttons

You can move through the Help system and view Help topics by using the Help window's navigational aids, as shown in [Figure 1-12](#).

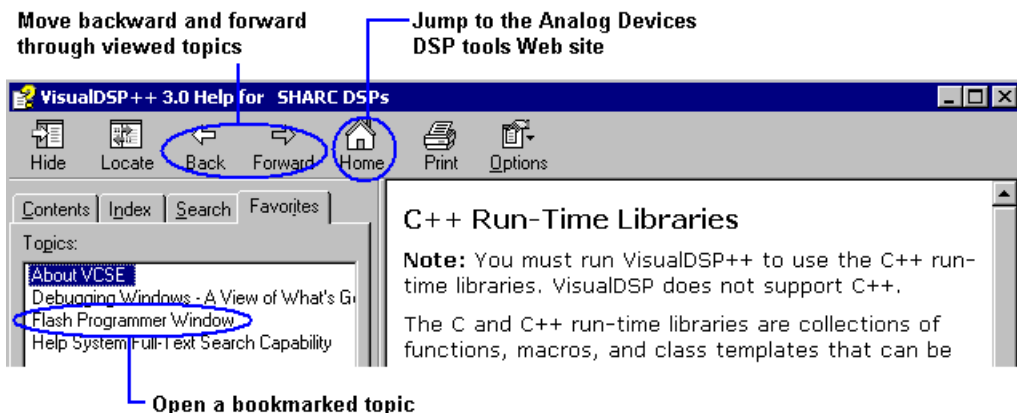


Figure 1-12. Help Window's Navigational Aids

Other standard Microsoft HTML Help buttons are described in [Table 1-7](#).

Table 1-7. Standard Microsoft HTML Help Buttons

Button	Purpose
 Hide	Hides the Help window's left pane. This button narrows the Help window.
 Show	Displays the Help window's left pane. This button restores a full view after you click Hide .
 Locate	Highlights the name of the current topic on the Contents page (left pane). After you jump around the Help system, this button shows the current topic's relation to other topics.

Copying Example Code from Help

You can copy code from the Help system and then paste it into your application. Be aware that the copied text may carry unwanted control codes. For example, if you copy a hyphen with a parameter, the actual code of the copied hyphen may be an ASCII 0x96 instead of an ASCII 0x2D. The hyphen may look OK, but it will cause an error when the command is completed.

Printing Help

You can print a specific Help topic, or you can print multiple Help topics (an entire section of online Help).

Table 1-8. How to Print Help Topics

To print	Do this
Current topic	Right-click within the help topic and choose Print .
Selected topic	On the Contents page: Right-click the topic  and choose Print .
Entire section of Help	On the Contents page: Right-click a book icon  or  and choose Print . Then choose Print the selected heading and all subtopics .

Tip: From the Help window's **Contents** page, click  , located at the top of the window.

Bookmarking Frequently Used Help Topics

You can bookmark a topic in online Help just like you might bookmark a page in a book. This feature is also called setting up favorite places.

Note: Each time you bookmark a Microsoft HTML Help topic, a record is recorded in the file, `HH.DAT`. This file not only records VisualDSP++ Help bookmarks, but also the bookmarks you place in other application Help systems that use `.CHM` files.

Once you have placed a bookmark onto a topic, you can view a list of bookmarked topics and quickly open one.

Placing a Bookmark at a Topic

1. Display the topic.
2. On the left side of the Help window, click the **Favorites** tab.
3. Click **Add**.

The Help system adds the topic and displays it in the alphabetized list.

You can remove a bookmark by selecting the name and clicking **Remove**.

Opening a Bookmarked Topic

1. On the left side of the Help window, click the **Favorites** tab.
2. Perform one of these actions:
 - Double-click the topic.
 - Select the topic and click **Display**.

Navigating in Online Help

To move around in the Help system, you can click the following.

- **A hyperlink within text.** The text is underlined and displayed in a color that is different from the regular black text.
- **A topic listed under a See Also heading.** The text is underlined and displayed in a color that is different from the regular black text.
- **A mini button or its associated text.** The button is a small gray square and the underlined text is in a different color.

- A topic name on the Contents page ([Figure 1-13](#))

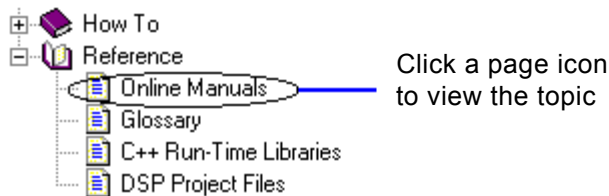


Figure 1-13. Contents Page – Online Manuals Topic

- An index entry on the Index page ([Figure 1-14](#))

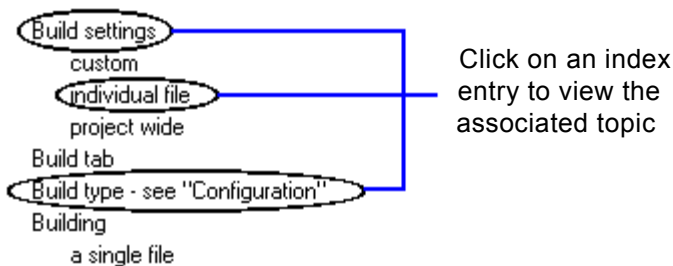


Figure 1-14. Index Entries on the Index Page

- A topic name on the Search page. The bottom portion of the Search page displays the located topics (hits) that include your search string.

Using the Search Features

VisualDSP++ Help provides both full-text and advanced search capabilities to help you find information.

Help System Search Rules

Different rules apply for each type of search.

Rules for Full-Text Searches

Observe these rules when formulating queries:

- Searches are not case-sensitive.
You can type your search in uppercase or lowercase characters. You can search for any combination of letters (a–z) and numbers (0–9).
- Searches ignore punctuation marks such as the period, colon, semi-colon, comma, and hyphen.
- Group the elements of your search by using double quotes or parentheses to set apart each element.
- You cannot search for quotation marks.

Note that if you are searching for a file name with an extension, group the entire string in double quotes, (“filename.ext”). Otherwise, the period breaks the file name into two separate terms. The default operation between terms is AND, which creates the logical equivalent to filename AND ext.

Rules for Advanced Searches

These rules apply to advanced searches:

- Expressions in parentheses are evaluated before the rest of the query.
- If a query does not contain a nested expression, it is evaluated from left to right. For example, “folder NOT file OR project” finds topics containing the word “folder” without the word “file,” or topics containing the word “project.” The expression “folder NOT (file OR project)”, however, finds topics containing the word “folder” without either of the words “file” or “project.”
- You cannot nest expressions deeper than five levels.

Full-Text Searches

The full-text search capability enables you to locate every occurrence of a text string within the Help system. You specify a particular word or phrase, and the search function finds only the topics that contain that word or phrase.

You can search previous results, match similar words, and search through the topic titles only.

A basic search consists of the word or phrase that you want to locate. You can use similar word matches, a previous results list, or topic titles to further define your search.

You can run an advanced search, which uses Boolean operators and wild-card expressions to further narrow the search criteria. [Figure 1-15 on page 1-69](#) shows an example of a Boolean search that uses the “AND” operator.

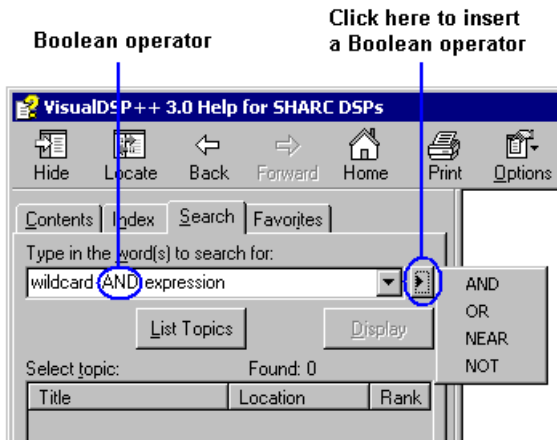


Figure 1-15. Boolean Search with the “AND” Operator

To find information with full-text search:

1. Click the Help viewer’s **Search** tab.
2. In **Type in the word(s) to search for**, type the word or phrase you want to find.
3. Select **Search previous results** to narrow your search.
4. Select **Match similar words** to find words that are similar to the search string.
5. Select **Search titles only** to search only the topic titles.
6. Click the **Options** button  at the top of the Help Viewer window to highlight all instances of search terms found in topic files. Then choose **Search Highlight On**.
7. Click **List Topics**, select the topic you want, and then click **Display**.

Note that you can sort the topic list by clicking the **Title**, **Location**, or **Rank** column heading.

Advanced Search Techniques

You can use the following search techniques to narrow your searches.

- Wildcard expressions
- Boolean operators
- Nested expressions

Using Wildcard Expressions

Wildcard expressions enable you to search for one or more characters by using a question mark or asterisk. [Table 1-9](#) describes the results of these different kinds of searches.

Table 1-9. How to Use Wildcard Expressions to Define a Search

To find	Example	Results
A single word	project	Locates topics that contain the word “project.” Other grammatical variations, such as “projects” are located.
A phrase	“project window” (note the quotation characters) project window	Locates topics that contain the literal phrase “project window” and all its grammatical variations. Without the quotation characters, the query is equivalent to specifying “project AND window,” which finds topics containing both of the individual words, instead of the phrase.
Wildcard expressions	link* -or- .C??	Locates topics that contain the terms “linker,” “linking,” “links,” and so on. The asterisk cannot be the only character in the term. Locates topics that contain the terms “.CPP” or “.CXX.” The question mark cannot be the only character in the term.

Using Boolean Operators

Use the Boolean AND, OR, NOT, and NEAR operators to precisely define your search by creating a relationship between search terms.

Insert a Boolean operator by typing the operator (AND, OR, NEAR, or NOT) or by clicking the arrow button.

Note that if you do not specify an operator, AND is used. For example, the query “call stack” is equivalent to “call AND stack.”

Table 1-10 describes the results of using Boolean operators to define a search.

Table 1-10. How to Use Boolean Operators to Define a Search

To find	Example	Results
Both terms in the same topic	new AND plot	Locates topics that contain both the words “new” and “plot”
Either term in a topic	new OR plot	Locates topics that contain either the word “new” or the word “plot” or both
The first term without the second term	new NOT plot	Locates topics that contain the word “new”, but not the word “plot”
Both terms in the same topic, close together	new NEAR plot	Locates topics that contain the word “new” within eight words of the word “plot”

You cannot use the |, &, or ! characters as Boolean operators. You must use OR, AND, or NOT.

Using Nested Expressions

Use nested expressions to create complex searches for information. For example, new AND ((plot OR waterfall) NEAR window) finds topics containing the word “new” along with the words “plot” and “window” close together, or topics containing “new” along with the words “waterfall” and “window” close together.

Viewing Online Documents

VisualDSP++ includes three types of user documentation, described in [Table 1-11](#).

Table 1-11. Types of User Documentation

Files	Purpose
.CHM	VisualDSP++ online Help system files and VisualDSP++ manuals are provided in Microsoft HTML Help format. Installing VisualDSP++ automatically copies these files to the <code>VisualDSP\Help</code> folder. Online Help is ideal for searching the entire tools manual set. Invoke Help from the VisualDSP++ Help menu or via the Windows Start button. The .CHM files require Internet Explorer 4.0 (or higher) or the installation of a component that provides a .CHM file viewer.
.PDF	Manuals and data sheets in Portable Documentation Format are located in the installation CD's <code>Docs</code> folder. Viewing and printing a .PDF file requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). Running <code>setup.exe</code> on the installation CD provides easy access to these documents. You can also copy PDF files from the installation CD onto another disk.
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation is located on the installation CD in the <code>Docs\Reference</code> folder. Viewing or printing these files requires a browser, such as Internet Explorer 4.0 (or higher). You can copy these files from the installation CD onto another disk.



The VisualDSP++ software installation procedure does not copy PDF versions of books and data sheets or supplemental reference documentation to the VisualDSP directory.

Printing Online Documents

You can print documents from the VisualDSP++ Tools Installation CD-ROM.

To print online documents:

1. Insert the VisualDSP++ Tools Installation CD-ROM in your CD-ROM drive.
2. Open the **Docs** folder by using one of these options:
 - From the **VisualDSP++ Tools Installation** main menu, click **View Documentation**. (If the main menu does not appear, run `setup.exe`.)
 - In Windows Explorer, select your CD-ROM drive (for example, **d:**) and open the **Docs** folder.
3. Open the folder where the document is located.

The `Data Sheets` folder contains copies of DSP data sheets.

The `Hardware Manuals` folder contains copies of hardware manuals.

The `Reference` folder includes the HTML files that comprise the Dinkum Abridged C++ library and the FlexLM network license documentation.

The `Tools Manuals` folder contains copies of VisualDSP++ tools manuals.

4. Double-click the document that you want to print. Selecting a PDF file opens Adobe Acrobat Reader and displays the document. Selecting an HTML file opens a browser and displays the document.
5. From the **File** menu, choose **Print** and specify the pages that you want to print (and other print options).

2 ENVIRONMENT

This chapter describes the features of the VisualDSP++ environment, including the main window, debugging windows, window operations, and customization.

The topics are organized as follows.

- “Parts of the User Interface” on page 2-1
- “VisualDSP++ Windows” on page 2-15
- “Window Operations” on page 2-43
- “Debugging Windows” on page 2-49

Parts of the User Interface

VisualDSP++ is an intuitive, easy-to-use user interface for programming Analog Devices DSPs. When you open VisualDSP++, the application’s main window appears. [Figure 2-1 on page 2-2](#) shows an example of the VisualDSP++ main window.

This work area contains everything you need to build, manage, and debug your DSP project. You can set up preferences that specify the appearance of application objects (fonts, visibility, and so on).

Parts of the User Interface

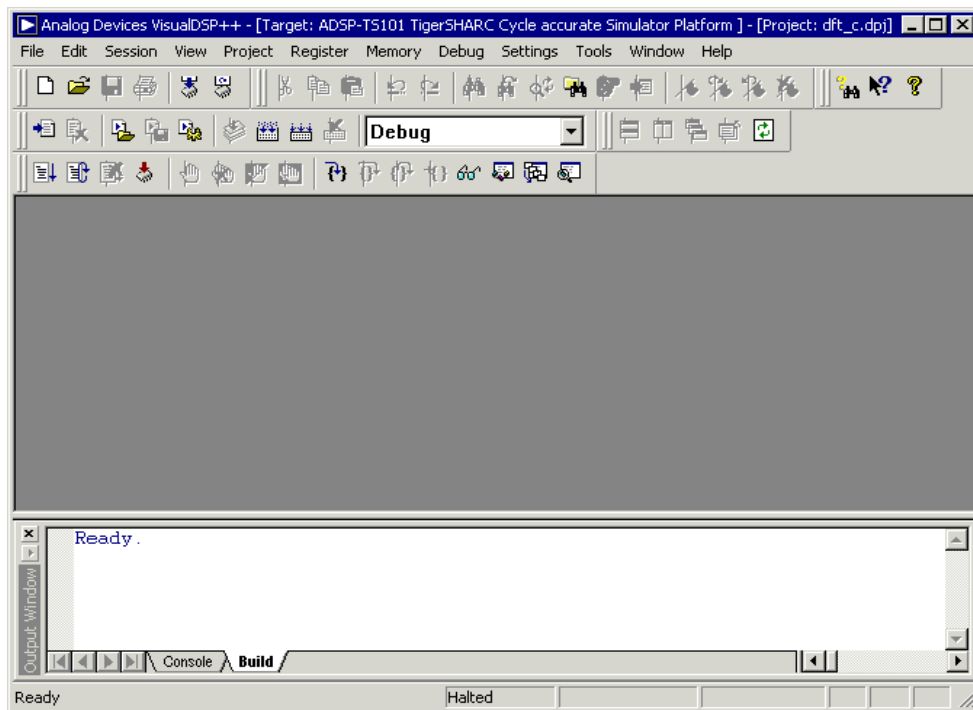


Figure 2-1. Example of VisualDSP++ Main Window

The VisualDSP++ main window includes these parts:

- Title bar
- Menu bar
- **Project** window
- Control menu
- Toolbars

- **Output** window
- Status bar

VisualDSP++ also provides many debugging windows to show you what is going on.

You need to learn only one interface to debug all your DSP applications.

VisualDSP++ supports ELF/DWARF-2 (Executable Linkable Format, Debugging Information Format) executable files. VisualDSP++ supports all executable file formats produced by the linker.

Title Bar

Figure 2-2 shows the different parts of the title bar.

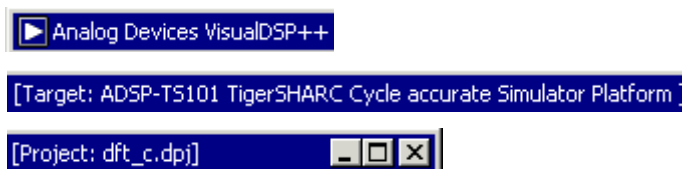


Figure 2-2. Example Title Bar (Split into Three Parts to Fit the Page)

The title bar includes these components:

- Control menu button 
- Application name – Analog Devices VisualDSP++
- Name of the active target
- Project name

Parts of the User Interface

- Filename (when an editor window is maximized in the main window)
- Standard Windows buttons

Clicking the control menu button opens the control menu, which contains commands for positioning, resizing, minimizing, maximizing, and closing the window. Double-clicking the control button closes VisualDSP++. The control menu and title bar right-click menu (see below) are identical.

Additional Information in Title Bars

A register window title bar displays its numeric format (such as hexadecimal). An editor window title bar displays the name of the source file.

Title Bar Right-Click Menus

A menu like the one below appears when you right-click within the VisualDSP++ title bar or within the title bar of a child (sub) window.



Figure 2-3. Right-Clicking in the Window's Title Bar

From the VisualDSP++ title bar's right-click menu, you can:

- Resize or move the application window
- Close VisualDSP++

Control Menu

Commands on a control menu (system menu, shown in [Figure 2-4](#)) move, size, or close a window.



Figure 2-4. VisualDSP++ Control Menu

Program Icons

Click a program icon to open a control menu.



Program icon for the application and debugging windows



Program icon for editor windows

When you place the mouse pointer over a control menu command, a brief description of the command appears in the status bar at the bottom of the application window.

Editor Windows

A floating editor window's control menu includes **Next**, which moves the focus to another window.

When an editor window floats in the main application window, its program icon resides at the left side of its title bar. When an editor window is maximized, the program icon resides at the left end of the menu bar.

Parts of the User Interface

Debugging Windows

Each debugging window has a control menu. You can open a debugging window's control menu only when the window is floating in the main window. For more information, see [“Debugging Windows” on page 2-49](#).

Menu Bar

The menu bar, shown in [Figure 2-5](#), appears directly below the application title bar and displays menu headings, such as **File** and **Edit**.

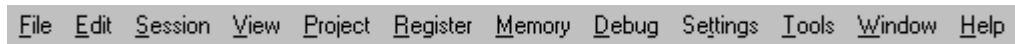


Figure 2-5. VisualDSP++ Menu Bar

To display menu commands and submenus, click a menu heading. You can also access many menu bar commands as follows.

- Click toolbar buttons
- Type keyboard shortcuts
- Right-click the mouse and choose a command from a context menu

Command Information

When the mouse pointer is over a menu bar command (or a toolbar button), a short description (tool tip) of the command appears in the status bar at the bottom of the main window.

Context-sensitive Help is available for each command.

To learn more about an individual menu command:

- Press **Shift+F1** or click the toolbar's Help button .

The pointer becomes a Help pointer .

- Move the Help pointer over a menu command.

If necessary, navigate through submenus.

- Click the mouse.

View the description of the command in the ensuing Help window.

Toolbars and User Tools

A toolbar is a set of buttons. You can run a command quickly by clicking a toolbar button.

Use toolbars to organize the tasks you use most often. Position the toolbars on the screen for fast access to the tools that you plan to use.

The application includes standard (built-in) toolbars. You can create custom toolbars.

Parts of the User Interface

Built-In Toolbars

Table 2-1 shows the standard (default) toolbars.

Table 2-1. Built-In Toolbars

Name	Toolbar
File	
Edit	
Help	
Project	
Window	
Debug	
Multiprocessor	

Table 2-1. Built-In Toolbars (Cont'd)

Name	Toolbar
User Tools	
Workspaces	

To obtain information about a tool, move the mouse pointer over the tool and press the F1 key.

Toolbar Customization

By default, nine standard toolbars appear near the top of the application window, below the menu bar.

You can change the appearance of toolbars by:

- Moving, docking, or floating the toolbars
- Adding or removing buttons to or from toolbars
- Displaying *cool look* buttons, *large buttons*, or both

You can also:

- Hide toolbars from view
- Add and delete custom-built toolbars

Parts of the User Interface

Toolbars: Docked vs. Floating

By default, toolbars are located under the application's menu bar. You can move them to the following locations.

- Over a docked window
- On the main window
- Anywhere on the desktop

When a toolbar is attached to a window, it is called a *docked* toolbar. You can tell when a toolbar is going to dock by the size and shape of its moving outline as you drag it. Its outline becomes slightly smaller than its floating outline. To prevent a toolbar from docking, press and hold the **Ctrl** key while dragging the toolbar to a new location.

You can detach a toolbar from a window and move it to another location anywhere on the desktop. A *floating* toolbar is a stand-alone window, as it is not docked. A docked toolbar does not show its name, but a floating toolbar displays its title.

Figure 2-6 shows a floating Help toolbar.



Figure 2-6. Example of a Floating Toolbar

Toolbar Button Appearance

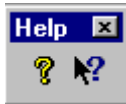
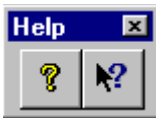
You can choose the appearance of the toolbar buttons. Two options, *cool look* and *large buttons*, provide slightly different button appearances.

The cool look option includes a pair of vertical bars on the toolbar's left side, but removes the square box from each button. The vertical bars visually separate toolbar buttons into groups (toolbars).

The large buttons option makes the area of each button larger.

Table 2-2 shows how small and large buttons appear with the cool look option turned off (disabled) and on (enabled).

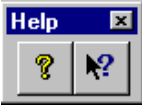
Table 2-2. Toolbars in Different Viewing Options

Option Settings	Docked	Floating
Cool look – Off Large buttons – Off		
Cool look – On Large buttons – Off		
Cool look – Off Large buttons – On		
Cool look – On Large buttons – On		

Toolbar Shape

You can change the shape of a floating toolbar. [Table 2-3](#) shows two toolbar shapes.

Table 2-3. Toolbars in Two Orientations

Horizontal	Vertical
	

Depending on the number of tools in the toolbar, you can create other length and width arrangements.

Toolbar Rules

When working with toolbars, be aware of these rules:

- You can customize a built-in toolbar (for example, you can remove a button from the **File** toolbar), but you cannot delete a built-in toolbar. You can reset the buttons in a built-in toolbar to their original default settings.
- You can change the name of a user-defined toolbar, but not the name of a built-in toolbar. For example, you cannot change the name of the **File** toolbar.

User Tools

Save time running commands by configuring user tools. You can configure up to ten user tools.

A user tool runs a command, which can:

- Contain parameters to launch an application
- Be a Tcl command

You access configured user tools from the **Tools** menu or from the **User Tools** toolbar, as shown in [Figure 2-7](#).



Figure 2-7. Default User Tools

When a user tool is configured, its menu name (label) appears in the **Tools** menu. The label also appears when you move the mouse pointer over a user tool button.

Status Bar

The status bar, located at the bottom of the main application window, provides various informational messages. [Figure 2-8](#) shows different information displayed on the status bar.

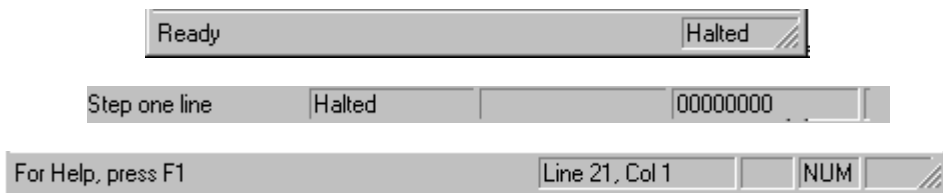


Figure 2-8. The Status Bar's Appearance Depends on Context

Parts of the User Interface

The type of information that appears in the status bar depends on your context (what you are doing).

When you move the mouse pointer over a toolbar button or a menu bar command, a brief description of the button or command appears.

While you are editing a file, the right side of the status bar provides editor window information, described in [Table 2-4](#).

Table 2-4. Status Bar Information While Editing

Item	Indicates
Line ###	Cursor current line number
Col ###	Cursor current column number
CAP	The keyboard's Caps Lock key is enabled
NUM	The keyboard's Num Lock key is enabled
SCRL	The keyboard's Scroll Lock key is enabled

VisualDSP++ Windows

From the application's main window, you can open a **Project** window, editor windows, an **Output** window, and various debugging windows.

Project Window

The **Project** window has a **Project** tab  .

When a project is VDK-enabled, the **Kernel** tab  also appears, as shown in [Figure 2-9](#).

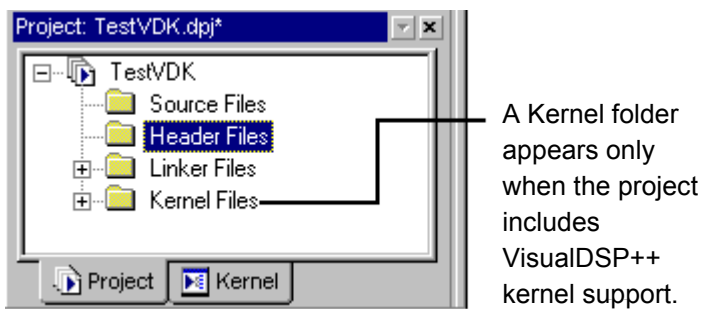


Figure 2-9. Kernel Tab in the Project Window

Project Tab

Use the **Project** tab to display a tree of your project's folders and files. Nodes are arranged in a hierarchy similar to the file structure in Windows Explorer.

Figure 2-10 shows the node hierarchy in the **Project** view.

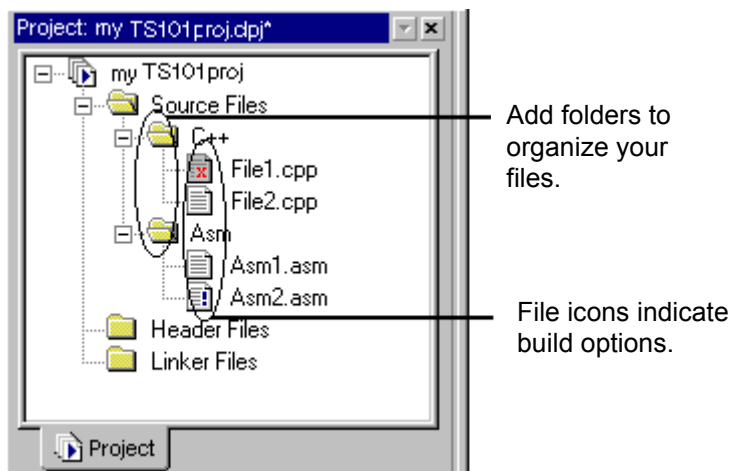


Figure 2-10. Node Hierarchy in the Project View

Project Nodes

The **Project** window comprises three types of nodes, described in [Table 2-5](#).

Table 2-5. Types of Nodes in the Project Window

Node	Icon	Description
Project		Only one project is permitted
Folder		A closed folder
		An opened folder, revealing its contents
File		A file that uses project settings
		A file with options that differ from the project options
		A file excluded from the current configuration

Project View Right-Click Menus

From the **Project** view, the right-click menus (also called popup menus or context menus) described in [Table 2-6](#) are available.

Table 2-6. Project View Right-Click Menus

From	Description
Node	The menu's content depends on the context (selected node).
Title bar	From this menu, you can dock, hide, or float the window. You can also view project properties.

Project Folders

Project window folders   organize files within a project. You can specify properties for folders.

Folders can be nested to any depth. Folders carry no attributes to the build process, as they do not reflect the file system. Folders do not appear in directory listings, as in Windows Explorer.

When you add files to the project tree with automatic file placement, each file is placed in the first folder that has been configured with the same file extension. After automatic placement, you can move a file anywhere manually.

To move a file out of one folder and into another folder, select the file and drag it onto the other folder.

Project Files

In the **Project** window, files are represented by the following icons.

Table 2-7. Icons in the Project Window

Icon	Description
	Files that use project options
	Files that use options that differ from project options
	Files excluded from the current configuration

The files appear in an expandable and collapsible node tree.

Source files are the C/C++ language or assembly language files in your project. Source files provide the project with code and data. You can add, delete, and modify source files.

Each project must include an `.LDF` file, which contains command input for the linker. If you do not include an `.LDF` file in the project, the project is built with a default `.LDF` file.

A DSP project can also include data files and header files.

File Associations

VisualDSP++ associates the file extensions in [Table 2-8 on page 2-20](#) as the input to particular DSP code development tools.

Table 2-8. File Associations

Tool	File Extensions
Compiler	.C, .CPP, and .CXX
Assembler	.ASM, .S, and .DSP
Linker	.LDF, .DLB, and .DOJ



Note the following:

- VisualDSP++ is case insensitive to file extensions.
- VisualDSP++ supports C++, but VisualDSP does not support C++.

Automatic File Placement

With the **Project** view selected, automatic file placement enables you to drag and drop files into designated folders in the **Project** window. This feature saves time when you add files to a project.

Folder properties that you specify and file placement rules determine where files are placed. By default, project folders are associated with the file extensions listed in [Table 2-9](#).

Table 2-9. Files Associated with Project Folders

Folder	Default Associations
Source Files	.C, .CPP, .CXX, .ASM, .DSP, .S
Header Files	.H, .HPP, .HXX
Linker Files	.LDF, .DLB, .DOJ
Kernel Files	.VDK

File Placement Rules

The following rules dictate file placement when you add files to a project.

- Dragging and dropping files

When you drag and drop a file into the **Project** window, the file is added to the first folder associated with the file's extension.

- Using menu commands to add files

Files are added to the folders that you select in the **Project** window. If you add a file to a project that has no folders, the file is added at the project level (root level).

If you select the project node or a file node, the file is added to the first folder associated with the file's extension.

Example

You create a folder labeled “C Source Files” and specify it with .C, .CPP, and .CXX file extensions. You create a second folder labeled “Asm Files” and associate it with .ASM files.

If you drag three files (file1.cpp, file1.asm, and file2.c) into the **Project** window, file1.cpp and file2.c go into the C Source Files folder, and file1.asm goes into the Asm Files folder.

Note: After automatic file placement, you can manually move a file anywhere by selecting and dragging the file.

Kernel Tab

The **Kernel** tab of the **Project** window is available only to VDK-enabled projects.

Use the **Kernel** tab to view, add, modify, and delete kernel elements such as thread types, priorities, semaphore, and events. VisualDSP++ automatically updates `vdk_config.cpp` and `vdk_config.h` to reflect the changes that you make.

The example in [Figure 2-11 on page 2-23](#) shows an expanded view of the kernel elements for a VDK-enabled project.

Refer to the *VisualDSP++ Kernel (VDK) User's Guide* for complete details about VDK.

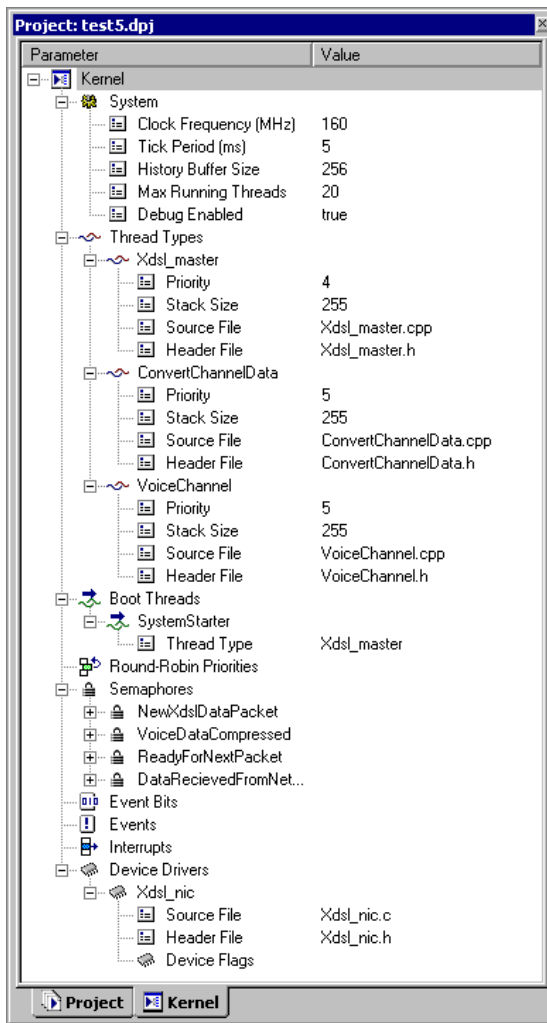


Figure 2-11. Expanded View of Kernel Elements

Project Window Right-Click Menus

From the **Project** window, you can access four different right-click menus that enable you to operate on **Project** window objects (the project, folders, or files). These menus are:

- **Project** window menu
- Project icon menu
- Folder icon menu
- File icon menu

Depending on the context (selected object), right-click menus provide an alternative means of performing an action. You can perform many of the actions from the menu bar commands or toolbar buttons.

Project Window Menu

The **Project** window's right-click menu enables you to:

- Hide the **Project** window from view
- Dock the **Project** window to the frame
- Float the **Project** window
- View project properties

Project Icon Menu

The Project icon  right-click menu is shown in [Figure 2-12](#).



Figure 2-12. Project Icon Right-Click Menu

This menu provides a *project* context, which enables you to:

- Build the project
- Clean (delete intermediate and target files)
- Add folders and files
- View and specify project options
- View project properties

Folder Icon Menu

The Folder icon   right-click menu is shown in [Figure 2-13](#).



Figure 2-13. Project Folder Right-Click Menu

The folder menu provides a *container* context from which to perform these local operations:

- Add or delete a folder
- Add files to the folder
- View folder properties

File Icon Menu

The File icon right-click menu is shown in [Figure 2-14](#).



Figure 2-14. File Icon Right-Click Menu

This menu provides a *file* context from which to perform these operations:

- Open the selected file for editing
- Build the file
- Remove the file from a project
- Specify options for the file
- View the file's properties

Editor Windows

Use editor windows to develop source code and edit project files.

Figure 2-15 shows items that you can customize in editor windows.

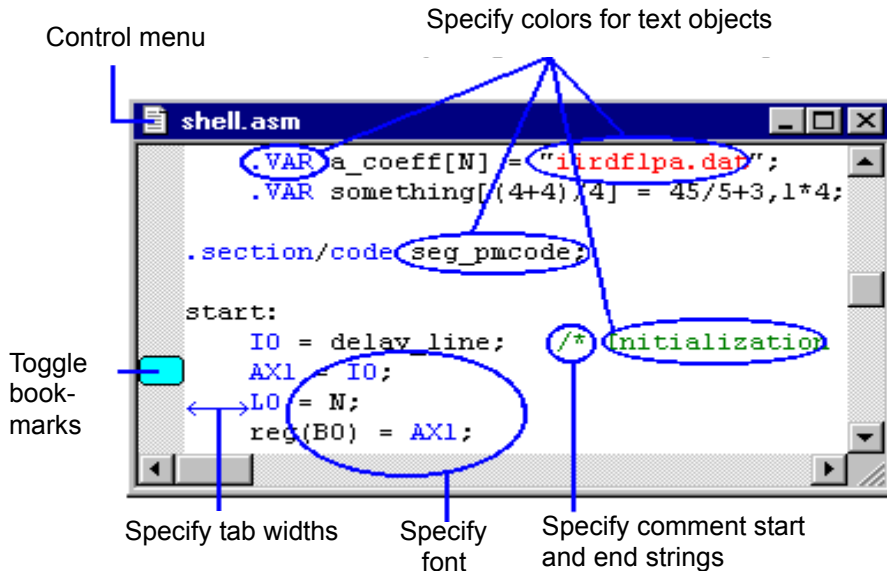


Figure 2-15. Items that can be Customized

You can open as many editor windows as you like, and you can perform the following functions.

- Define color-coded comments, strings, keywords, and tabs
- Preview and print window data
- Define headers and footers
- Set bookmarks
- Find and replace, wrap-around search, regular expression matching

- Go to a specified line number
- Jump to the next or previous syntax error
- Copy, cut, paste, undo and redo more than 500 levels of edits for each open file
- Enable the **Editor Tab** option to switch quickly between source files (see [“Editor Tab Mode” on page 2-30](#)).
- Locate matching brace characters and auto-position brace characters (to line up with the preceding opening brace)
- Open header files from the right-click menu. When you right-click on a #include statement, choose **Open Document** “filename.h” to open that file.
- Drag-and-drop highlighted sections of text (usually a valid source statement) to an open **Expressions** window. When dropped, the text is automatically added to the window and is evaluated.

Right-Click Menu

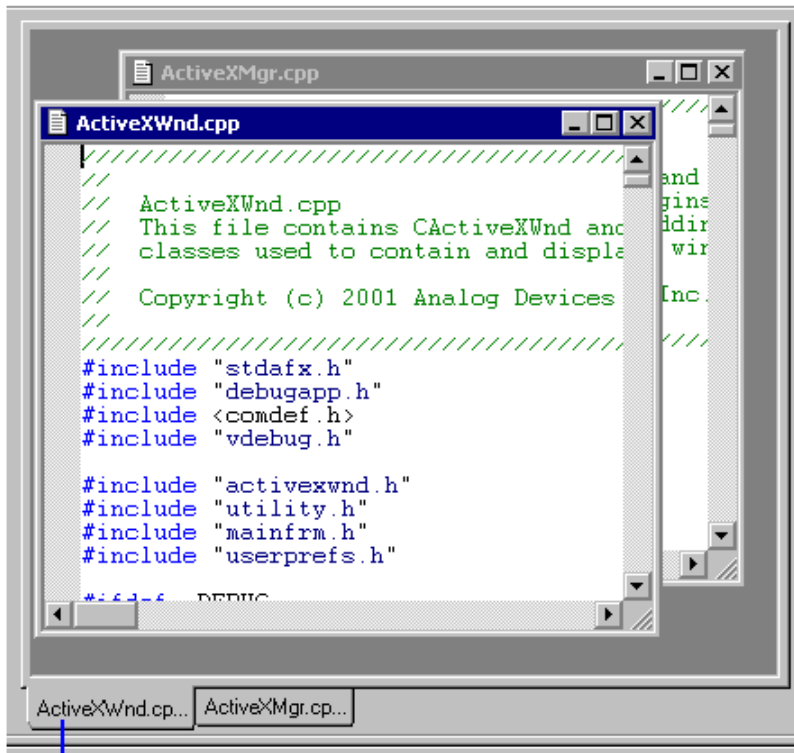
Use the editor window’s right-click menu to perform these functions:

- Undo or redo the last edit
- Cut, copy, or paste text
- Toggle a bookmark

Editor Tab Mode

Editor tab mode provides an alternative, tab-based user interface for managing multiple source files in editor windows. Choosing **Editor Tab** from the **View** menu displays a tab for each open source file at the bottom of the editor window. You can then click the tabs to switch between files quickly.

Figure 2-16 shows an editor window with the **Editor Tab** option enabled.



Click a tab to view the source file

Figure 2-16. Editor Tab Mode Enabled

Output Window

The **Output** window does the following.

- Displays standard I/O text messages such as file load status and error messages
- Displays build status information for the current project build
- Provides access to errors in source files
- Acts as an interface to the Tool Command Language (Tcl) used for scripting (see [“Issuing Commands from the Output Window” on page C-3](#))

The **Output** window shown in [Figure 2-17](#) contains build status information.

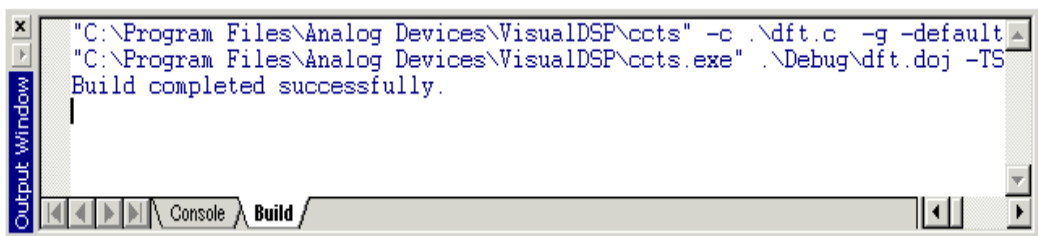


Figure 2-17. Build Status Information in the Output Window

Display the **Output** window by choosing **Output Window** from the **View** menu.

Output Window Tabs

Clicking the **Output** window's two tabs, **Console** and **Build**, displays views that provide different information and capabilities.

Build Tab

The **Build** tab (Figure 2-18) enables you to display error messages generated during a build. Double-click on an error message to jump to the offending code in an editor window.

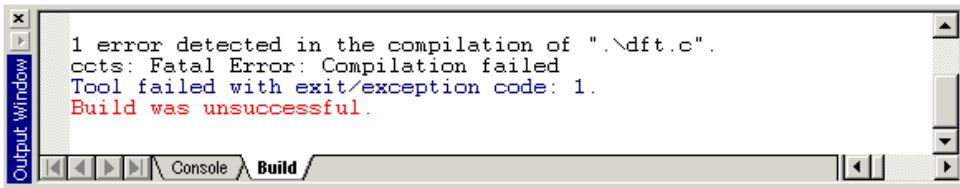


Figure 2-18. Error Messages in the Output Window

Scroll through error messages by choosing **Next Error** or **Prev Error** from the **Edit** menu.

By default, VisualDSP++ output is blue, and tool output is black, but you can change these colors in the **Preferences** dialog box.

Console Tab

The **Console** tab (Figure 2-20 on page 2-41), enables you to:

- View VisualDSP++ or target status error messages
- View STDIO output from C/C++ programs
- Scroll through previous commands by pressing the keyboard's up arrow (↑) and down arrow (↓) keys
- Perform multi-line selection, copy, paste, and clear
- Issue Tcl commands
- Auto-complete Tcl commands

- Execute a previously issued Tcl command by double-clicking on the command
- Enter multi-line Tcl commands by adding a backslash character (\) to the end of a statement
- Use bookmarks
- Toggle a bookmark by pressing **Ctrl+F2**
- Move to the next bookmark by pressing the keyboard's **F2** key

All text displayed in the **Console** view is also written to the VisualDSP++ log file.

Output Window Error Messages

The DSP code development tools that perform batch processing can produce error and warning messages when returning a result. These informational messages appear in the **Output** window's **Build** view.

Every error is identified with a unique six-character code, such as pp0019, that is consistent from release to release. Error descriptions include an explanation of the condition that caused the error and a suggested remedy to fix the problem. Where applicable, error messages include the source file's name and the line number of the offending code.

Error Message Severity Hierarchy

Each error message has one or more severity levels.

Table 2-10. Error Message Severity Levels

Severity Level	Description
Fatal error	Identify errors so severe that further processing of the input is suspended. Fatal errors are sometimes called catastrophic errors.
Error	Identify problems that cause the tool to report a failure. An error might allow further processing of the input to permit additional problems to be reported.
Warning	Identify situations that do not prevent the tool from processing the input, but may indicate potential problems
Remark	Provide information of possible interest

You can change the severity of a compiler message marked “-D” (discretionary). You cannot change the severity of compiler messages that are non-discretionary.

Syntax of Help for Error Messages

In Help, each error message can include several parts. The information that is displayed depends on the tool and the message.

Note: To view all the details, you must view the error message in the Help system window. If you run a tool from a command line interface (such as a **Command Prompt** window or **MS-DOS Prompt** window), the error message shows only the ID code, error text, and error location.

Table 2-11 describes the syntax for error message help.

Table 2-11. Syntax for Error Message Help

Part	Description
Identification code	Six-character code, unique to the error. The first two characters identify the tool: • pp (preprocessor) • cc (compiler) • li (linker) • ea (assembler) • ar (archiver) • ld (loader) • el (expert linker)
Error text	Text that appears after the identification code in the Output window
Description	Detailed description of the error
Severity	The degree of hardship imposed by the error. Some messages can take more than one severity level. You can change the severity level of an error marked “discretionary.” You cannot change errors marked “non-discretionary.”
Recovery	Extra information, provided only if applicable
Example	Example code
How to fix	The remedy for correcting the error
Related Information	Link(s) to more information

How to Promote, Demote, and Suppress Compiler Messages

A compiler message ID code with a “-D” (discretionary) suffix indicates that its severity can be overwritten. Refer to the tool documentation for command line switches that override message severity. The VisualDSP++ environment’s **Project Options** dialog box includes options that override severity.

You can promote, demote, or suppress a discretionary message. For example, you might promote a remark or warning to an error. You might decide to demote an error to a warning or remark.

If, for example, a condition in the input crashes the tool, you can restrict the problem to report as an error (instead of a fatal error).

Another way to suppress the reporting of an individual error message is by using pragmas in the input source via the tool’s command line. For more information about pragmas, refer to your *VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs*.

The examples below demonstrate how you can promote, demote, and suppress messages. The following source file (`test.c`) is being compiled.

```
#include <stdio.h>
int foo(void)
{
    printf("In foo\n"); // doesn't return a value
}

int main(void)
{
    int x; // no initial value
    printf("x = %d\n", x);
    return foo();
}
```

- Example 1: Compiling from the Command Line (Interface)

Compiling the `test.c` file yields these two warning messages:

```
$ cc21k -c test.c
"test.c", line 5: warning #1069-D: missing return statement
at end of non-void
function "foo"
}
^
"test.c", line 10: warning #549-D: variable "x" is used
before its value is set
printf("x = %d\n", x);
^
```

Note: The compiler appended “-D” (discretionary) to each of the two warning messages (#1069-D and #549-D).

- Example 2: Promoting Warnings to Errors

The following command line promotes the two warnings to errors.

```
$ cc21k -c test.c -Werror 549,1069
"test.c", line 5: error #1069-D: missing return statement at
end of non-void
function "foo"
}
^
"test.c", line 10: error #549-D: variable “x” is used before
its value is set
printf("x = %d\n", x);
^
```

2 errors detected in the compilation of “test.c”.
ccts: Fatal Error: Compilation failed

- Example 3: Demoting Messages to Remarks

You can demote messages to remarks. By default, however, the compiler does not display anything less significant than a warning.

The `-Wremarks` flag in the following command outputs the two warnings plus five other remarks.

```
$ cc21k -c test.c -Wremarks
```

The `-Wremark 549,1069` flag in the following command specifies that two specific messages be demoted to remarks. The command produces no output, because all the messages are changed to remarks, which are not displayed.

```
$ cc21k -c test.c -Wremark 549,1069
```

The following command changes the two warnings to remarks and then displays all seven remarks.

```
$ cc21k -c test.c -Wremark 549,1069 -Wremarks
```

- Example 4: Suppressing Messages

The following command suppresses two specific warning messages. The command outputs five remarks, but the two warnings are not displayed even though the `-Wremarks` flag requests all the remarks.

```
$ cc21k -c test.c -Wsuppress 549,1069 -Wremarks
```

- Example 5: Suppressing the Reporting of Warnings and Remarks

You can suppress remarks. You can also suppress both warnings and remarks.



You cannot suppress warnings without also suppressing remarks.

You control the output of warnings and remarks from the Project Options dialog box in VisualDSP++ or from the command line. Refer to your DSP's compiler, assembler, and linker manuals for available flags (options).

From the **Compile** page (Warnings category) of the **Project Options** dialog box, you can specify the options listed in [Table 2-12](#).

Table 2-12. Options Available from the Compile Page

Option	Purpose
Implicit function declarations	Warns on all implicit functions. This option corresponds to the compiler's -flags-compiler command line switch.
Functions not inlined	Issue a warning when the compiler is unable to generate inline code for a function that has the inline keyword
Enable remarks	Issues remarks, which are diagnostic messages of a milder nature than warnings. This option corresponds to the compiler's -Wremarks command line switch.
Disable all warnings and remarks	Withholds warning messages. This option corresponds to the compiler's -w command line switch.
Additional options	Enables you to enter more compiler options

How to View Error Message Details

Each DSP tool error message has associated explanatory text. You can view the information in the Help window by selecting the six-character error identifier (for example, cc0251) in the **Build** view and by pressing the F1 key. A complete explanation of the error message appears in the Help window.

Log File

The VisualDSP++ log file contains all status and error messages written to the **Output** window's **Console** view.

Figure 2-19 shows a sample log file.

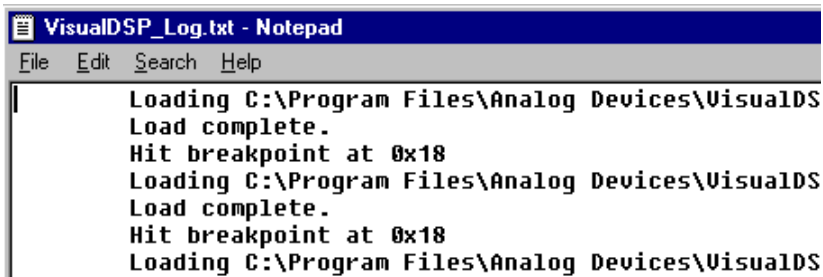


Figure 2-19. Example – Portion of a Log File

Note that:

- The file path specified in the log file assumes that you installed VisualDSP++ by accepting default settings.
- All sessions append to the log file. Occasionally, open the file and delete parts of it (or all of it) to conserve disk space.

Output Window Customization

You can specify preferences that:

- Configure **Output** window fonts and colors
- Enable command auto-completion

By default, the **Output** window resides at the bottom of the main application window. You can resize or move the **Output** window, which is a Windows docking bar, to a different portion of the screen by dragging it to the selected location.

The **Output** window's **Console** view can interact with the VisualDSP++ Tcl engine. All Tcl input and output is sent to the **Console** view, shown in [Figure 2-20](#).

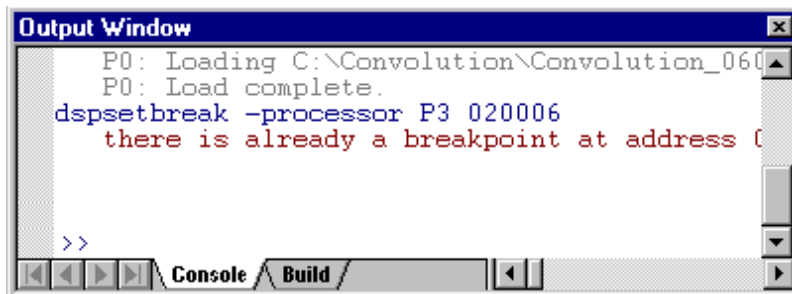


Figure 2-20. Messages in the Output Window's Console View

These messages are saved to the log file `VisualDSP_Log.txt`, which is located in the `C:\Program Files\Analog Devices\VisualDSP\Data` directory.

Right-Click Menu

The **Output** window's right-click menu is shown in [Figure 2-21](#).

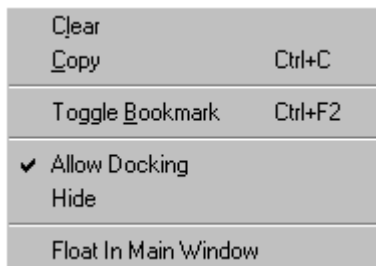


Figure 2-21. Output Window's Right-Click Menu

This menu enables you to:

- Dock the window
- Float the window
- Hide the window (display the hidden window by choosing **Output Window** from the **View** menu)
- Copy selected text
- Clear the text in the windows
- Toggle bookmarks

Window Operations

Similar to many Windows applications, VisualDSP++ provides ways to adjust your view of the user interface.

Window Manipulation

The **Window** menu commands, shown in [Figure 2-22](#), enable you to manipulate your windows display and update windows during program execution. Refer to your Windows documentation for more information.

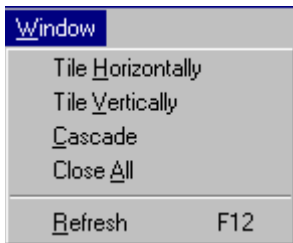


Figure 2-22. Window Menu Commands

Right-Click Menu Options

Each window presents a menu when you right-click in the window or on its title bar. The menu options in [Table 2-13](#) affect window behavior.

Table 2-13. Window Right-Click Menu Commands

Option	Description
Allow Docking	Enables/disables docking
Close	Closes the window
Float in Main Window	Causes the window to become a normal MDI child window (like an editor window) and disables its docking ability

Scroll Bars and Resize Pull-Tab

Scroll bars appear along the right and bottom edges of the application or document window, as shown in [Figure 2-23](#).

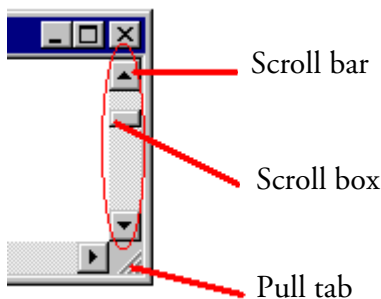


Figure 2-23. Scrolling to Move the Viewing Area

The scroll boxes inside the scroll bars indicate your vertical and horizontal location in the document. Use the mouse to scroll to other parts of the document.

When the application window is not maximized, the resize pull-tab appears in the lower-right corner of the window. Click and drag the pull-tab to resize the application window.

Windows: Docked vs. Floating

A window attached to the application's frame is referred to as a *docked window*.

You can detach a window from the main window and move it to another location anywhere on the desktop. A *floating window* stands alone, because it is not docked.

Depending on your needs, you can:

- Dock a window to the application's main window (frame)
- Float a window

A window's right-click menu provides commands to dock or float the window. The **Allow Docking** option and the **Float In Main Window** option are mutually exclusive.

Example of a Docked Window

The **Project** window shown in [Figure 2-24](#) is docked (**Allow Docking** is selected).

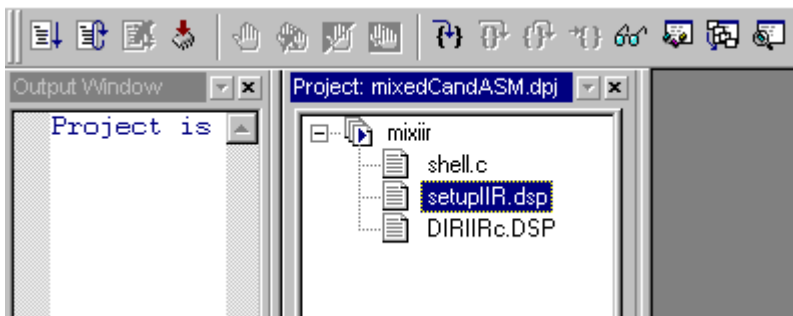


Figure 2-24. Example of a Docked Project Window

To prevent a window from docking, hold down the keyboard's **Ctrl** key while dragging the window to another position.

Window Operations

Examples of Floating Windows

The Project window in [Figure 2-25](#) is floating in the main window (**Float In Main Window** is selected).

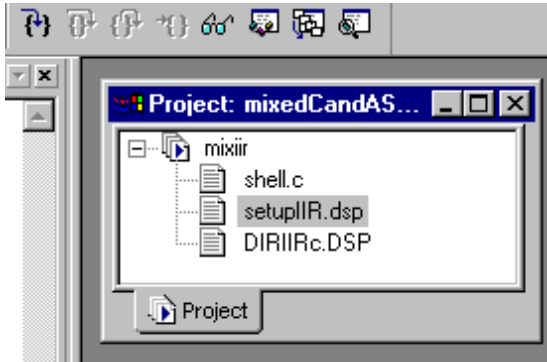


Figure 2-25. Project Window Floating in Main Window (1 of 2)

The Project window in [Figure 2-26](#) is also floating in the main window (**Float In Main Window** is selected).

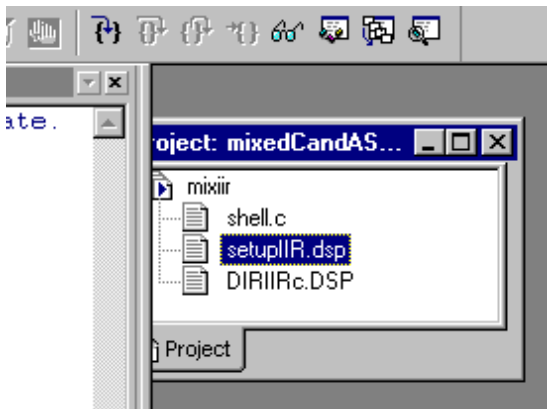


Figure 2-26. Project Window Floating in Main Window (2 of 2)

The Project window in [Figure 2-27](#) is floating, but not in the main window (**Float In Main Window** is not selected).

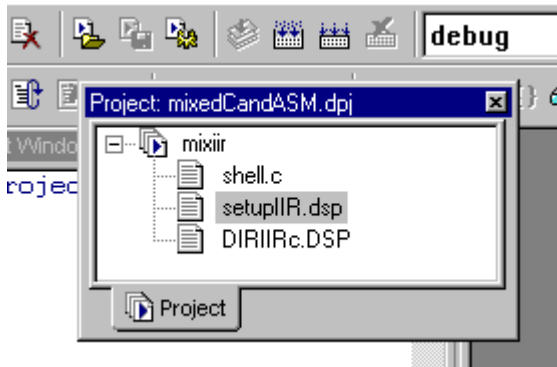


Figure 2-27. Project Window Floating but Not in Main Window

Window Position Rules

The following rules apply to window positions.

- Unless **Allow Docking** is disabled, a window must reside within the main window.
- An editor window cannot be docked to the main window.
- A window specified as an MDI child cannot be positioned over a docked window.
- Unless the **Output** window is floating in the main window, you cannot position a window specified as an MDI child over the **Output** window.

Standard Windows Buttons

The standard Windows buttons are located on the right side of the title bar, as shown in [Figure 2-28](#).



Figure 2-28. Title Bar Showing Standard Window Buttons

These buttons resize and close the window as described in [Table 2-14](#).

Table 2-14. Standard Windows Buttons

Button	Name — Purpose
	Minimize – reduces the window to its Windows icon
	Maximize – enlarges the window to fill the screen
	Restore – returns the window to its last non-minimized, non-maximized position after you maximize the window
	Close – closes the application window and exits the program

Debugging Windows

VisualDSP++ provides debugging windows to display DSP program operation and results. [Table 2-15](#) describes these windows.

Table 2-15. Debugging Windows

Window	Provides
Output	A Console view that displays standard I/O text messages such as file load status, error messages, and a Build view that displays build messages. You can interactively enter Tcl commands and view Tcl output.
Editor	Syntax coloring, context-sensitive expression evaluation, and status icons that indicate breakpoints, bookmarks, and the current PC position
Disassembly	Code in disassembled format. This window provides fill and dump capability.
Expressions	The means to enter an expression and see its value as you step through program execution
Trace	A history of processor activity during program execution, including buffer depth (instruction lines), cycle count, and instructions executed such as memory fetches, program memory writes, and data/memory transfers
Locals	All local variables within a function. Use this window with step or halt commands to display variables as you move through your program.
Linear Profiling Results	(Simulation only) Samples of the target's PC register taken at every instruction cycle, which provides an accurate picture of where instructions were executed. Linear profiling is much slower than statistical profiling.
Statistical Profiling Results	(JTAG emulation only) Random samples of the target processor's program counter (PC) and a graphical display of the resulting samples, showing where the application spends time
Call Stack	A means of moving the call stack back to the previous debug context

Debugging Windows

Table 2-15. Debugging Windows (Cont'd)

Window	Provides
Register	Current values of registers. You can change register contents and change the number format.
Memory	A view of DSP memory. Similar number format and edit features as register windows, plus fill and dump capability.
Memory Map	The memory map of the selected processor
Plot	A graphical display of values from memory addresses. The window supports linear and FFT (real and complex) visualization modes and allows you to export an image to a file, the clipboard, or to a printer.
Multiprocessor	Current status of each processor in a multiprocessor system (emulator sessions only). This window allows you to define and manage groups of processors for synchronous multiprocessor commands.
State History	(VDK-enabled projects only) History buffer of threads and events
Target Load	(VDK-enabled projects only) Percent of time the target spent in the idle thread
VDK Status	(VDK-enabled projects only) At a program halt, thread state and status data

Disassembly Windows

By default, a **Disassembly** window appears when you open a new session.

[Figure 2-29](#) and [Figure 2-30](#) show examples of **Disassembly** windows.

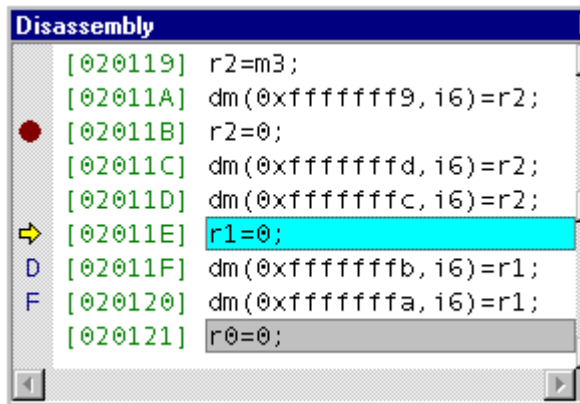


Figure 2-29. Disassembly Window (Example 1)

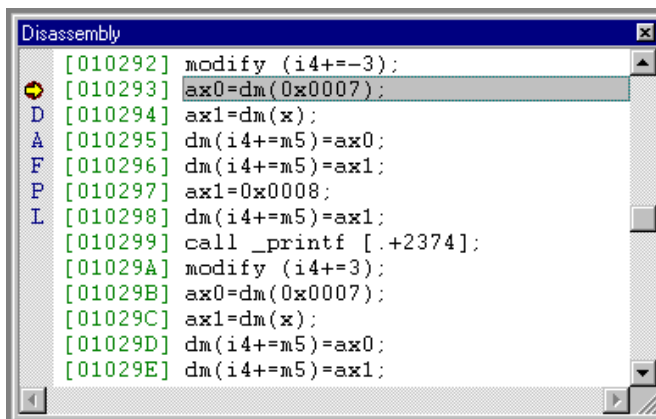


Figure 2-30. Disassembly Window (Example 2)

Debugging Windows

Disassembly windows display code in disassembled form, which is useful for temporarily modifying the code to test a change or to view code when no source is available. The **Disassembly** window allows you to examine the assembly code generated by the C/C++ compiler.

To make changes permanent, modify the code, and rebuild the project.

Disassembly windows provide:

- Number format and edit features, similar to register windows
- Dump and fill capability

By default, the current source line to be executed is highlighted by a light-blue horizontal bar, as shown in the following example.

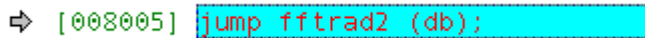
A screenshot of a disassembly window. On the left, there is a green arrow pointing to the right, followed by the address [008005] in green text. To the right of the address, the instruction 'jump fftrad2 (db);' is displayed in blue text. The entire line, including the address and instruction, is highlighted with a light blue horizontal bar.

Figure 2-31. Current Source Line in the Disassembly Window

You can configure the color of the current source line and other window items.

Other Disassembly Window Features

From the **Disassembly** window, you can perform the operations described in [Table 2-16](#).

Table 2-16. Disassembly Window Operations

To...	Place the mouse pointer over...
Move to a different address	An address field and double-click. Then select the address from the ensuing Go To dialog box.
Insert or remove a breakpoint	An instruction and double-click
Toggle (enable or disable) a breakpoint	An instruction and right-click. Then choose the appropriate command from the ensuing menu.

Right-Click Menu

The **Disassembly** window's right-click menu provides access to the commands shown in [Figure 2-32](#).

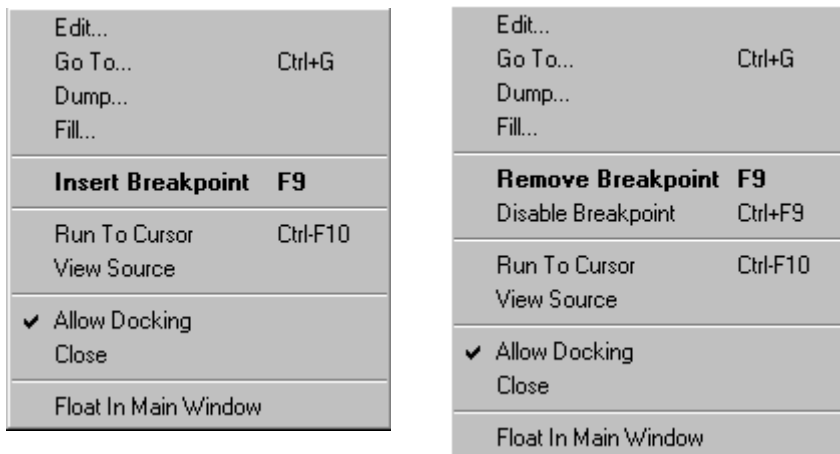


Figure 2-32. Disassembly Window Right-Click Menus

Disassembly Window Symbols

The **Disassembly** window denotes program execution stages with symbols at the far left of the **Disassembly** window. The display of pipeline stages is available only when your system is connected to a simulator target.

[Table 2-17](#) shows the symbols displayed at the left of the **Disassembly** window.

Table 2-17. Disassembly Window Symbols

Symbol	Description
	Current source line
	The current instruction is being aborted due to a branch or jump instruction
	A breakpoint is enabled
	A breakpoint is disabled
F	This instruction is currently in the Fetch Address stage of the pipeline.
D	This instruction is currently in the Instruction Decode stage of the pipeline.
E	This instruction is currently in the Execute stage of the pipeline.

The display of pipeline stages is available only when:

- The selected session is connected to a simulator target.
- **Enable pipeline display** is selected on the **General** tab page of the **Preferences** dialog box, available from the **Settings** menu.

Expressions Window

The **Expressions** window, shown in [Figure 2-33](#), enables you enter an expression to evaluate in your program. Expression evaluations are based on the current debug context.

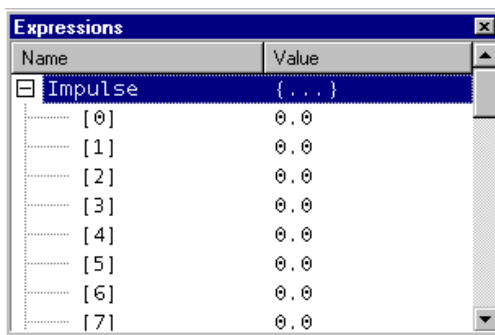


Figure 2-33. Expressions Window

Because of the way registers are saved and restored on the stack, the register value on which the expression relies may be incorrect if you change VisualDSP++'s context with the **Call Stack** window.

The **Expressions** window's right-click menu ([Figure 2-34](#)) includes commands that let you change the display's number format.

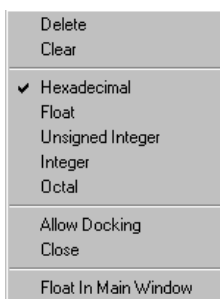


Figure 2-34. Expressions Window Right-Click Menu

Trace Windows

You perform a trace (also called an execution trace or a program trace) to analyze the run-time behavior of your DSP program, to enable I/O capabilities, and to simulate source-to-target data streaming. [Figure 2-35](#) shows data displayed in a **Trace** window.

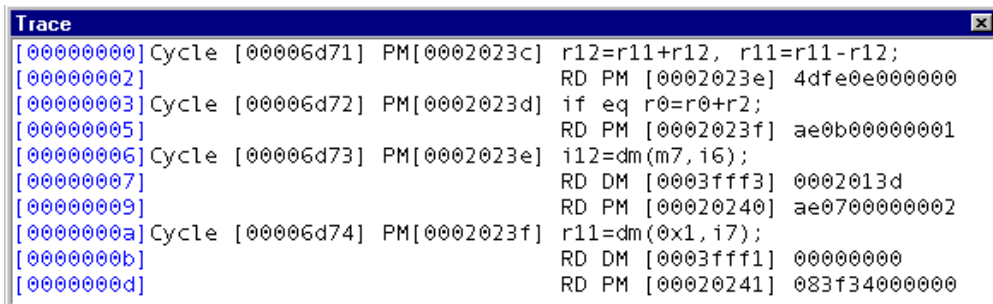


Figure 2-35. Example of Data in a Trace Window

The Trace window displays:

- Buffer depth (**Custom** in the **Trace Buffer Depth** dialog box)
- The clock cycle when the instruction occurred
- The address of the instruction that was executed
- The disassembled instruction

Memory results have the following fields.

- Access type (RD or WR)
- Memory type (PM or DM)
- The address, in brackets ([])
- The data value that was written or read

Locals Window

The **Locals** window displays the value of local variables within a function, as shown in [Figure 2-36](#).

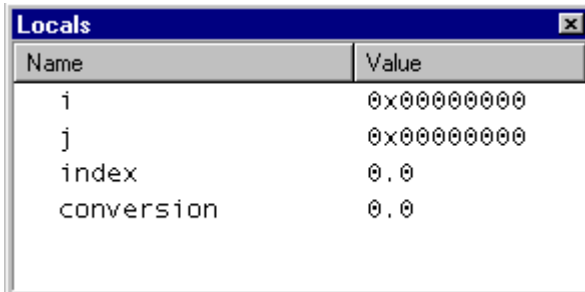


Figure 2-36. Locals Window

Use this window with a **Step** or **Halt** command to display the current value of variables as you move through your program.

Complex variables, C structures, and C++ classes appear with a plus  sign. Click on the plus sign to display all variable information.

The window's right-click menu provides the commands shown in [Figure 2-37](#).

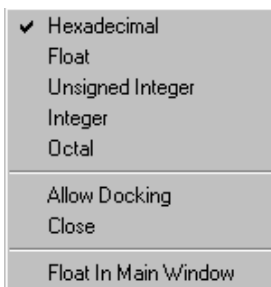


Figure 2-37. Locals Window Right-Click Menu

Statistical/Linear Profiling Results Window

Depending on the target, the window's title is **Statistical Profiling Results** or **Linear Profiling Results**. The window comprises two panes, as shown in [Figure 2-38](#).

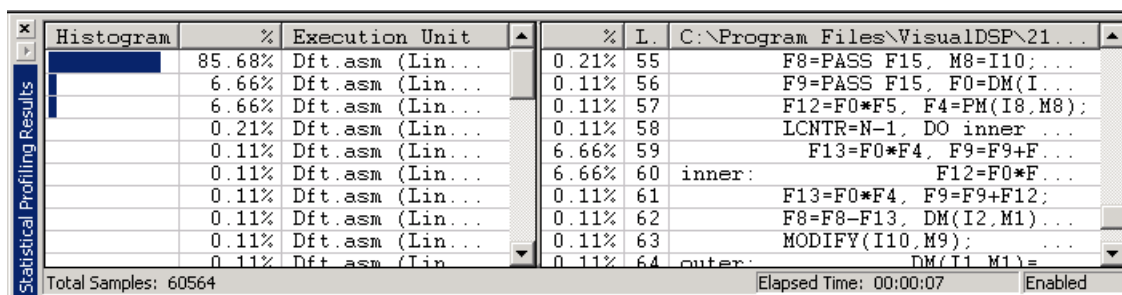


Figure 2-38. Example of a Linear Profiling Results Window

Window Components

The window, which comprises two panes and a status bar, provides a right-click menu from which you can perform various window functions.

Left Pane

The window's left pane displays a list of the executed functions, assembly source lines, and PCs (with no debug information). The time that each item spent on execution appears as a histogram and as a percent. The order of the items in the display is determined by the percentage of global execution time that each item took to execute.

The left pane includes the information described in [Table 2-18 on page 2-59](#).

Table 2-18. Left Pane Information

Column	Displays	Purpose
Histogram	Horizontal bars	Graphically represents the execution percentage
% -or- Count	A percent with two decimal places, for example: 15.01% -or- a number	Displays execution in percent or as a count. Right-click and choose View Execution Percent to view execution as a percent, or choose View Sample Count to view the PC sample count.
Execution Unit	Functions, assembly source lines, and PCs for which no debug information exists	These items are sorted by the percentage of global execution time that each item took to execute. The highest percentage items appear at the top of the list

If you double-click on a line with a function or assembly source line in the left pane, the right pane displays the corresponding source file and jumps to the top of that function or assembly source line, respectively. If you double-click on a PC address with no debug information, the **Disassembly** window opens to that address.

Debugging Windows

Right Pane

The right pane includes the information described in [Table 2-19](#).

Table 2-19. Information in the Right Pane

Column	Displays
%	Execution percent in text format with two decimal places (for example, 1.03%) -or- the PC sample count for each source line
Line	Line numbers of the source file
File	Entire source file. Each source line occupies one line in the grid control.

Status Bar

The *status bar* at the bottom of the window indicates the total number of collected PC samples, the total elapsed time, and whether statistical profiling is enabled.

Right-Click Menu

The **Statistical Profiling Results** and **Linear Profiling Results** windows provide a right-click menu. The menu commands depend on the context (whether you right-click in the left pane or right pane) and the current settings.

[Table 2-20 on page 2-61](#) describes the menu commands.

Table 2-20. Profiling Results Window Right-Click Menu Commands

Command	Description
Enable	Enables or disables profiling
Load Profile	Opens the Select a Statistical /Linear Profile to Load dialog box from which you can load profile data saved from a previous run
Save Profile	Saves the current run's data to a file
Concatenate Profile	Merges profiling data stored from a previous run with current data
Clear Profile	Clears statistics saved from a previous run
View Execution Percent	Displays the execution percent in each execution unit or source line. This value is the sample count for that execution unit divided by the total number of samples.
View Sample Count	Displays the sample count for that execution unit
Mixed -or- Source	Sets the display mode for C/C++ source lines from the right pane only. Choose Mixed to display both C/C++ source lines and assembly lines. C/C++ source lines appear in black type, and assembly lines appear in gray. Profiling data appears for each assembly line. Choose Source to display only the C/C++ source lines.
Properties	Opens the Profile Window Properties dialog box, from which you can view or change window settings. When you perform linear profiling, you can select display options such as cache hits, cache misses, execution count, reads, and writes.

Window Operations

You can select various options for the **Statistical/Linear Profiling Results** window and perform various window operations.

Changing the Window View

After you specify properties for the **Statistical/Linear Profiling Results** window and enable profiling, the profiler collects data when you run a program. Depending on the filtering options that you select, the window's **Execution Unit** column displays:

- Function names (such as `main`)
- Single addresses, for example, `PC(0x20000)`
- Address ranges, for example, `[20020-20030]`

 Single addresses and address ranges are in hexadecimal format. The “0x” notation, however, appears beside single addresses only.

Displaying a Source File

Double-clicking on a function name in the **Execution Unit** column not only displays the source of the function in the right pane but also the profiling data for each line of the function. [Figure 2-39](#) shows code for the `fir` function.

%	L...	C:\Program Files\VisualDSP\21k\examples\ASM_Examples\DFT...	
	45	/* DFT Subroutine	
0.00%	46	dft: B8=sine; /* Sine pointer */	
0.00%	47	L8=@sine;	
0.00%	48	B9=sine; /* Derive cosine from sin...	
0.00%	49	I9=sine+N/4; /* shifting pointer over ...	
0.00%	50	L9=@sine; /* and using a circular b...	
0.00%	51	I10=0; /* I10 is used to increme...	
0.00%	52	L10=0; /* frequency of sine look...	
0.00%	53	F15=0; /* Zero to clear accumula...	
0.00%	54	LCNTR=N, DO outer UNTIL LCE;	
0.21%	55	F8=PASS F15, M8=I10; /* Update frequen...	
0.11%	56	F9=PASS F15, F0=DM(I0,M1), F5=PM(I9,M8);	
0.11%	57	F12=F0*F5, F4=PM(I8,M8);	
0.11%	58	LCNTR=N-1, DO inner UNTIL LCE;	
6.66%	59	F13=F0*F4, F9=F9+F12, F0=DM(I0,M1), F5=PM(I9,M8);	
6.66%	60	inner: F12=F0*F5, F8=F8-F13, F4=PM(I8,M8);	
0.11%	61	F13=F0*F4, F9=F9+F12;	
0.11%	62	F8=F8-F13, DM(I2,M1)=F9; /* Write real res...	
0.11%	63	MODIFY(I10,M9); /* Increment freq...	

Figure 2-39. Code Displayed for a Function

Working with Ranges

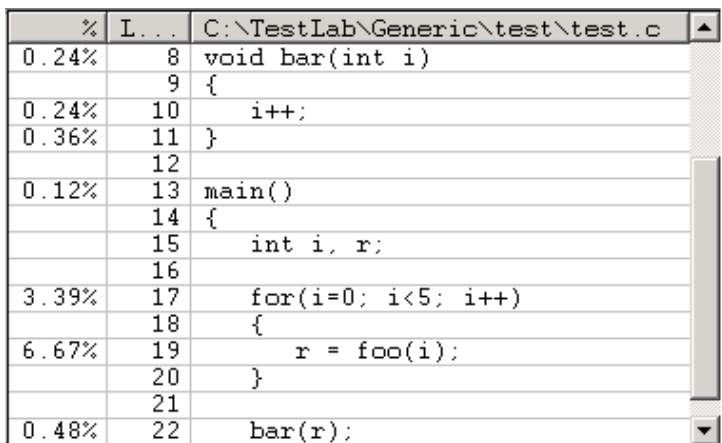
Clicking on the icon in an address range expands or contracts the list of functions within that address range.

When expanded, the list of functions appears and profiling data appear immediately after the address range.

Debugging Windows

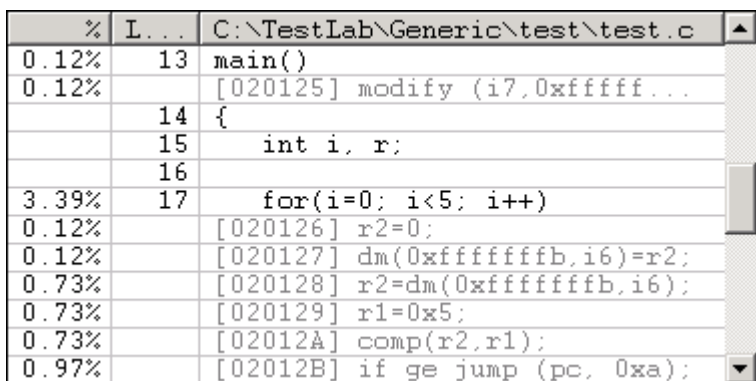
Switching Display Modes

The right-click menu's **Mixed** and **Source** commands simplify switching between two views. [Figure 2-40](#) shows the source mode view and [Figure 2-41](#) shows the mixed mode view.



%	L...	C:\TestLab\Generic\test\test.c
0.24%	8	void bar(int i)
	9	{
0.24%	10	i++;
0.36%	11	}
	12	
0.12%	13	main()
	14	{
	15	int i, r;
	16	
3.39%	17	for(i=0; i<5; i++)
	18	{
6.67%	19	r = foo(i);
	20	}
	21	
0.48%	22	bar(r);

Figure 2-40. Source Mode View



%	L...	C:\TestLab\Generic\test\test.c
0.12%	13	main()
0.12%		[020125] modify (i7,0xfffff...
	14	{
	15	int i, r;
	16	
3.39%	17	for(i=0; i<5; i++)
0.12%		[020126] r2=0;
0.12%		[020127] dm(0xffffffffb,i6)=r2;
0.73%		[020128] r2=dm(0xffffffffb,i6);
0.73%		[020129] r1=0x5;
0.73%		[02012A] comp(r2,r1);
0.97%		[02012B] if ge jump (pc, 0xa);

Figure 2-41. Mixed Mode View

When you view the window in mixed mode, profiling data for each assembly line is displayed, as shown in [Figure 2-42](#). Mixed mode displays profiling statistics for individual assembly instructions.

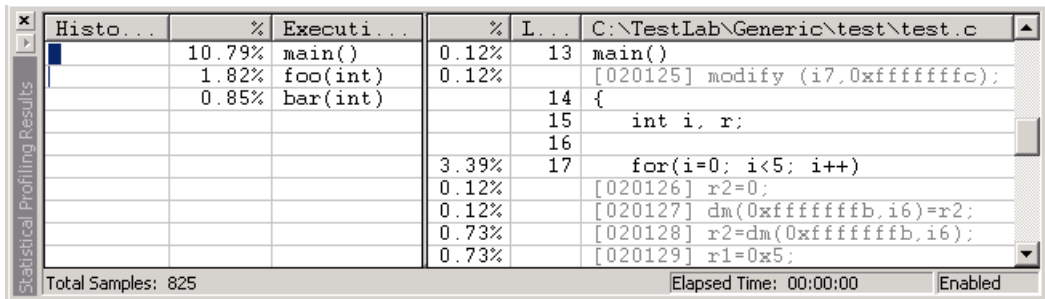
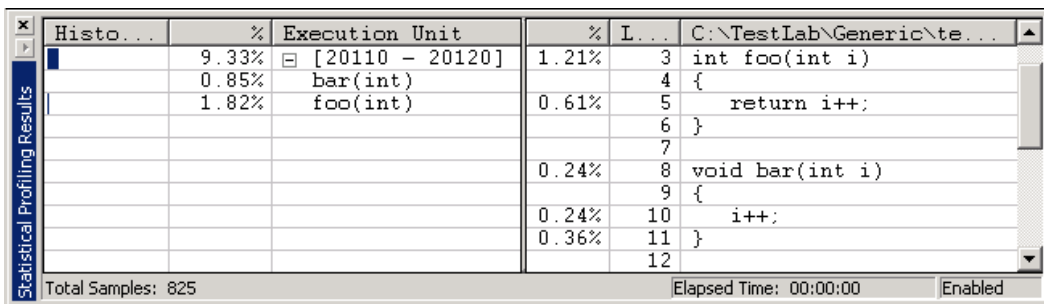


Figure 2-42. Profiling Data for Each Assembly Line (Mixed Mode)

Debugging Windows

Filtering PC Samples with No Debug Information

Since you spend most of your time building a “debug version” of your code, eliminate non-debug code, such as C run-time library initialization code. [Figure 2-43](#) shows where a lot of time is spent before filtering.

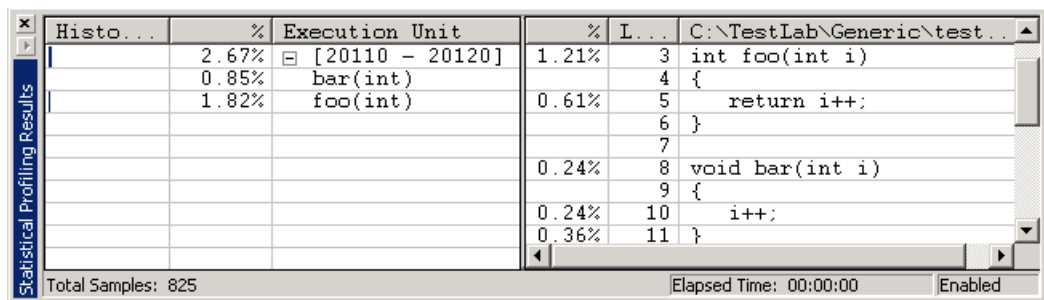


Histo...	%	Execution Unit	%	L...	C:\TestLab\Generic\te...
	9.33%	[20110 - 20120]	1.21%	3	int foo(int i)
	0.85%	bar(int)		4	{
	1.82%	foo(int)	0.61%	5	return i++;
				6	}
				7	
			0.24%	8	void bar(int i)
				9	{
			0.24%	10	i++;
			0.36%	11	}
				12	

Total Samples: 825 Elapsed Time: 00:00:00 Enabled

Figure 2-43. Profiling Results Before Filtering

The profiling results after filtering ([Figure 2-44](#)) reflect the difference.



Histo...	%	Execution Unit	%	L...	C:\TestLab\Generic\test...
	2.67%	[20110 - 20120]	1.21%	3	int foo(int i)
	0.85%	bar(int)		4	{
	1.82%	foo(int)	0.61%	5	return i++;
				6	}
				7	
			0.24%	8	void bar(int i)
				9	{
			0.24%	10	i++;
			0.36%	11	}

Total Samples: 825 Elapsed Time: 00:00:00 Enabled

Figure 2-44. Profiling Results After Filtering

Call Stack Window

The **Call Stack** window (Figure 2-45) enables you to double-click on a stack location to move the call stack back to a previous debug context.

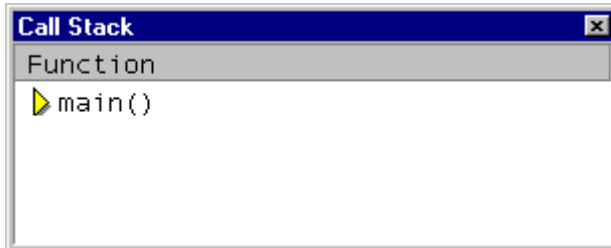


Figure 2-45. Example of the Call Stack Window



This window functions with C/C++ code only.

Use this window to analyze the state of parent functions when erroneous data is being passed to the currently executing function and to see the context from which the current function is being called. You can walk up the call stack and view local variables in different scopes.

Memory Windows

From a memory window, you can:

- View and edit memory contents
- Display the address of a value. Move the mouse pointer over the value, and hold down the keyboard's **Ctrl** key.
- Lock the number of columns currently displayed. This action resizes the window horizontally without altering the display
- Track one expression

Debugging Windows

Memory windows, similar to register windows, provide:

- Number format and edit features
- Fill and dump capability

Memory Number Formats

The memory windows that follow show examples of different memory number formats.

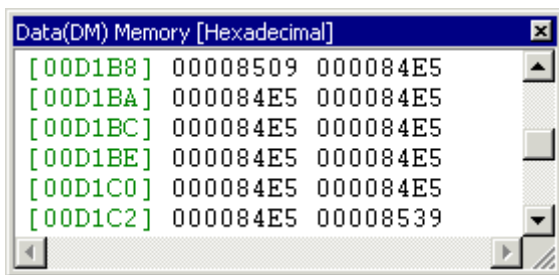


Figure 2-46. SHARC Memory Window in Hexadecimal Format

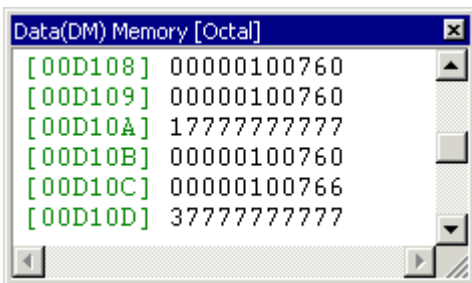


Figure 2-47. SHARC Memory Window in Octal Format

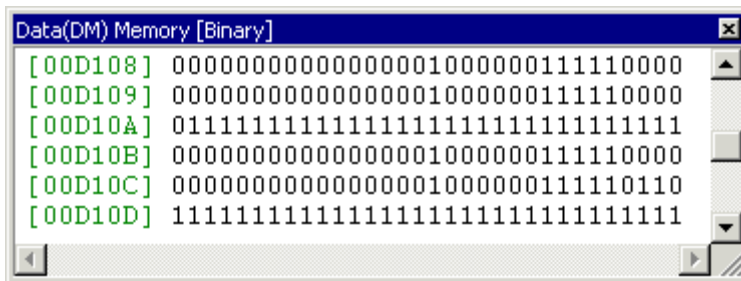


Figure 2-48. SHARC Memory Window in Binary Format

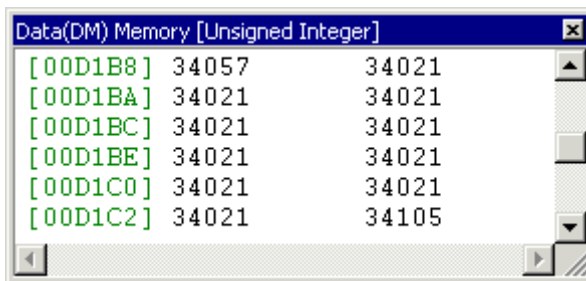


Figure 2-49. SHARC Memory Window in Unsigned Integer Format

Debugging Windows

Right-Click Menu

Memory windows provide the right-click menu shown in [Figure 2-50](#).

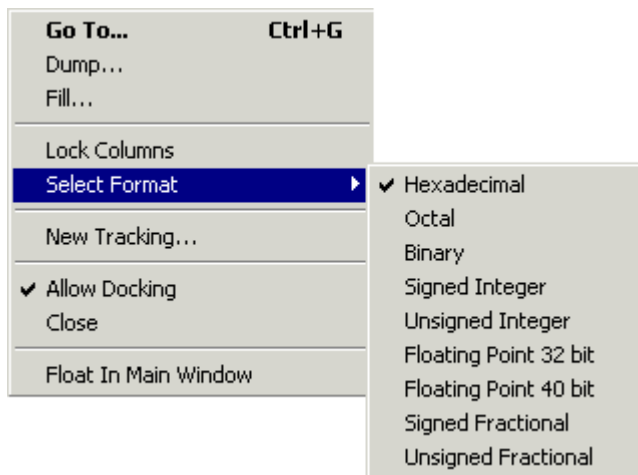


Figure 2-50. Memory Window Right-Click Menu

The **Select Format** command enables you to change the number format of the display.

Expression Tracking in a Memory Window

While you step through your code, a memory window configured for expression tracking shows the memory at the address specified by the expression (see [Figure 2-51 on page 2-71](#)).

Title bar displays the
tracking expression

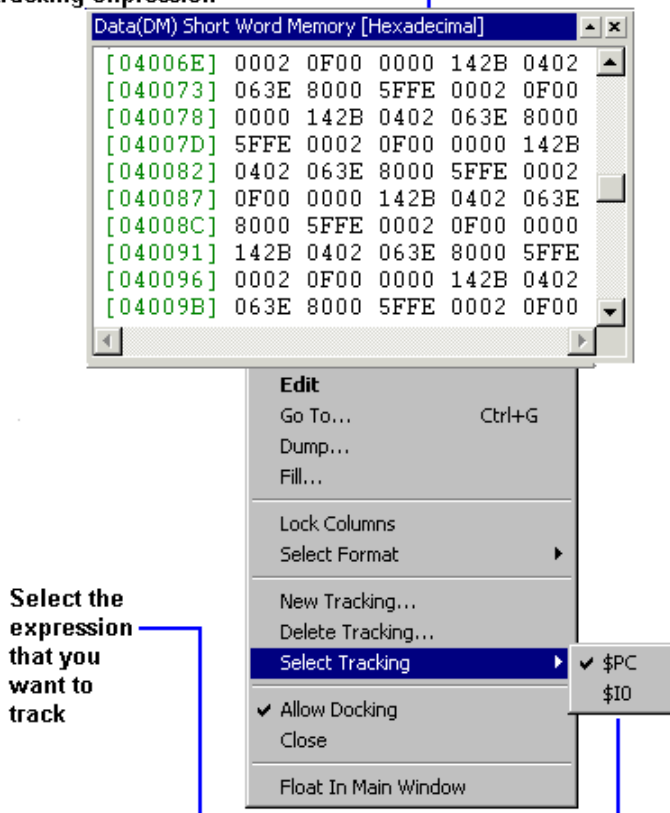


Figure 2-51. Expression Tracking in a Memory Window

Debugging Windows

Every time the target halts, the tracking expression is evaluated and the memory window jumps to that address. For example, if “\$PC” is used as the tracking expression, the memory window behaves like the **Disassembly** window.

Note:

- In a memory window, you can configure several expressions for tracking.
- You can track only one expression at a time in a memory window.
- The active expression appears in the memory window’s title bar.
- The memory window’s right-click menu displays a list of configured expressions, and you can select one of them for tracking.
- To track multiple expressions, open multiple memory windows and track one expression per window.

Memory Window Display Customization

You can specify the colors used for symbols, data, address values modified values, and undefined memory regions. You can also adjust the width of the window to display a particular number of data columns. For example, the memory window in [Figure 2-52](#) is sized to display five columns.

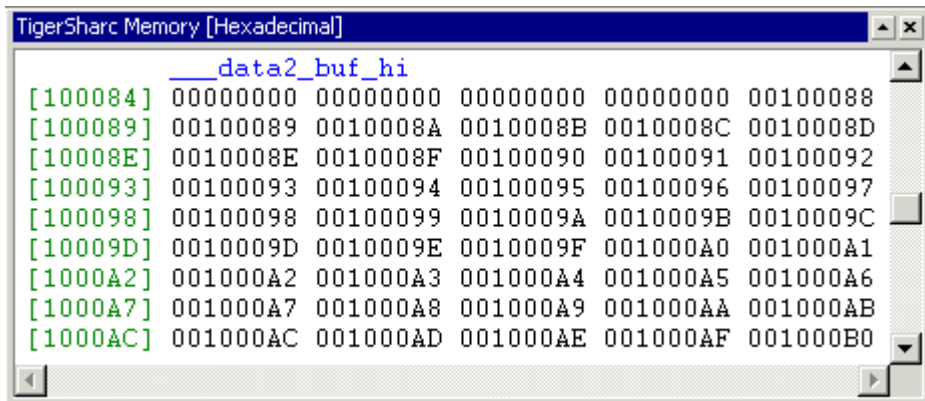


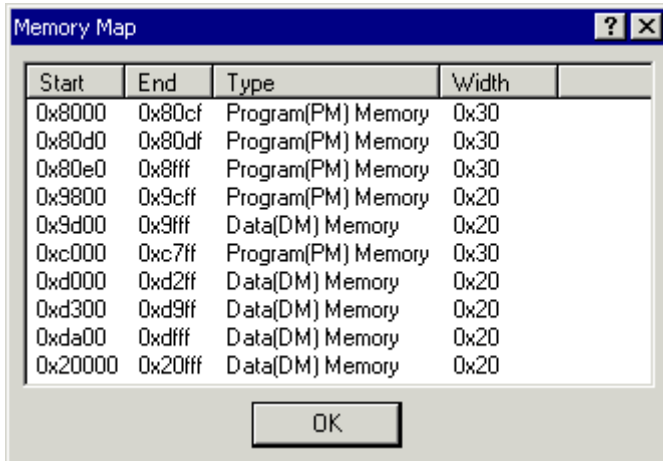
Figure 2-52. Memory Window (Five-Column Display)



The commands displayed from the menu bar's **Memory** menu and the memory window's right-click menu depend on the processor that you are debugging.

Memory Map Window

The **Memory Map** window displays the memory map for the selected processor. [Figure 2-53](#) shows a memory map for a SHARC DSP.



The screenshot shows a window titled "Memory Map" with a standard Windows-style title bar (blue background, white text, and buttons for help, maximize, and close). The window contains a table with four columns: "Start", "End", "Type", and "Width". The table lists several memory segments for a SHARC DSP. At the bottom of the window is an "OK" button.

Start	End	Type	Width
0x8000	0x80cf	Program(PM) Memory	0x30
0x80d0	0x80df	Program(PM) Memory	0x30
0x80e0	0x8fff	Program(PM) Memory	0x30
0x9800	0x9cff	Program(PM) Memory	0x20
0x9d00	0x9fff	Data(DM) Memory	0x20
0xc000	0xc7ff	Program(PM) Memory	0x30
0xd000	0xd2ff	Data(DM) Memory	0x20
0xd300	0xd9ff	Data(DM) Memory	0x20
0xda00	0xdfff	Data(DM) Memory	0x20
0x20000	0x20fff	Data(DM) Memory	0x20

Figure 2-53. Memory Map Window

If no DSP program is loaded into the processor, the memory map displays all available memory in the processor.

If a program is loaded, the memory map is the map defined in the `memory` section of the program's `.LDF` file.

For each portion of memory, the window displays the start address, end address, and width.

Register Windows

VisualDSP++ provides access to various register windows. In a register window you can view and change register contents and change the presentation (number format).

Figure 2-54 shows the **Register** menu tree. The actual menus may differ because of last minute changes to the VisualDSP++ software.

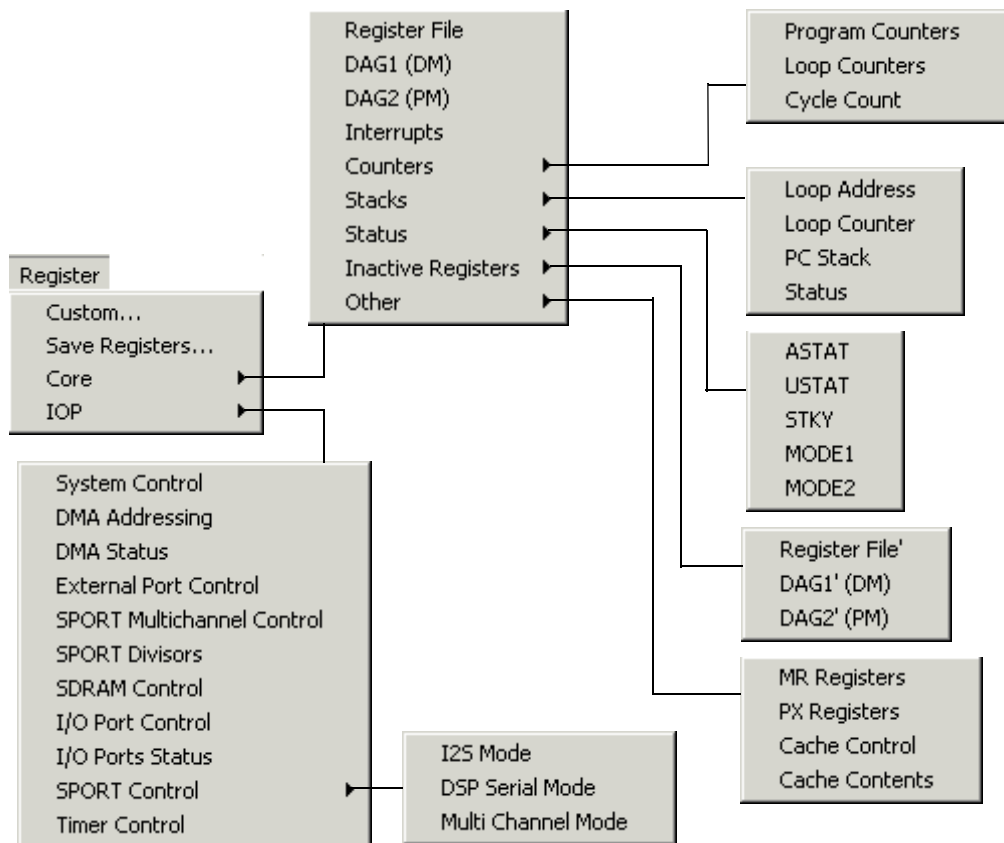


Figure 2-54. Register Windows Available for SHARC DSPs

Debugging Windows

Figure 2-55 shows an example of an active register file in a register window.

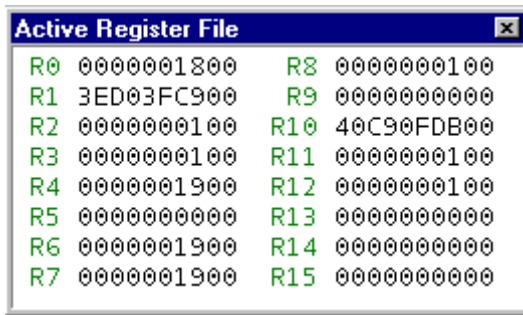


Figure 2-55. Example of a Register Window

Register windows support various number formats that include standard formats such as hexadecimal, octal, and binary.

You can change a register's data directly from a register window. The modified register content is used during program execution. Edits to data do not affect your source files. To make changes permanent, edit the source file and rebuild your project.

Stack Windows

Depending on your processor, you have access to various stack windows, such as the **Loop Address** stack. For more information about your processor's stack windows, consult online Help.

Custom Register Windows

While debugging, you can configure and display custom register windows. Each custom register window has a user-specified title and displays only the registers you choose to monitor.

For example, the following custom register window displays the contents of five registers.

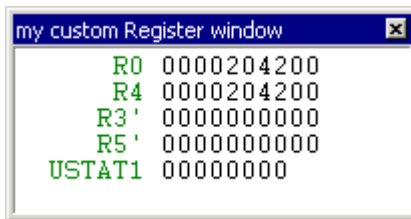


Figure 2-56. Example of a Custom Register Window

A custom register window appears immediately after you create it.

Multiprocessor Window

Use the **Multiprocessor** window (Figure 2-57) to select and control the different processors in a multiprocessor debug session.

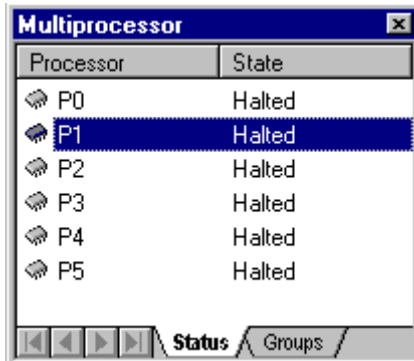


Figure 2-57. Multiprocessor Window

The window consists of two views, **Status** and **Groups**.

Multiprocessor Groups

If a session contains three processors (A, B, and C) and a group is created that contains A and C, the **MP Run** command runs A and C only, and B remains unaffected.

Focus

Processor focus changes, depending on the currently selected window. To move focus among the processors, click on a processor listed in the **Multiprocessor** window ([Figure 2-57 on page 2-78](#)).

You can pin any debug window to a specific processor. Select the processor in the **Multiprocessor** window and right-click in the window that you want to pin. Then choose **Pin to Processor** to lock the window to the selected processor. A window pinned to a processor always displays data from that processor, regardless of the currently focused processor.

For example, if a register window is pinned to Processor 1 and a memory window is pinned to Processor 2, selecting the register window moves the focus to Processor 1. Selecting the memory window moves the focus to Processor 2. The **Multiprocessor** window's **Status** view reflects the change in focus.

Right-Click Menu

The **Multiprocessor** window's right-click menu offers these commands:



Figure 2-58. Multiprocessor Window's Right-click Menu

Multiprocessor Window Tabs

The Multiprocessor window has two tabs, **Status** and **Groups**

Status Tab

Use the **Status** tab ([Figure 2-59](#)) to show the status of each processor in a multiprocessor system. The processor with focus is highlighted with a horizontal bar.

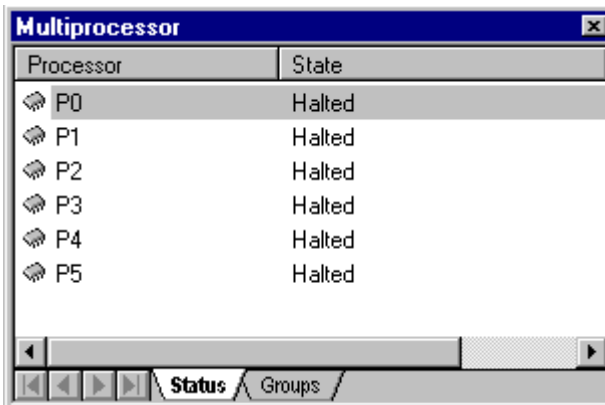


Figure 2-59. Multiprocessor Window – Status View

You can change focus by clicking on a processor in the list.

Groups Tab

Use the **Groups** tab ([Figure 2-60 on page 2-81](#)) to show the current list of multiprocessor groups. A **Default** group is created when you create a new multiprocessor session. The members of the **Default** group are the processors that you checked off under **Multiprocessor System** in the **New Session** dialog box.

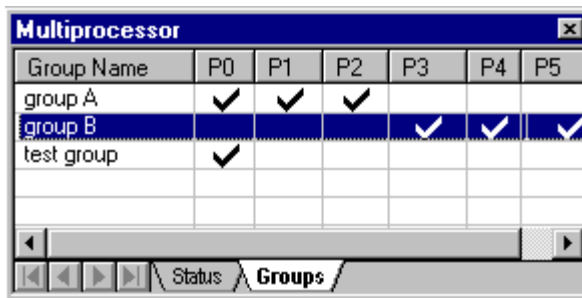


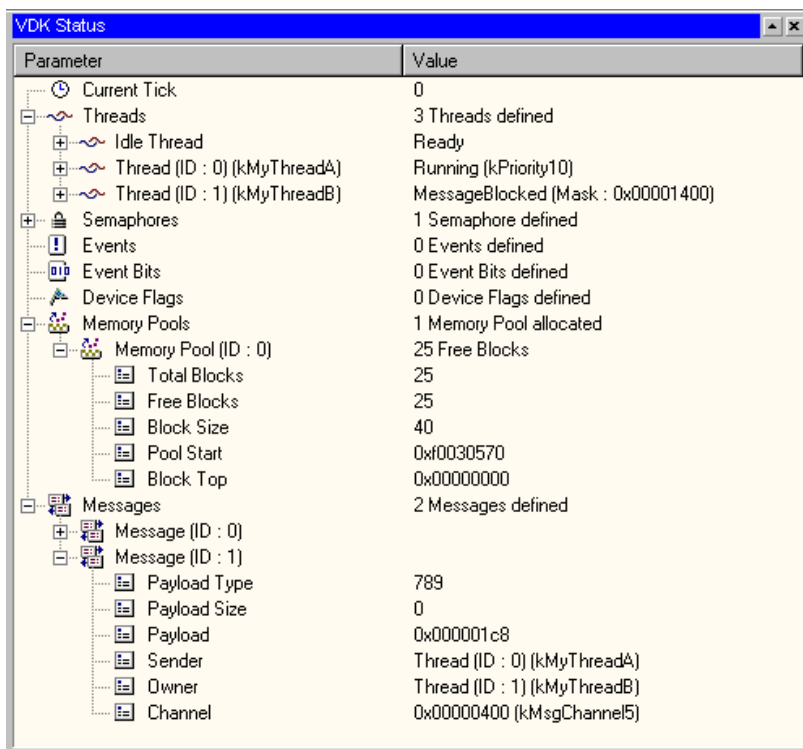
Figure 2-60. Multiprocessor Window – Groups View

With the **Groups** view selected, you can assign one or more processors to a group to apply a multiprocessor operation (**MP Run**, **MP Halt**, **MP Step**, **MP Reset**, and **MP Restart**) only to the processors in the currently selected group.

Right-clicking in the **Multiprocessor** window with **Groups** view selected displays a context menu that lets you add or remove a group.

VDK Status Window

The VDK Status window (Figure 2-61) is available if you enable VDK support for a project.



Parameter	Value
Current Tick	0
Threads	3 Threads defined
Idle Thread	Ready
Thread (ID : 0) (kMyThreadA)	Running (kPriority10)
Thread (ID : 1) (kMyThreadB)	MessageBlocked (Mask : 0x00001400)
Semaphores	1 Semaphore defined
Events	0 Events defined
Event Bits	0 Event Bits defined
Device Flags	0 Device Flags defined
Memory Pools	1 Memory Pool allocated
Memory Pool (ID : 0)	25 Free Blocks
Total Blocks	25
Free Blocks	25
Block Size	40
Pool Start	0xf0030570
Block Top	0x00000000
Messages	2 Messages defined
Message (ID : 0)	
Message (ID : 1)	
Payload Type	789
Payload Size	0
Payload	0x000001c8
Sender	Thread (ID : 0) (kMyThreadA)
Owner	Thread (ID : 1) (kMyThreadB)
Channel	0x00000400 (kMsgChannel5)

Figure 2-61. VDK Status Window

When you halt execution of a VDK program, VisualDSP++ reads data for threads, semaphores, events, event bits, device flags, memory pools and messages and displays the state and status data in this window.

When one of the above VDK entities is created, it is added to the display. An entity is removed from the display when it is destroyed.

Initially, information is displayed in a collapsed state, which shows only the name of the entity and, in some cases, its current state. When a thread is in the Ready state, its priority is displayed.

Clicking the plus sign  next to the name of an entity expands the view.

The possible thread states are as follows.

- Running
- Ready
- SemaphoreBlocked
- EventBlocked
- DeviceFlagBlocked
- MessageBlocked
- SemaphoreBlockedWithTimeout
- EventBlockedWithTimeout
- DeviceFlagBlockedWithTimeout
- MessageBlockedWithTimeout
- Sleeping
- Unknown

Refer to the *VisualDSP++ Kernel (VDK) User's Guide* for details.

VDK State History Window

VDK state history is available only for DSP executables with VDK support. During execution of a VDK-enabled program, if **Full Instrumentation** has been specified for the project, thread and event data are collected in a history buffer. When you halt a running program, the history buffer data is plotted in the **VDK State History** window, described in [Figure 2-62](#). Some features become available only when the data cursor is enabled.

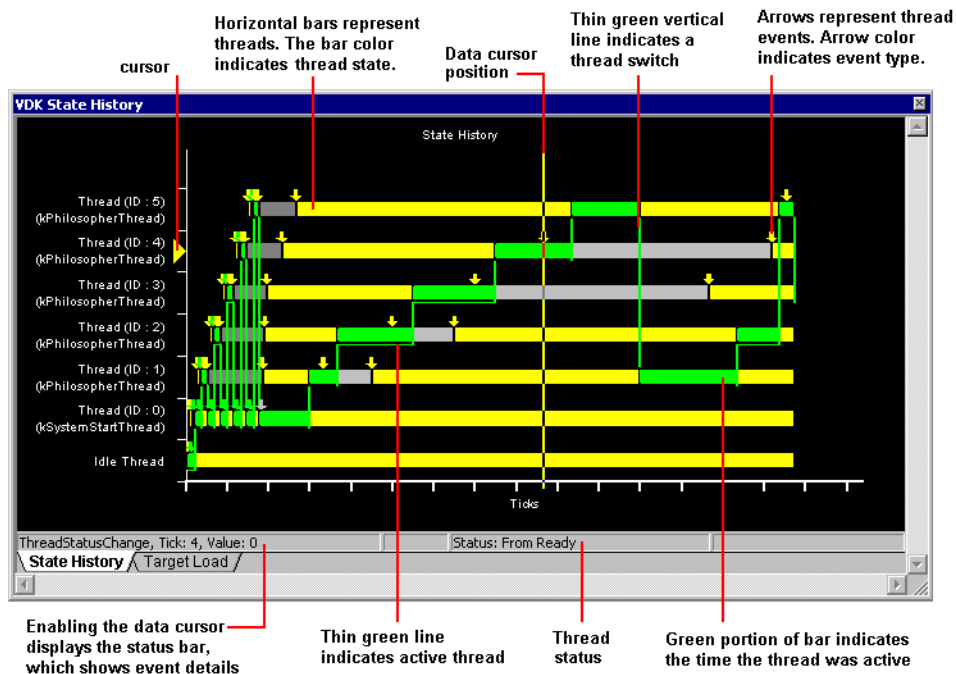


Figure 2-62. Example of a VDK State History Window

Each thread appears as a horizontal bar (thread status bar). The thread's name appears to the left of the bar. When a thread is destroyed, its name no longer appears. Each thread event appears as an arrow above a thread.

Thread Status and Event Colors

Threads and events are coded by color, based on thread status and event type. The colors appear in the horizontal bars (threads) and colored arrows (events) used throughout the plot. Events of the same type are drawn in the same color.

Right-click on the plot and choose **Legend** to display legends that define each color in the **VDK State History** window. To customize colors, right-click on the plot and choose **Properties**.

Trace thread switch history by following the thin green line, which winds through the display, passing under threads to indicate the running thread at any particular time. When a context switch occurs and changes the running thread, a vertical green line is drawn from the previously running thread to the next running thread.

When you use the data cursor, a yellow triangle to the left of a thread name identifies the currently running thread.

Window Operations

The state history status bar (bottom of plot) shows the event's details and thread status. Event details include the event type, the tick when the event occurred, and an event value. The value for a thread-switched event indicates the thread being switched in or out.

Right-click on the plot and choose **Data Cursor** to activate the data cursor, which is used to display event and thread status details. Based on the event that occurred, the thread status changes. Press the keyboard's right arrow key or left arrow key to move to the next or previous event. When the data cursor hits a thread switch event, it moves to the thread being switch in. The yellow triangle to the right of the thread name indicates the currently active thread

Debugging Windows

You can zoom in on a region to examine that area in more detail. Hold the left mouse button down while dragging the mouse to create a selection box. Then release the mouse button to expand the plot. To restore the plot to its original scale, right-click on the plot and choose **Reset Zoom**.

Right-Click Menu

The **VDK State History** window's right-click menu provides easy access to operations you can perform from the state history plot.

Target Load Window

Clicking the **Target Load** tab from the **VDK State History** window displays the **Target Load** window. A target load plot ([Figure 2-63](#)) shows the percentage of time that the target spent in the idle thread.

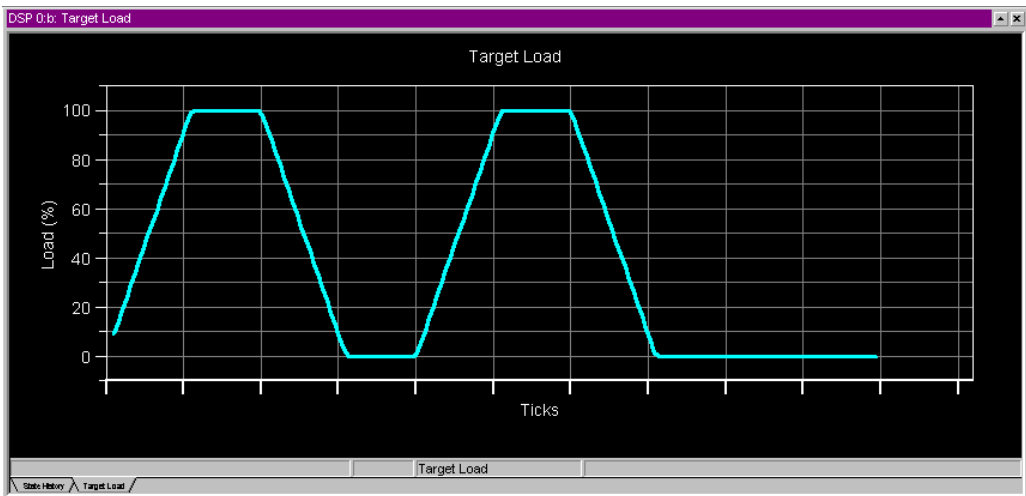


Figure 2-63. Example Target Load Window

A load of 0% indicates that VDK spent all of its time in the idle thread. A load of 100% indicates that the target did not spend any time in the idle thread.

Load data is processed by means of a moving window average.

About Debugging Windows

This section describes useful information about debugging windows.

Editor Window Features

An editor window provides:

- Status icons
- Expression evaluation
- Two view formats (source mode or mixed mode)

Syntax Coloring

Specify colors to help you locate information in the types of files listed in [Table 2-21](#).

Table 2-21. File Types That Support Syntax Coloring

File Type	File Extension
Assembly	.ASM
C	.C
Linker Description Files	.LDF
C++	.CPP
Header	.H
Tool Command Language	.Tcl

Debugging Windows

Right-Click Menu

The editor window's right-click menu provides the commands shown in [Figure 2-64](#).

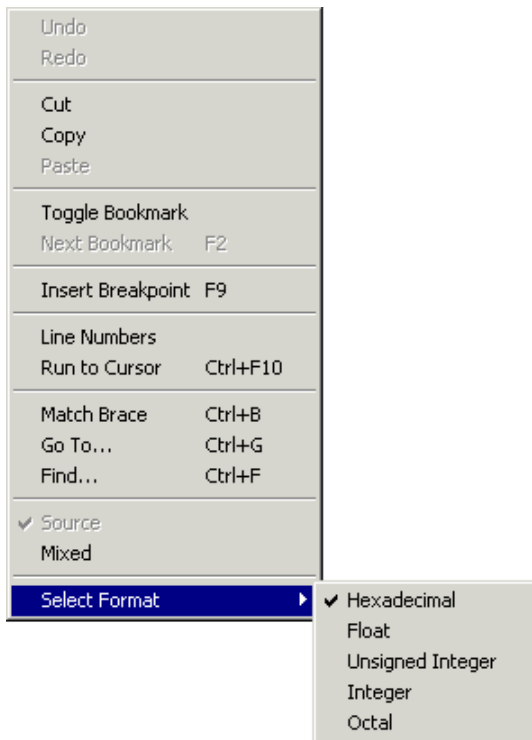


Figure 2-64. Editor Window's Right-Click Menu

Note that an additional command, **Source Tcl Script**, is available when you are editing a Tcl script.

Editor Window Symbols

The editor window's gutter (left margin) displays icons that indicate breakpoints, bookmarks, and the current position of the program counter (PC). [Table 2-22](#) describes these icons.

Table 2-22. Editor Window Symbols

Symbol	Indicates
	The current source line to be executed (PC location)
	An enabled breakpoint
	A disabled breakpoint
	A bookmark

Bookmarks

Bookmarks are pointers in editor windows. You bookmark a location to return to it quickly later.

Context-Sensitive Expression Evaluation

You can evaluate an expression in an editor window only if your .DXX program is loaded for debugging.

As you move the mouse pointer over a variable, with the pointer still on top of the variable, VisualDSP++ evaluates the variable. If the variable is in scope, the value appears in a tool tip window.

Debugging Windows

Viewing an Expression

You can view an expression in different ways.

When the editor window is in mixed mode, you can view a register value by moving the pointer over a register in an assembly instruction. The register contents are displayed in a tool tip.

Highlighting an Expression

You can highlight an expression in the editor window and then move the pointer on top of the highlighted expression to display its value in a tool tip.

Source Mode vs. Mixed Mode

You can specify an editor window's display format. Your two options are source mode and mixed mode.

Source Mode

Source mode, shown in [Figure 2-65](#), displays C code only.

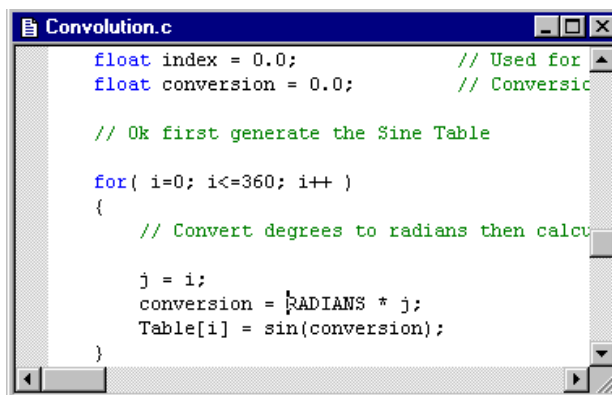


Figure 2-65. Editor Window in Source Mode Format

Mixed Mode

Mixed mode displays the assembled code after the line of the corresponding C code. The assembly code takes a specified color.

i You must compile the source file with debugging information to view the source file in mixed mode.

Figure 2-66 shows an example of the mixed mode format.

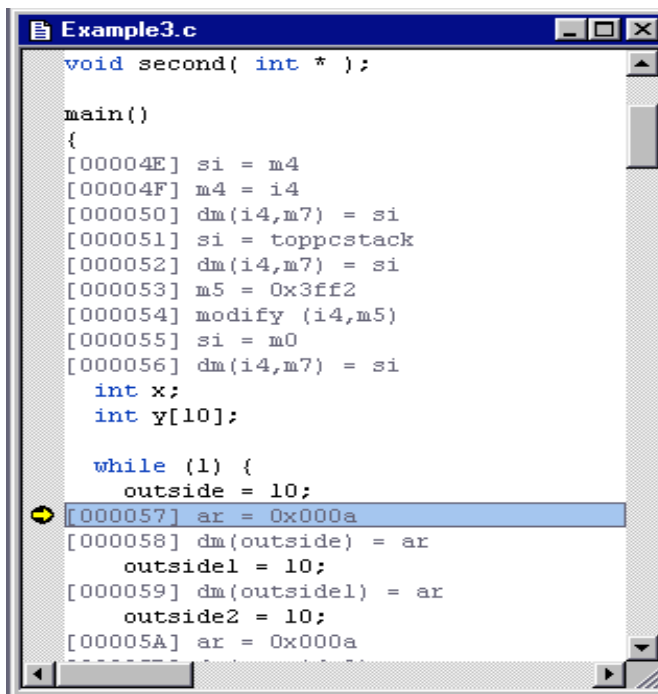


Figure 2-66. Editor Window in Mixed Mode

Expressions in an Expressions Window

Table 2-23 describes the types of expressions that you can enter in an Expressions window.

Table 2-23. Types of Expressions Allowed in an Expressions Window

Expression	Description
Memory address	The memory addresses can be data (DM) or program (PM). The size of an address is 32 bits. Precede memory identifiers with a \$ sign; for example, \$dm16(0x2000).
Register expression	Precede register names with a \$ sign (for example, \$r0, \$ax0, or \$imask).
C/C++ statements	Use standard C/C++ arithmetic and logical operators.
Multiprocessor expression (emulator sessions only)	Expressions can be evaluated on a particular processor by using the format: <pre>@processor_name(expression)</pre> where processor_name is the name of one of your MP processors, and expression is the expression that you want to evaluate on that processor. For example, on an MP system with two processors (master and slave), this expression evaluates the PC register on master:@master(\$PC)

About Expressions

The **Expressions** window displays the current value of each expression as you step through your program. Expressions are evaluated based on the current debug context.

For example, if you enter expression “a” and a global variable “a” exists, you see its value. If you then step into a function that has local variable “a”, you see the local value until the debug context leaves the function. When a variable goes out of context, a string displays next to the variable to inform you that the variable is out of context.

The expressions described above are C expressions. The current syntax also allows you to use registers in expressions. For example, the following is a valid expression:

```
$R0 + $I0
```

Register expressions and C expressions can be mixed in an expression.

Register expressions follow these rules:

- Precede register names with a \$ character.
- Register names can be uppercase or lowercase characters.
- Registers have no context. A register expression always evaluates to the current value of the register.

Plot Windows

Use a plot window to display a *plot*, which is a visualization of values obtained from DSP memory. You can display one or multiple plot windows.

Figure 2-67 shows an example of a plot in a plot window.

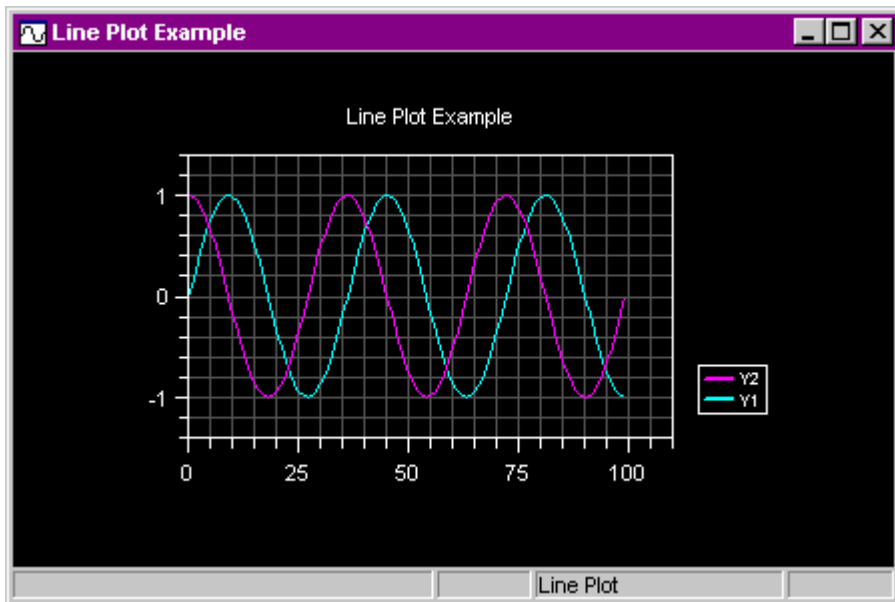


Figure 2-67. Example of a Plot Window

You specify the contents and presentation of the plot. You can modify a plot's configuration and immediately view the revised plot.

From a plot window, you can zoom in on a portion of a plot or view the values of a data point.

You can print a plot, save the plot image to a file, or save the plot's data to a file. For details, refer to the online Help in VisualDSP++.

Plot Window Features

Plot windows include a status bar and a right-click menu.

Status Bar

The status bar, located at the bottom of the plot window, displays the plot type and other information, depending on the plot type and other settings.

The following examples show different plot information displayed on the status bar.



Figure 2-68. Examples of Status Bar Information for Plots

In a waterfall plot, the status bar indicates the azimuth and elevation viewing angles. If you zoom in on a region, the status bar indicates that zoom is enabled. When you use the data cursor, the status bar shows the selected point's data value.

Debugging Windows

Right-Click Menu

The plot window's right-click menu is shown in [Figure 2-69](#).

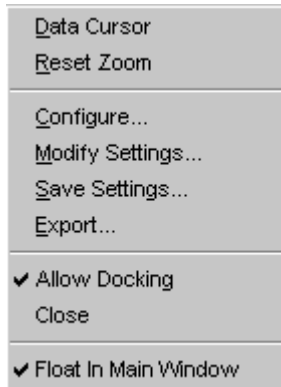


Figure 2-69. Plot Window's Right-Click Menu

This menu provides access to the standard window options (docking, closing, and floating in the main window) and to the plot window features described in [Table 2-24 on page 2-97](#).

Table 2-24. Plot Window Operations

Feature	Description
Data cursor position	You can move the plot window's data cursor over a data point and view the point's memory data value in the left side of the plot window's status bar. Use the keyboard's arrow keys to move around on the graph.
Zooming	You can zoom in to view a specified region of the plot. You can also reset the plot window to its initial full-scale display.
Plot configuration	From the plot window, you can access the Plot Configuration dialog box, from which to add, remove, or modify data sets. You can also change the plot type and rename the plot.
Settings modification	You can customize the plot's appearance. You can specify settings for the plot (grids, colors, margins, fonts, axes, and so on), and you can specify settings for each data set (data processing).
Settings storage and retrieval	You can save plot configuration settings for future use. Plot settings are stored, but the data is not stored. You can retrieve settings (.VPS file) and load new plot data.
Export	You can export the plot image to various destinations, including the Windows clipboard. Save the plot image as a file (JPG, GIF, TIF, EPS, TXT, or DAT format) or print a hard copy.

Plot Window Statistics

You can view various statistics (mean, standard deviation, signal-to-noise ratio (SNR), minimum data value, and maximum data value) while displaying a plot. Note that statistics apply only to the portion of data that is visible. When the plot is zoomed, the statistics are re-calculated only for the visible area.

Figure 2-70 shows statistics displayed for a portion of audio data.

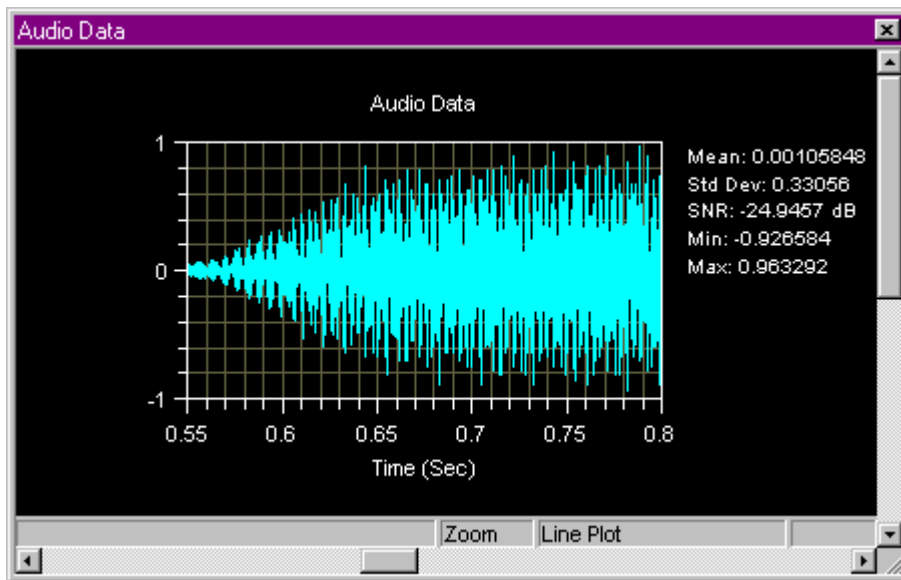


Figure 2-70. Statistics Displayed for a Portion of Audio Data

For details about viewing statistics, refer to the VisualDSP++ online Help.

Plot Configuration

A plot configuration comprises two parts:

- Data values
- Presentation (configuration) settings

You create data sets and configure the data for each data set. A *data set* is a series of data values in DSP memory. You specify the memory location, the number of values, and other options that identify the data. 3-D plots require additional specifications for row and column counts.

VisualDSP++ offers many plot presentation options. You choose the type of plot (for example, waterfall) and the axis associated with each data set. You configure options for titles, grids, fonts, colors, and axis presentation.

You can recall a plot from saved settings. You must identify the settings (.VPS file) to be used. VisualDSP++ uses these settings and reads DSP memory to generate and display a plot in a plot window.

Plot Window Presentation

You can customize the presentation of a plot window to fit your needs. You configure presentation settings from the **Plot Setting** dialog box, which you invoke as follows.

- Right-click from within a plot window
- Click the **Settings** button from in **Plot Configuration** dialog box

Debugging Windows

The **Plot Settings** dialog box provides the tabs shown in [Figure 2-71](#).

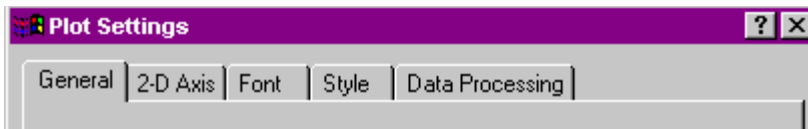


Figure 2-71. Tabs in the Plot Setting Dialog Box

Options on the tabbed pages enable you to configure the plot window's presentation. On the **Style** page, for example, you can easily specify symbols for a data set as well as line type and width, as shown in [Figure 2-72](#).

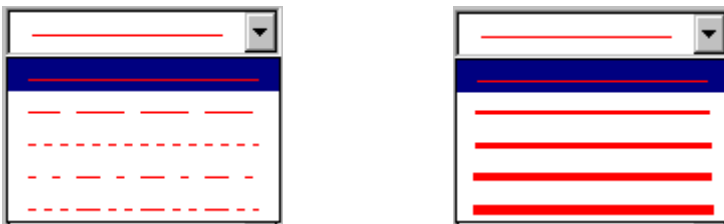


Figure 2-72. Line Styles

In addition to the many presentation options, you can select a rectangular area, as shown in [Figure 2-73](#), and zoom in on it.

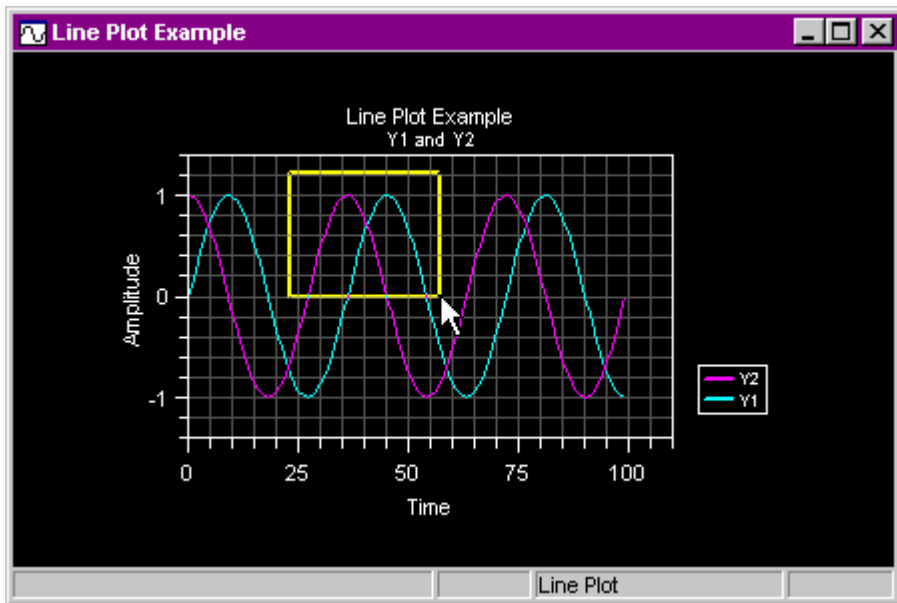


Figure 2-73. Zooming In on a Selected Area

Plot Presentation Options

You can configure a plot's presentation. Depending on the type of plot, many options are available.

In the **Plot Settings** dialog box, these options are grouped by function on tabbed pages, described in [Table 2-25](#).

Table 2-25. Plot Settings Options by Page

Page	Options That You Can Specify
General	Title and subtitle, grid lines, margins, background colors, and legend
2-D Axis	For X-axis and Y-axis: axis titles, start and increment values, scales
3-D Axis	For X-axis, Y-axis, and Z-axis: axis titles, Z-axis settings, step sizes, scale multipliers, color and mesh
Font	Font name, color, and size
Style	For a data set: line type, width, color; symbol and type
Data Processing	For a data set: data processing algorithm, sample rate, number of stored, and triggering

You can specify a plot's presentation options before you generate the plot (while configuring the plot), or you can specify plot options after generating the plot.

Image Viewer

The VisualDSP++ **Image Viewer** window enables you to perform these operations:

- View an image. You can display BMP, JPEG, PPM, or MPEG data from DSP memory or from a file on your PC.
- Correct the gamma attributes of an image. For a color image, you can adjust the red, green, and blue pixel values. On a grayscale image, you can adjust darkness only.
- Copy an image to the Windows clipboard
- Print an image or save it to a file
- Export an Image

You select the image source (from DSP memory or a file on your PC) and specify image attributes. If the image is located in DSP memory, you must specify the image's address, size, and format.

Debugging Windows

The **Image Viewer** window, shown in [Figure 2-74](#), renders the image and provides scroll bars and buttons for zooming in and out.

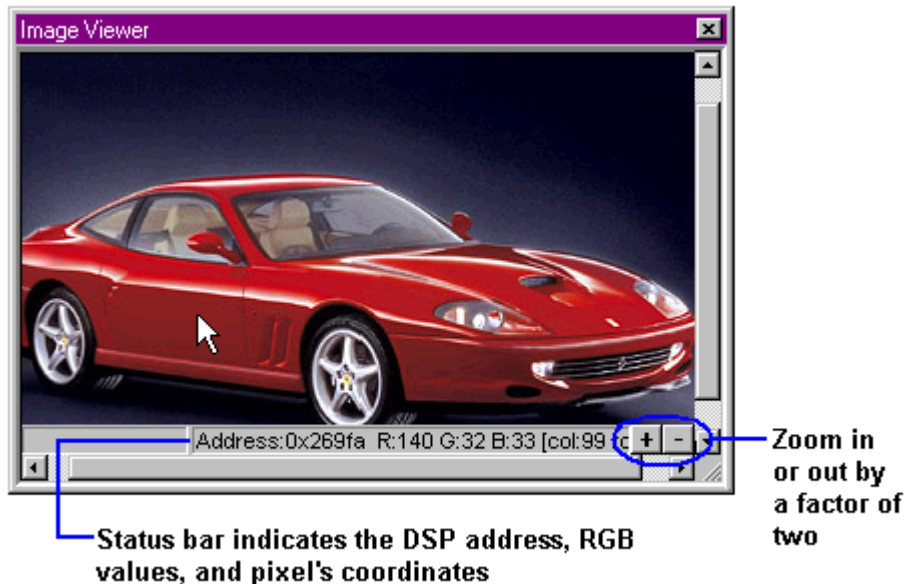


Figure 2-74. Image Viewer Window

As you move the mouse over the image, the status bar indicates:

- DSP address where the selected pixel is located
- Red, green, and blue (RGB) pixel values
- Pixel coordinates (column and row)

i Pixel color depth is 24 bits for color images and 8 bits for grayscale images.

Right-Click Menu

Figure 2-75 shows the **Image Viewer** window's right-click menu.



Figure 2-75. Image Viewer Window's Right-Click Menu

Table 2-26 describes the menu commands.

Table 2-26. Right-Click Menu Commands

Command	Purpose
Configure	Opens the Image Configuration dialog box, from which you can specify image attributes
Update Now	Reads the image data from DSP memory
Reset Zoom	Displays the image in its original size
Export	Opens the Export Image dialog box, from which you can copy or print the image
Gamma Adjust	Opens the Gamma Correction dialog box, from which you can adjust image color

Table 2-26. Right-Click Menu Commands (Cont'd)

Command	Purpose
Play Video	Plays an MPEG video clip
Stop Video	Ends the playing of a video clip

Image Configuration Dialog Box

When using the Image Viewer, you must configure specifications for the image. [Table 2-27](#) describes the buttons and fields in the **Image Configuration** dialog box.

Table 2-27. Buttons and Fields in the Image Configuration Dialog Box

Item	Purpose
DSP Memory	Specifies Image or Video
File	Specifies a file on your PC. You then specify the file name and path. Clicking Browse opens the Select Image Import File dialog box, from which you navigate to the file.
Memory selection	Specifies the memory
Image start address (hex)	Specifies the first location of the image data
Horizontal pixels	Specifies the number of horizontal pixels
Vertical pixels	Specifies the number of vertical pixels
Bits per pixel	Specifies the number of bits per pixel. For color images, only 24 bits per pixel are currently allowed. For grayscale images, only 8 bits per pixel are currently allowed.
Stride	Specifies the skip count. The default is one.
Image format	Specifies the format (RGB or Gray Scale)
Video bytes	Specifies the number of bytes (video images only)

Gamma Correction Dialog Box

When using the Image Viewer, you can adjust the image gamma. For color images, you can adjust red, green, and blue independently or in tandem. For grayscale images, you can only adjust the black-white balance.

[Table 2-28](#) describes the buttons and fields in the **Gamma Correction** dialog box.

Table 2-28. Buttons and Fields in the Gamma Correction Dialog Box

Item	Purpose
Red	Specifies the red value
Green	Specifies the green value
Blue	Specifies the blue value
Link	Adjusts the red, green, and blue values at the same time (not for video images)
Gray	Specifies the black value (grayscale images only)

Export Image Dialog Box

When using the Image Viewer, you can export an image to the Windows clipboard, a file, or to the printer.

The **Export Image** dialog box contains the buttons and fields described in [Table 2-29](#).

Table 2-29. Buttons and Fields in the Export Image Dialog Box

Item	Purpose
Clipboard	Copies the image to the Windows clipboard
File	Specifies a file name and path. The file name and path appear in the text box. Click Browse to navigate your system.
Printer	Sends the image to the default printer

3 DEBUGGING

This chapter describes VisualDSP++ debugging tools that you use during single-processor and multiprocessor (emulation only) debug sessions. The topics are organized as follows.

- “Debug Sessions” on page 3-2
- “Code Analysis Tools” on page 3-7
- “Program Execution Operations” on page 3-17
- “Simulation Tools” on page 3-22
- “Simulation Options” on page 3-23
- “Image Viewer” on page 3-35
- “Flash Programmer” on page 3-35

Debug Sessions

You run the DSP projects that you develop as *sessions* (debug sessions).

A session is defined by the elements described in [Table 3-1](#).

Table 3-1. Specifying a Debug Session

Element	Description
Debug target	The debug target is the software module that controls a type of debug target (a simulator or emulator). The <i>simulator</i> is software that mimics the behavior of a DSP chip. Simulators are used to test and debug DSP code before a DSP chip is manufactured. An <i>emulator</i> is software that “talks” to a hardware board that contains one or more actual DSP chips.
Platform	For a given debug target, several platforms may exist. For a simulator, the platform defaults to the identically named DSP simulator. When the debug target is an EZ-ICE® board, the platform is the board in the system on which you want to focus. When the debug target is a JTAG emulator, the platforms are the individual JTAG chains.
Processor	Multiple processors can exist for a given debug target and platform. When you create an executable file, the processor is specified by the Linker Description File (.LDF) and other source files.

When you set up a session, you set the focus on a series of more specific elements.

The target platform and processor settings specify the debug session. A default session name is automatically generated. You can further identify the session by modifying the default name, choosing a more meaningful name.



A well-chosen name can prevent confusion later.

Debug Session Management

You can run several debug sessions at once and can dynamically switch between sessions.

You typically run multiple debug sessions to write different versions of your program to compare their operating efficiencies. Another reason for running multiple sessions is to debug completely different programs without having to run multiple instances of VisualDSP++.

Breakpoints

You can set breakpoints in your executable program. A breakpoint may be set at any address in program memory. Program execution halts at the address at which the breakpoint is located.

Watchpoints

Watchpoints are like breakpoints. Watchpoints, however, trap on a specified condition.

You can set watchpoints on registers, stacks, and memory ranges. Reaching the condition halts program execution and updates all windows.



Watchpoints are available during simulation only.

Multiprocessor (MP) Debugging

Often, performance-based products require two or more DSPs. A system built with multiple DSPs is called a *multiprocessor* system, and a system built with a single DSP is called a *single-processor* system.

Setting Up a Multiprocessor Debug Session

 The SHARC simulator does not support multiprocessor debugging.

The first step in setting up a multiprocessor debug session is to develop a multiprocessing project by using the multiprocessing capabilities of the linker and an `.LDF` file to describe the multiprocessing system.

Refer to your DSP's Linker and Utilities Manual, especially the sections about `SHARED_MEMORY{ }` and `MPMEMORY{ }` commands.

The second step is to use the VisualDSP ICE Configurator utility to describe the hardware to the VisualDSP++ software if you are running a JTAG emulator session. VisualDSP++ uses this description when you set up your debug session. Refer to your DSP's *Hardware Reference* for information about the ICE Configurator.

After specifying your hardware system, build your project.

The first time that you launch VisualDSP++ for a new project, the **New Session** dialog box opens to enable you to configure the MP session. The next time you launch VisualDSP++, the debug session is automatically configured for you.

Debugging a Multiprocessor System

Debugging a multiprocessor system requires that you synchronously run, step, halt, and observe program execution operations in all the processors at once.

The following capabilities help to speed a multiprocessor debug session.

- Multiprocessor debug commands that operate like the commands that you use while debugging a single processor, except that they work synchronously on *all active processors* in the currently selected MP group
- **Multiprocessor** window
 - The **Status** tab enables you to view the status of each processor and switch processor focus.
 - The **Groups** tab enables you to group processors into multiple, logical units to which all MP commands are applied.
- Window pinning. Note that you can use pinning and the processor status items in the **Multiprocessor** window with single-processor debug commands to debug individual processors in an MP session.
- Window color specification

Focus and Pinning

In a multiprocessor debug session, you often have to examine the behavior of a single processor to better understand its interaction with the other processors on the target.

When you debug a single processor in an MP session, the processor being debugged has the *focus*.

By *pinning a window* to a processor, you dedicate the window, such as a memory window, to a particular processor in a multiprocessor group. Pinning statically associates a window to a specific processor.



Before debugging, open and pin the register windows and Memory windows you plan to use. If you do not pin them, these windows display information for any processor that has focus.

Debug Sessions

When a window is pinned to a processor, a pin icon appears in the window's upper-left corner. For example:



Window Title Bar Information

Figure 3-1 shows a pinned window in a multiprocessor debug session.

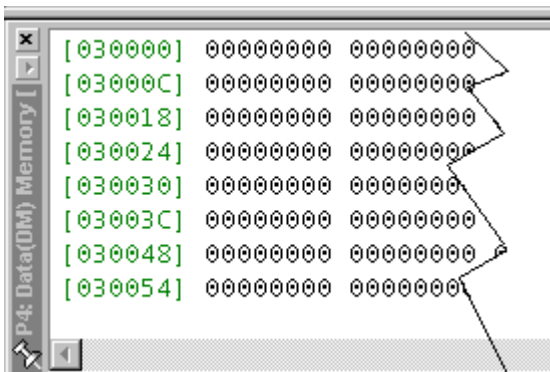


Figure 3-1. Pinned Window in a Multiprocessor Debug Session

The title bar of a pinned window shows:

- Processor name
- Pushpin icon  to indicate that the window is pinned
- Window title
- Number format, such as Hexadecimal (for windows that support multiple formats)

Additional Focus Indication

If configured, VisualDSP++ shades unfocused windows with a specified color. You can specify the background color of focused and unfocused windows.

Code Analysis Tools

You use code analysis tools to examine your code's behavior and locate areas that may be optimized for better performance.

VisualDSP++ provides these code analysis tools:

- Statistical profiles and linear profiles
- DSP memory plots

Statistical Profiling and Linear Profiling

VisualDSP++ provides two profiling methods that measure program performance by sampling the target's PC register to collect data. During program development you use linear profiling with simulator targets, and you use statistical profiling with emulator targets.

The **Linear Profiling Results** window and **Statistical Profiling Results** window display the data collected by these two profiling methods and indicate where the application is spending its time.

The window's title (**Linear Profiling Results** or **Statistical Profiling Results**) depends on whether this tool is used during simulation or emulation.

Simulation

Linear profiling with the simulator is not statistical because the simulator samples **every** PC executed. This feature provides an accurate and complete picture of what was executed in your program.

Linear profiling is much slower than statistical profiling. Simulator targets support linear profiling but do not support statistical profiling.

Emulation

A statistical profile measures the performance of a DSP program by sampling the target's Program Counter (PC) register at random intervals while the target is running the DSP program. The areas of the program where most of the PCs are concentrated are where most of the time is spent in executing the program.

Statistical profiling provides a more generalized form of profiling that is well suited to JTAG emulator debug targets. Emulator targets do not support linear profiling.

JTAG sampling is completely non-intrusive, so the process does not incur additional runtime overhead.

Traces

A trace captures a history of processor activity during program execution. You run a trace (also called an execution trace or a program trace) to analyze the run-time behavior of your DSP program, enable I/O capabilities, and simulate source to target data streaming.

A trace includes the following information.

- Buffer depth (instruction lines)
- Cycle count
- Instructions executed such as memory fetches, program memory writes, and data/memory transfers

Viewing the disassembled instructions that were performed can also help you to analyze code behavior.

DSP Memory Plots

You can display DSP memory as a *plot* in a plot window. This section describes the various plot types available in VisualDSP++. For information about plot windows, see [“Plot Windows” on page 2-94](#).

Plot Types

You specify a plot as one of the plot types described in [Table 3-2](#).

Table 3-2. Available Plot Types

Plot Type	Description	Requires
Line plot	Displays points connected by a line	Y value for each data point
X-Y plot	Similar to a line plot, but also uses X-axis data	X value and Y value for each data point
Constellation plot	Displays a symbol at each data point	X value and Y value for each data point
Eye diagram	Typically used to show the stability of a time-based signal	Y value for each data point
Waterfall	3-D plot typically used to show the change in frequency content of signal over time	Z value for each data point
Spectrogram plot	2-D plot displays amplitude data as a color intensity	Z value for each data point

The X, Y, and Z values are read from DSP memory.

Line Plots

A line plot (shown in [Figure 3-2](#)) displays a range of DSP memory values connected by a line. The values read from DSP memory are assigned to the Y-axis. The corresponding X-axis values are automatically generated.

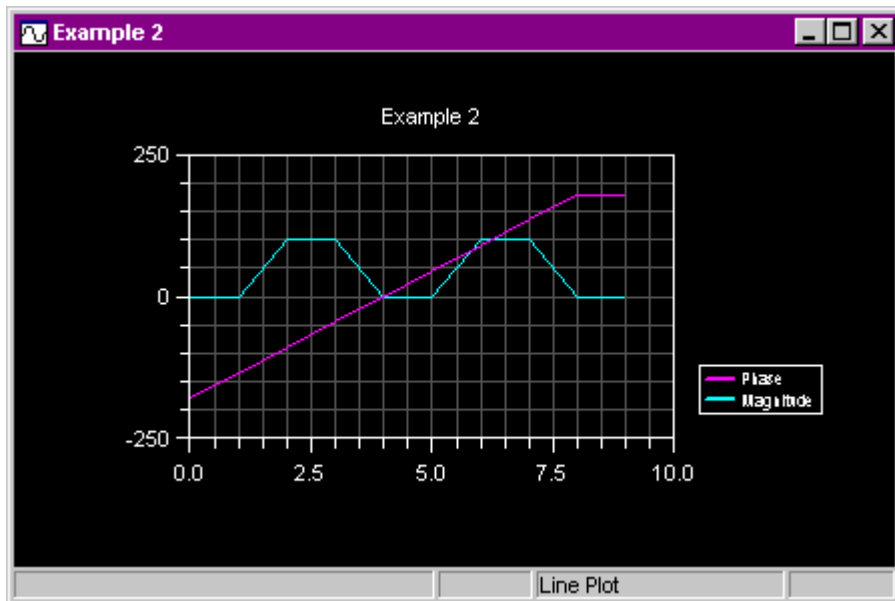


Figure 3-2. Line Plot Example

You can plot multiple data sets on a single graph.

X-Y Plots

An X-Y plot (shown in [Figure 3-3](#)) requires an X value and a Y value for each data point. Unlike a line plot, an X-Y plot requires the X-axis data.

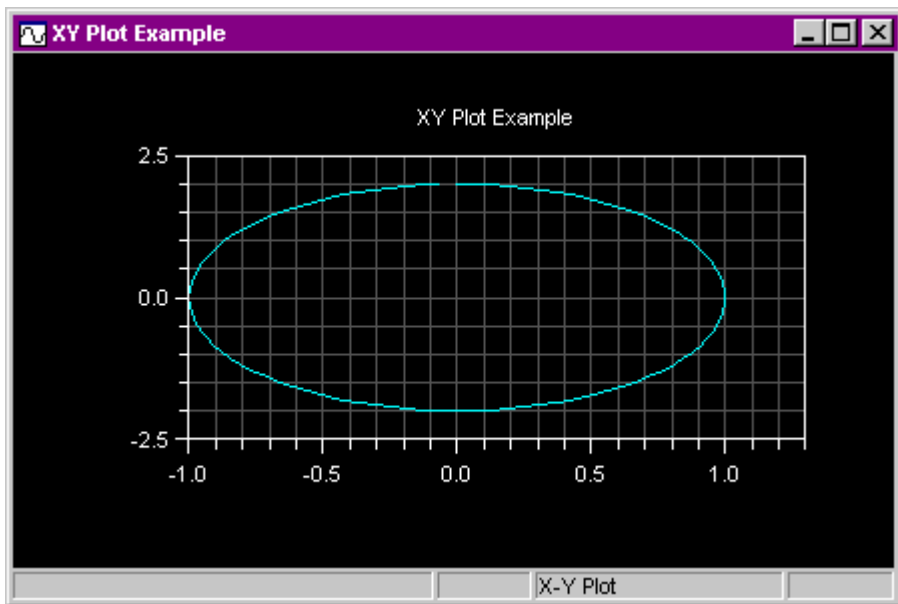


Figure 3-3. X-Y Plot Example

The X data and Y data are specified separately in a user-defined memory location. The number of X and Y points must be equal.

Constellation Plots

A constellation plot (shown in [Figure 3-4](#)) displays a symbol at each (X,Y) data point.

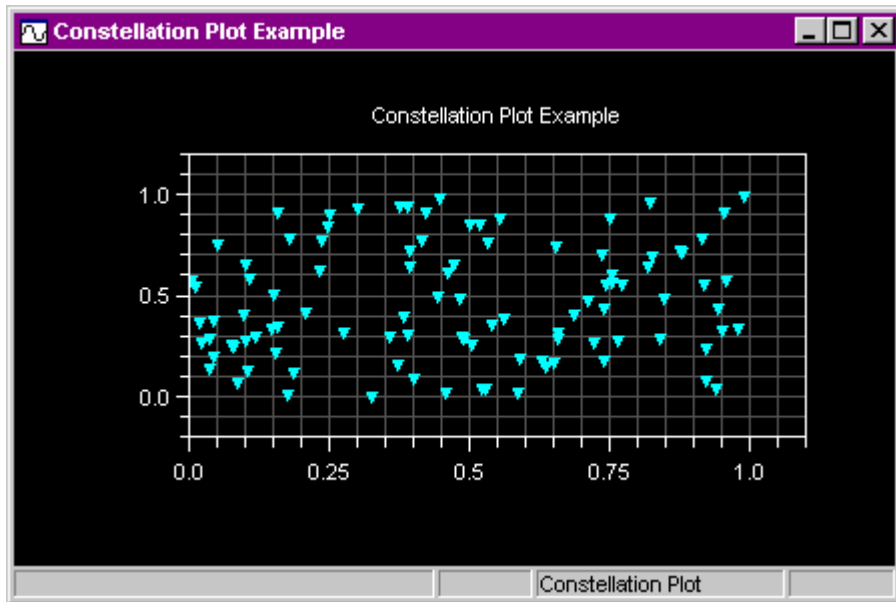


Figure 3-4. Constellation Plot Example

The X and Y data are specified separately in a user-defined DSP memory location. The number of X and Y points must be equal.

Eye Diagrams

An eye diagram plot (shown in [Figure 3-5](#)) is typically used to show the stability of a time-based signal. The more defined the eye shape, the more stable the signal.

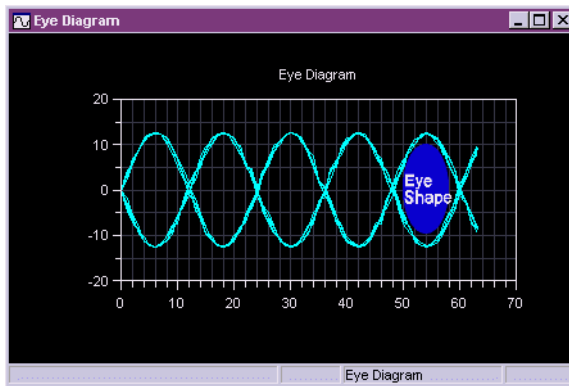


Figure 3-5. Eye Diagram Plot Example

This plot works like a storage oscilloscope by displaying an overlapped history of a time signal. The eye diagram plot processes the input data and optionally looks for a threshold crossing point (default 0.0). A trace is plotted when the threshold crossing is reached. Plotting continues for the remainder of the trace data.

When a breakpoint occurs (or a step is performed), the plot data is updated and a new trace is plotted. The eye diagram uses a data shifting technique that stores the desired number of traces in a plot buffer (default is ten traces). When the number of traces is exceeded, the first trace shifts out of the buffer and the new trace shifts into the last buffer location. This technique is referred to as *first in, first out* (FIFO).

You can modify options for threshold value, rising trigger, falling trigger, and the number of overlapping traces.

Waterfall Plots

A waterfall plot (shown in [Figure 3-6](#)) is typically used to show the change in frequency content of signal over time.

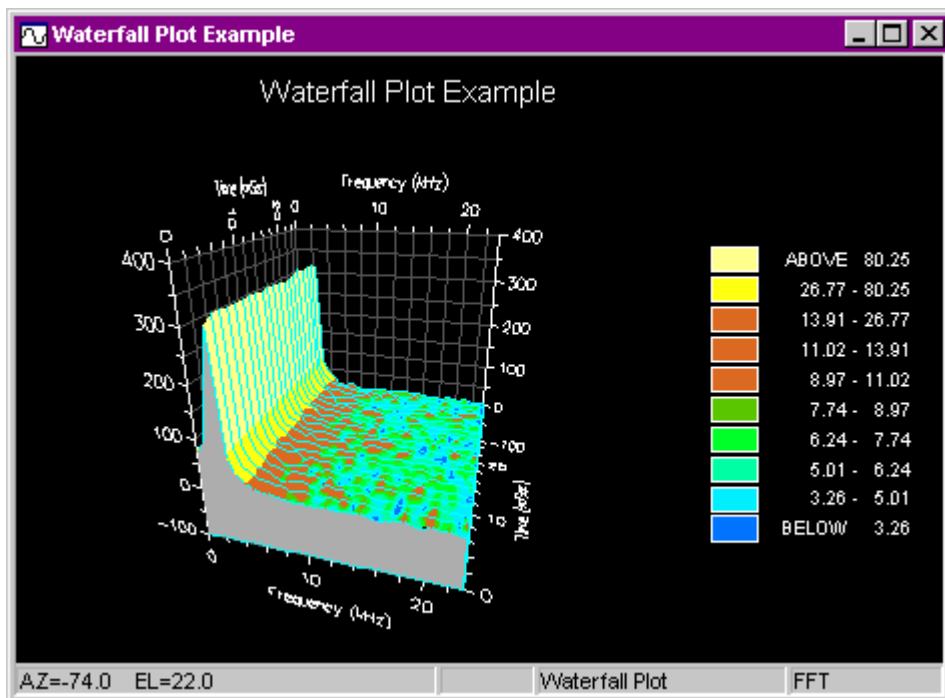


Figure 3-6. Waterfall Plot Example

The plot comprises multiple line plot traces in a 3-D view. Each line plot trace represents a slice of the waterfall plot.

The easiest way to create a waterfall plot is to define a 2-D array in C code (a grid). The first array dimension is the number of rows in the grid, and the second dimension is the number of columns in the grid. The number of columns is equal to the number of data points in each line trace.

A time-based signal is sampled at a predefined sampling rate and stored as a slice in the grid (row 0, columns 0 through N).

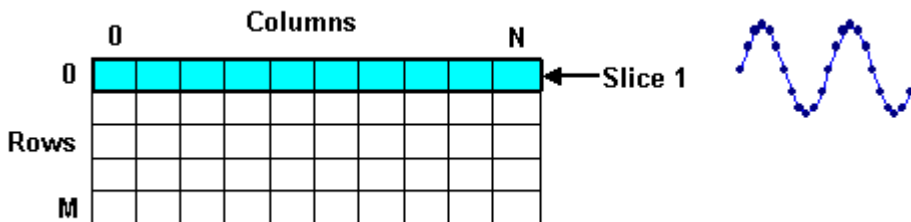


Figure 3-7. Grid of Sampled Data

The next time signal is sampled and stored (in row 1, columns 0 through N). This process continues until all the rows are filled.

By default, an FFT performed on each slice results in a frequency output display. You can use a color map on the **3-D Axis** page of **Color Settings** dialog box to enhance the display. Each color corresponds to a range of amplitude values.

The plot output displays a legend showing each color and associated range of values.

You can rotate the waterfall plot to any desired azimuth and elevation by using the keyboard's arrow keys.

Spectrogram Plots

A spectrogram plot (shown in [Figure 3-8](#)) displays the same data as a 3-D waterfall plot, except in a 2-D format.

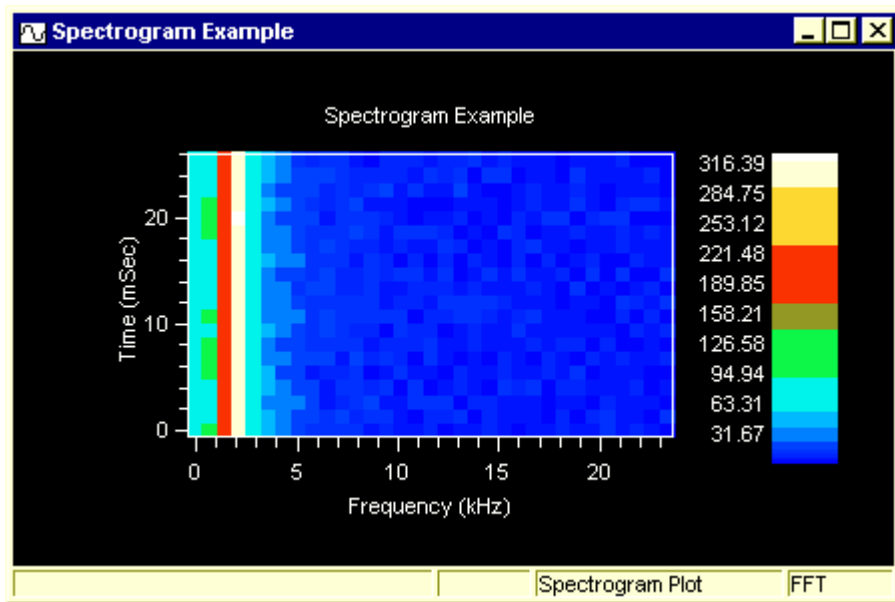


Figure 3-8. Spectrogram Plot Example

Each (X,Y) location displays as a color representing the amplitude of the data. By default, an FFT performed on each slice results in a frequency output display. A legend displays the colors and associated range of values.

Program Execution Operations

When you start up VisualDSP++, by default, it attaches to the previous session. You can override this behavior, and instead, force VisualDSP++ to start a new session.

When you load and run your program, use VisualDSP++ features to step, break, and halt the program.

Selecting a New Debug Session at Startup

If you had a problem, such as a corrupted workspace, in your last debug session, use the following procedure to force a fresh session at startup.

Note: VisualDSP++ must be closed before performing the following procedure.

1. Hold down the keyboard's **Ctrl** key.



Do not release the **Ctrl** key until the **Session List** dialog box appears, as described in the next step.

2. Invoke VisualDSP++ as you normally do.

Typical methods include the using the Windows **Start** button sequences, clicking desktop icons, or using Windows Explorer.

The **Session List** dialog box appears.

3. Specify and activate a debug session.

If you launch VisualDSP++ in stand-alone mode, ensure that the session is configured correctly **before** you load your program.

Loading the DSP Executable Program

Once you specify the debug session, you can begin the session by loading the DSP executable program.

After a successful build of the target executable, VisualDSP++, if configured, loads the executable automatically to the current session when the session processor type matches the project's processor. When the current session processor does not match the project's processor type, you are prompted to choose another session.

If automatic load is not configured, VisualDSP++ does not try to load the executable automatically after a successful build.



The target must be an executable (.EXE) file.

This debugging feature saves time because you do not have to load the executable target manually and you can start to debug right after a successful build of the project.

Using Program Execution Commands

You can run program execution commands from the **Debug** menu or by clicking toolbar buttons.

Executable files run until an event such as a breakpoint, watchpoint, or user-issued **Halt** command stops execution. When program execution halts, all windows are updated to current addresses and values.

Multiprocessor (MP) commands operate like single-processor commands with one exception—they perform an action on *all* processors in the MP group. **MP Run**, **MP Halt**, and **MP Step** are synchronous operations, which means that all processors in the currently selected group execute on exactly the same clock cycle. Some commands have keyboard shortcuts.

Use the commands described in [Table 3-3 on page 3-19](#) to control program execution.

Table 3-3. Commands Used to Control Program Execution

Command	Description
Run or MP Run	Runs an executable. The program runs until an event stops it, such as a breakpoint or user intervention. When program execution halts, all windows update to current addresses and values.
Halt or MP Halt	Stops program execution. All windows are updated after the DSP halts. Register values that have changed are highlighted, and the status bar displays the address where the program halted.
Run to Cursor	Runs the program to the line where you left your cursor. You can place the cursor in editor windows and Disassembly windows.
Step Over	(C/C++ code only in an editor window) Single steps forward through program instructions. If the source line calls a function, the function executes completely, without stepping through the function instructions.
Step Into or MP Step Into	(editor window or Disassembly window) Single steps through the program one C/C++ or assembly instruction at a time. Encountered functions are entered.
Step Out Of	(C/C++ code only in an editor window) Performs multiple steps until the current function returns to its caller, and stops at the instruction immediately following the call to the function.

Restarting the Program

You can set the program counter (PC) to the first address of the interrupt vector table.

Performing a Restart during Simulation

In the simulator, restart works like a reset; however, the target's memory does not change. All registers are reset to their initial values.



Memory is not reset. Thus, C and assembly global variables are **not** reset to their original values. Your program may behave differently after a restart. To re-initialize these values, reload your .DXE file.

Performing a Restart during Emulation

In the emulator, restart works exactly like a reset. Only registers with default reset values are affected. All other registers remain unchanged.

Using Breakpoints

An enabled breakpoint halts program execution at a specific instruction or address. You can enable and disable breakpoints as well as add and delete breakpoints.

A disabled breakpoint is set up, but not turned on. A disabled breakpoint does not stop program execution. It is dormant and may be used later.

A *break* occurs when the conditions that you specify are met.

You can quickly place an unconditional breakpoint at an address in a **Disassembly** window or editor window by using one of these options:

- Select the address and click the **Toggle Breakpoint** button .
- Double-click on the line in the **Disassembly** or editor window.

Symbols in the left margin of a **Disassembly** window or editor window indicate breakpoint status, as shown in [Table 3-4](#).

Table 3-4. Breakpoint Status Symbols

Symbol	Indicates
	An enabled (set) breakpoint
	A disabled breakpoint (recognized, but cleared)

 In addition to software breakpoints, you can also use *hardware breakpoints* in an emulator debug session.

Using Unconditional and Conditional Breakpoints

You can configure a breakpoint to occur when the program counter (PC) reaches a specific address. This type of breakpoint is an *unconditional breakpoint*, because it occurs when it is reached.

You can configure a breakpoint to occur when various conditions (criteria) are met. This type is called a *conditional breakpoint*. The conditions may include:

- A user-defined *expression* that must evaluate to TRUE
- A *skip* (count) that specifies the number of times to skip over the breakpoint before finally halting

If both an expression and skip are set, execution stops when the breakpoint is reached “*n*” times when the expression is TRUE, where *n* represents the skip count. When the expression is empty, execution stops when the breakpoint is reached “*n*” times.

Using Watchpoints

Similar to breakpoints, watchpoints stop program execution when user-specified conditions are satisfied. Watchpoints, however, allow you to set a *condition* such as a memory read or stack pop, to halt events.

 You can use watchpoints only during simulation.

Watchpoints, unlike breakpoints, are not attached to a specific address. A watchpoint halts anywhere in your program once the watchpoint conditions are satisfied.

Simulation Tools

Interrupts

Before you even have the DSP, you can use interrupts within VisualDSP++ to simulate the DSP's behavior.

Use interrupts to simulate external interrupts in your program. When you use interrupts with streams, your program simulates real-world operation of your DSP system.

Input/Output Simulation (Data Streams)

In many products, DSPs exist as part of a larger system where they can act as a host or a slave. They can drive other devices or take part in processing a subset of data. Because of their extensive I/O capabilities, Analog Devices DSPs excel in these roles.

You can use data streams to transmit data between:

- A device and a file
- A device and a device

Simulation Options

Depending on your selected DSP, several simulator options are available on submenus under the **Settings** menu.

The **Anomalies** submenu provides these options:

- **Shadow Write** (ADSP-2116x DSPs only)

This command opens the **Configure Simulator Event** dialog box, from which you can configure reporting for Shadow Write anomalies.

- **SIMD FIFO** (ADSP-2116x DSPs only)

This command opens the **Configure Simulator Event** dialog box, from which you can configure reporting for SIMD FIFO anomalies.

The **Events** submenu provides these options:

- **FP Denorm**

This command opens the **Configure Simulator Event** dialog box, from which you can configure the reporting of the generation of a floating-point denormal result. Refer to your DSP's Hardware Reference for more information about DSP floating-point operations.

- **Short Word** (ADSP-2106x DSPs except for the ADSP-21065L)

This command opens the **Configure Simulator Event** dialog box, from which you can configure reporting for short-word accesses that fail.

The **Simulator** submenu provides the **CLKDBL** option for ADSP-21161 DSPs only. This command enables the 2x clock double circuitry. You can use this command to configure CLKOUT as either 1x or 2x the rate of CLKIN.

Simulation Options

The **Load Sim Loader** submenu provides these options:

- **Boot from Host**
- **Boot from PROM**
- **Boot from Link**
- **Boot from SPI**
- **None of Above**

ADSP-21x6x Anomalies

If you know your silicon has anomalies, the simulator enables you to configure reporting for the following anomaly events.

- Short Word anomaly
- Shadow Write FIFO anomaly
- SIMD read from internal memory with Shadow Write FIFO hit anomaly

Short Word Anomaly (ADSP-2106x DSPs Except for the ADSP-21065L)

Short-word accesses (read or write) fail after following a stalled instruction. Any access (read or write) to short-word memory space can fail if it follows a stalled instruction or causes an instruction stall (see below for examples).

A DMA process cannot cause this anomaly. It is restricted to conditions set up in the core processor. A DMA process is not impacted by this anomaly (that is, the DMA will function correctly even if the anomaly occurs). Refer to your DSP's Hardware Reference.



The occurrence of the failure varies with temperature, voltage, and frequency.

An instruction can stall because of the following circumstances.

1. DAG Stalls

An instruction that loads a DAG register followed by an instruction that uses the same DAG for a memory access will cause the second instruction to stall.

```
L2= 8;
DM(I0, M0) = R1;      // Both L2 and I0 reside in DAG1,
                       // causing this instruction
                       // to stall.
```

Failure can occur if I0 points to a short-word space or if the above instruction sequence is followed by a short word access.

2. PM Memory Data Access (Cache Misses)

Any instruction that uses the PM bus to perform a data access will cause an instruction stall the first time it is executed.

```
PM(I8,M8) = R1;
```

3. Memory Block Conflicts

If an instruction requires two accesses to the same memory block, the instruction will stall.

```
DM(I0, M0)=R0, PM (I8,M8) = R1;
// A stall will occur when both address
// pointers point to the same memory block.
```

4. Wait States for External Memory Accesses

An instruction that contains an external memory access where the wait state setting for that memory bank is greater than 0 or the access is held off because of the ACK signal or through bus arbitration causes that instruction to stall.

5. Multiple Bus Accesses to IOP Registers in the Same IOP Register Group

An instruction may be stalled because of multiple buses trying to access IOP registers in the same group. For this anomaly, the only applicable case is if an external host or processor accesses the same group of registers at the same time as the core.

```
Ex. DM(MSGR0) = R1;    // Instruction executed while  
                        // the host or another processor  
                        // writes to MSGR2.
```

6. Executing Instructions from External Memory with Wait States

Any instruction being executed from external memory where the wait state setting for that memory bank is greater than 0 or the access is held off because of the ACK signal causes that instruction to be stalled.

A workaround for cases 1–5 above is to insert a NOP between the stallable instruction and the short-word access (or remove the stall condition).

Case 6 has no workaround.



Ensure that a failing sequence does not occur when you use the delayed branch (DB) option with jumps, calls, and returns. For example, ensure that the two instructions in the RTI (DB) do not cause an instruction stall in a return to code that includes short-word accesses.

Shadow Write FIFO Anomaly (ADSP-2116x Only)

For examples and workarounds, refer to anomaly 39 at the Analog Devices DSP Web site.

This anomaly has been identified in the shadow write FIFOs that exist between the internal memory array of the ADSP-21160M and core /IOP buses that access the memory. (Refer to the Hardware Reference for more details on shadow register operation.) A particular sequence of a core write followed by a read of the same internal memory address, in conjunction with a certain type of IOP activity can cause the core read to return incorrect data.

Under the circumstances described below, the Read from Addr 1 incorrectly returns the data for Addr 2.

This problem is caused by the shadow write FIFO erroneously returning data for a core read when data should have been returned from internal memory. During write operations, data is placed in the 1st stage of a two-stage shadow write FIFO. Data is moved from first to second stage when a second write is performed (by either DSP core or IOP). Similarly data is moved from the second stage of the FIFO to internal memory when neither the core nor the IOP accesses memory in a core cycle.

On read operations, address compare logic allows data to be fetched either from internal memory or from the FIFOs. Note that each memory block has one Shadow Register FIFO, and all core and IOP accesses to internal memory use this FIFO. The internal memory clock (not visible to the user) runs at twice the core clock frequency. So, each core cycle consists of two memory cycles with one of the two memory cycles dedicated to the core and the other dedicated to the IOP.

SIMD Read from Internal Memory with Shadow Write FIFO Hit Anomaly (ADSP-2116x Only)

For examples and workarounds, refer to anomaly 40 at the Analog Devices DSP Web site.

This anomaly has been identified in the Shadow Write FIFOs that exist between the internal memory array of the ADSP-21160M and core /IOP busses that access the memory. (Refer to the Hardware Reference for more details on shadow register operation.)

When performing SIMD reads that cross Long Word Address boundaries (that is, odd Normal Word addresses or non-Long Word boundary aligned Short Word addresses) and the data for the read is in the Shadow Write FIFO, the read results in Revision 0.0 behavior for the read.

How to Record a Simulator Anomaly or Event

You can record various simulator events for ADSP-21x6x DSPs.

To record a simulator anomaly event:

1. From the **Settings** menu, choose **Anomalies** or **Events**.
2. Choose the anomaly or event that you want to record. Two choices are currently available:
 - **Shadow Write**
 - **SIMD FIFO**
 - **FP Denorm**
 - **Short Word**

The **Configure Simulator Event** dialog box (Figure 3-9) appears. The anomaly or event that you want to record (for example, FP Denorm) is displayed in the title bar of the dialog box.

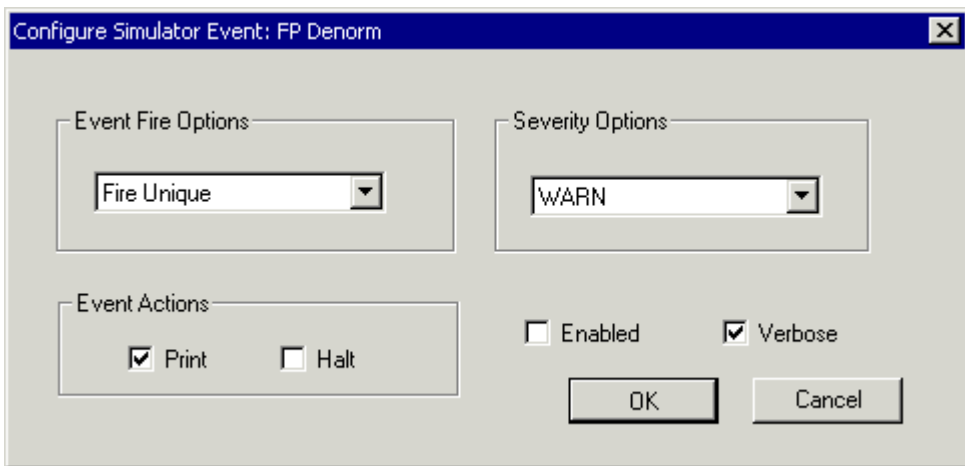


Figure 3-9. Configure Simulator Event Dialog Box

Use this dialog box to configure the simulator to handle the anomalies or events.

3. Specify options, described in [Table 3-5](#).

Table 3-5. Options in the Configure Simulator Event Dialog Box

Item	Purpose
Event Fire Options	These options specify the frequency of reporting an event. Fire Once logs the event only the first time it occurs. Fire Unique logs the event once for each unique event (a unique set of occurrences which is specific to the event type). Select this option to prevent the reporting of multiple messages for the same event. Events (for example, FP Denorm) – If a PC has had this event before, it is not unique and is not reported. Anomalies (for example, Shadow Write, SIMD FIFO, or Short Word) – If the PC at the time of the memory write and the PC at the time of the memory read taken as a pair have had this event before, it is not unique and is not reported. Fire All logs every occurrence of the event.
Severity	This option specifies the degree of the event. INFO writes a message in black typeface to the Output window. WARN writes a message in black typeface to the Output window. ERROR writes a message appears in the Output window in red typeface and rings a bell. FATAL writes a message appears to the Output window in red typeface and rings a bell.
Event Actions	Print writes messages to the Output window. Halt halts processing after the event has occurred. Using this option is similar to using a watchpoint.
Enabled	Enables this event check
Verbose	Specifies that multi-line messages are written to the Output window When this option is not selected, messages are one line long.

4. Click OK.

Clock Doubling (ADSP-21161 DSPs Only)

This section describes the crystal double mode enable pin, clock rate ratios, and CLKOUT pin configuration.

Crystal Double Mode Enable Pin

The /CLKDBL pin enables the 2x clock double circuitry. You can use the CLKDBL command to configure CLKOUT as either 1x or 2x the rate of CLKIN.

The CLKIN double circuit is primarily intended for an external crystal used with the internal clock generator and the XTAL pin. The internal clock generator, when used with the XTAL pin and an external crystal, supports an external crystal frequency up to 25 MHz.

You can use CLKDBL in XTAL mode to generate a 50-MHz input into the PLL. Enable the 2x clock mode (during RESET low) by tying **CLKDBL to GND**. Otherwise, CLKDBL is connected to VDDEXT for 1x clock mode.

For example, you can use a 25-MHz crystal to enable 100-MHz core clock rates and a 50-MHz CLKOUT operation when CLK_CFG1='0', CLK_CFG1='0', and CLKDBL='0'. You can also use this pin to generate different clock rate ratios for external clock oscillators as well.

Clock Rate Ratios

The possible clock rate ratio options (up to 100 MHz) for either CLKIN (external clock oscillator) or XTAL (crystal input) are shown in [Table 3-6 on page 3-32](#).

Simulation Options

Table 3-6. Clock Rate Ratio Options

CLKDBL	CLK_CFG 1	CLK_CFG 0	Core Clock Ratio	EP Clock Ratio
1	0	0	2:1	1x
1	0	1	3:1	1x
0	0	0	4:1	2x
0	0	1	6:1	2x
0	1	0	8:1	2x

An 8:1 ratio enables you to use a 12.5-MHz crystal to generate a 100-MHz core (instruction clock) rate and a 25-MHz CLKIN (external port) clock rate.



The maximum frequency of an external crystal cannot exceed 25 MHz. For all other external clock sources, the maximum CLKIN frequency is 50 MHz.

How to Configure the CLKOUT Pin

You can configure the DSP's CLKOUT pin to be either 1x or 2x the rate of CLKIN. A black check mark (✓) beside the **CLKDBL** command in the **Settings** menu indicates that this option is selected.

To double the clock speed, choose **CLKDBL** from the **Settings** menu.

SPI Simulation in Slave Mode

For ADSP-21161 DSPs, the external SPI is not modeled in simulation. Since the master controls the timing, there is no timing information for the transfers in slave mode. Except while booting, the BAUDR field of the slave SPICTL is used for the timing of the external master. Since this field has no effect in slave mode in the hardware, you can use it for simulation.

When the SPI is enabled in slave mode and the SPITX buffer is not empty, an SPI transmit/receive operation occurs. If the SPITX and the SPIRX buffers are empty, a word is read speculatively into SPIRX buffer. The SPI reads a maximum of one word ahead.

Once the SPIRX buffer is emptied by a DMA operation or a read, the SPI reads another word in the time specified by the BAUDR field of SPICTL.

During booting, the SPI operates at its fastest supported baud rate. When you rely on the BAUDR field in slave mode, a maximum value of 0x5 is used for this field.

The “End of file” and “File not connected” errors are suppressed until a read of SPIRX or a valid DMA is set up.

Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for details.

Boot Options

Depending on your selected DSP, the following boot options are available.

- **Boot from Host**
- **Boot from PROM**
- **Boot from Link**
- **Boot from SPI**
- **None of Above** (disables boot mode)

Simulation Options

You create a boot loader file (.LDR) based on the peripheral from which you are loading. In a simulation target session, choose a peripheral (boot option) and boot file as follows.

1. From the **Settings** menu, choose **Load Sim Loader** and a boot option (such as **Boot from Host**) to open the **Open a Boot File** dialog box.
2. Navigate to the .LDR file to boot load and select it. Then click **Open** and **OK**. A message instructs you to issue a reset instruction to execute the loader.
3. From the **Debug** menu, choose **Reset**. The simulator runs and boots in the boot-kernel (also called loader-kernel). A message instructs you to issue a run instruction, which executes the loader.
4. From the **Debug** menu, choose **Run (F5)** to execute the loader. The loader runs to completion and then displays a message indicating that the loader is finished and that it is OK to run the application.



Before clicking **Run**, you can load the symbols for the program as follows.

- a. From the **File** menu, choose **Load Symbols** to open the **Load a Processor Program's Symbols** dialog box.
 - b. Select the .DXE that was used to create the .LDR file. A message indicates that the selected symbols are loading.
5. From the **Debug** menu, choose **Run (F5)** to run the application.

For information about booting, refer to the SHARC DSP Hardware Reference for your processor or to the VisualDSP++ Linker and Utilities Manual for SHARC DSPs.

Image Viewer

The VisualDSP++ Image Viewer enables you to perform these operations:

- View an image. You can display BMP, JPEG, PPM, or MPEG data from DSP memory or from a file on your PC.
- Correct the gamma attributes of an image. For a color image, you can adjust the red, green, and blue pixel values. On a grayscale image, you can adjust darkness only.
- Copy an image to the Windows clipboard
- Print an image or save it to a file
- Export an Image

You select the image source (from DSP memory or a file on your PC) and specify image attributes. If the image is located in DSP memory, you must specify the image's address, size, and format.

For more information about Image Viewer, see [“Image Viewer” in Chapter 2, Environment](#).

Flash Programmer

The VisualDSP++ Flash Programmer provides a convenient graphical interface to on-board flash memory devices. Use the Flash Programmer to view various flash memory information, fill or erase portions of flash memory, or load an Intel hex formatted file into flash memory.

Flash Devices

Flash memory parts are non-volatile memories that can be read, programmed, and erased. In most applications, flash devices store:

- Boot code that the DSP loads at startup
- Data that must persist over time and through the loss of power

Flash device programming is typically performed with a device programmer at the factory or by the application developer. When a flash device is wired appropriately to the DSP, you can use the DSP to program the flash device.

Flash Programmer Functions

Use the Flash Programmer to:

- Load a flash algorithm (driver) program onto the DSP at any time
- View the flash manufacturer's code, device code, and part description
- Reset the flash
- Program the flash device by using an Intel Hex data file
- Fill portions of flash memory with a value and to allow quick data "punch-in"
- Erase the entire flash
- Erase a single sector
- Send custom commands to the driver for batch processes or user-defined behavior



The flash device used with the Flash Programmer must support the particular functionality.

The utility stores the most recently used information in the registry for retrieval when the utility is next started up, and a status indicator shows the utility's current state.

Flash Driver

To use the Flash Programmer, you must first load a flash driver onto the DSP. The driver is a DSP application that interfaces with the Flash Programmer and performs all the interaction with the flash device. Analog Devices supplies flash drivers (algorithms) for many EZ-KIT Lite evaluation systems. Additionally, the Flash Programmer enables you to create custom flash drivers or modify existing drivers for use on custom boards with a variety of processor and flash device combinations.

Flash Programmer Window

[Figure 3-10 on page 3-38](#) shows the Flash Programmer window.

Flash Programmer

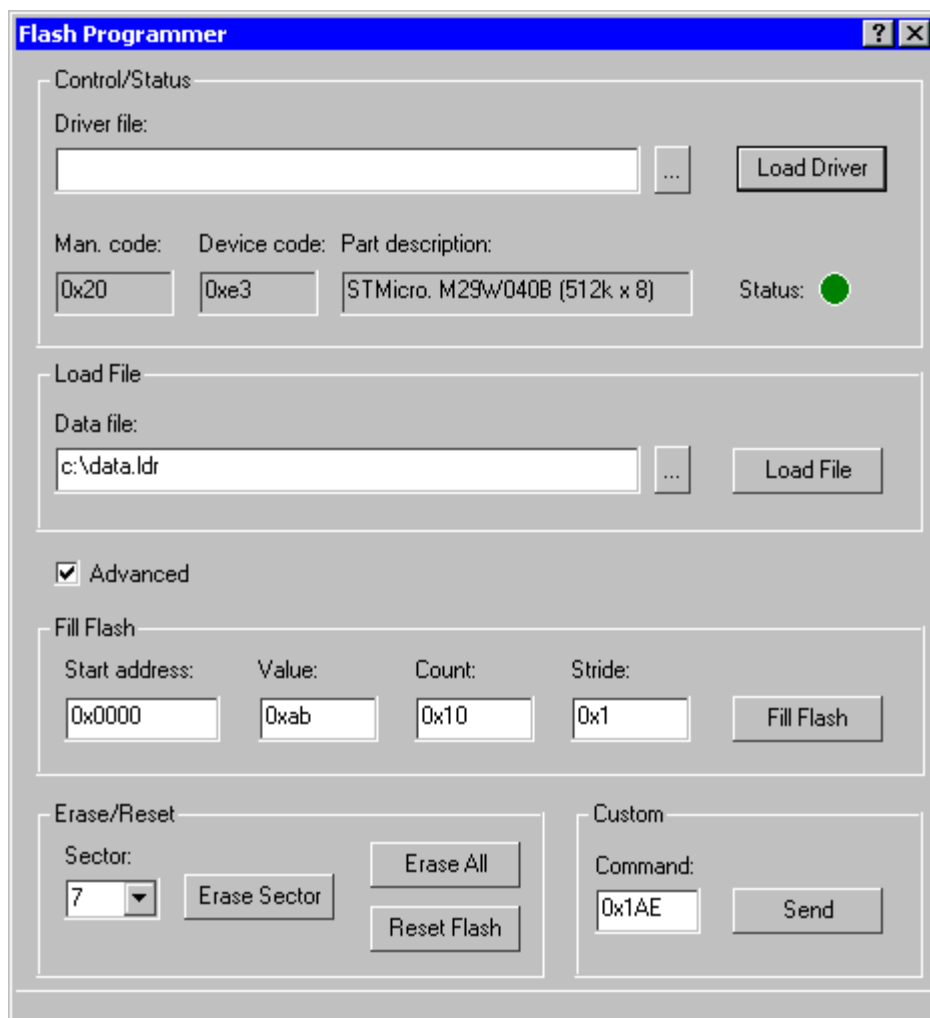


Figure 3-10. Flash Programmer Window

Table 3-7 describes the fields and buttons in the Flash Programmer window.

Table 3-7. Flash Programmer Window Controls

Control	Description
Driver File	Specifies the path and name of the driver file. Type the path and file name or browse to select the driver.
Load Driver	Loads the specified driver onto the DSP
Man. Code	Displays the flash memory's manufacturer code. You must first load the driver to view this data.
Device Code	Displays the flash memory's device code. You must first load the driver to view this data.
Part Description	Displays the flash memory's part description. You must first load the driver to view this data.
Status	Displays the utility's current status Red – The utility is not ready. You must load a driver. Green – The utility is ready to process a command. Yellow – The utility is busy processing a command.
Data File	Specifies the data file. Type the path and file name or browse to select the file. Note: Only valid Intel Hex files may be used. The VisualDSP++ loader produces files in this format.
Load File	Loads the specified data file onto the flash memory device
Advanced	Enables advanced features. When selected, the fields and buttons below it are enabled. When cleared, the fields and button are disabled (grayed out).
Start Address	Specifies the offset into the flash memory device
Value	Specifies the data value to be written

Table 3-7. Flash Programmer Window Controls (Cont'd)

Control	Description
Count	Specifies the number of locations to be written
Stride	Specifies the number of locations to skip between each write. Typically, this is 0x1. The value 0x2 specifies every other location.
Fill Flash	Loads the specified data value onto flash memory device
Sector	Specifies a single sector (or block) to be erased
Erase Sector	Erases the specified sector from the flash device
Erase All	Erases the flash device's entire memory
Reset Flash	Resets the internal state of the flash device and places it into read mode without modifying its contents
Command	Specifies the custom command to be run
Send	Sends the specified custom command to the driver. The value entered in Command is interpreted as a hexadecimal value; for example, 10 is interpreted as 10 hexadecimal or 16 decimal.

A REFERENCE INFORMATION

This appendix contains a collection of useful information to help you understand VisualDSP++ and speed up DSP program development. The information is organized as follows.

- [“Glossary” on page A-2](#)
- [“File Types” on page A-21](#)
- [“Keyboard Shortcuts” on page A-23](#)
- [“IDDE Command Line Parameters” on page A-29](#)
- [“Toolbar Buttons” on page A-30](#)
- [“Text Operations” on page A-36](#)

Glossary

The following terms are important toward understanding VisualDSP++.

Application Programming Interface (API)

A library of C/C++ functions and assembly macros that define VDK services. These services are essential for kernel-based application programs. The services include interrupt handling, thread management, and semaphore management, among other services.

Archiver

The VisualDSP++ archiver, `elfar.exe`, combines object files (`.DOJ`) into library files (`.DLB`), which serve as a reusable resource for project development. The linker searches library files for routines (library members) that are referred to by other objects, and links them in your executable program.

Breakpoint

User-defined halt in an executable program. Toggle breakpoints (turn them on or off) by double-clicking on a location in a **Disassembly** window or editor window.

Break condition

Hardware condition under which the target breaks and returns control of the target back to the user. For example, a break condition could be set up to occur when address `0x8000` is read from or written to.

Build

Performing a build (or project build) refers to the operations (pre-processing, assembling, and linking) that VisualDSP++ performs on projects and files. During a build, VisualDSP++ processes the files in your project that have been modified (or depend on files that have been modified) since the previous build. A build differs from a rebuild all. During a rebuild all, VisualDSP++ processes all the files in the project, regardless whether they have been modified.

Build type

Replaced by “configuration.”

Channel

A FIFO queue into which messages sent to a thread are placed. Each thread has 15 channels with messages being received in priority order from the lowest numbered channel to the highest.

Configuration

(or project configuration) You develop a project in stages (configurations). By default, a project includes two configurations: Debug and Release. A configuration refers to the collection of options (tool chain and individual options for files) specified for the configuration. You can add a configuration to your project at any time. You can delete a customized configuration that you created, but you cannot delete the Debug or Release configurations.

Context switch

A process of saving/restoring the processor’s state. The scheduler performs the context switch in response to the system change.

A hardware interrupt can occur and change the state of the system at any time. Once the processor’s state has changed, the currently running thread may be swapped with a higher-priority thread.

Glossary

When the kernel switches threads, the entire processor's state is saved and the processor's state for the thread being switched in is restored. This process is known as a context switch.

Critical region

A sequence of instructions whose execution cannot be interrupted or swapped out. Suspending all interrupt service routines (ISRs) before calling the critical region ensures that the execution of a critical region is not interrupted. Once the critical region routine has been completed, ISRs are enabled.

Current directory

Directory in which the `.DPJ` file is saved. The build tools use the current directory for all relative file path searches. See also "Default directories."

Data set

A series of data values in DSP memory used as input to a plot. You can create data sets and configure the data for each data set. You specify the memory location, the number of values, and other options that identify the data. 3-D plots require additional specifications for row and column counts.

Debug configuration

For a debug configuration, you can accept the default options, or you can specify the options you want and save them. The configuration refers to the specified options for all the tools in the tool chain. See also "Configuration."

Debug session

The combination of a target and a platform. For example, a session can be a JTAG emulator target connected to a platform consisting of five ADSP-21065Ls. Another example of a debug session is an ADSP-2192-12 EZ-KIT Lite target connected to an ADSP-2192-12 EZ-KIT Lite board.

The DSP projects you develop are run as debug sessions. The two types of sessions are hardware and software. The processor, target, and platform define the session. When you set up a session, you set the focus on a series of more specific elements.

Debug target

The communication channel between VisualDSP++ and a DSP (or group of DSPs). Targets include simulators, emulators, and EZ-KIT Lite evaluation systems. Several targets may be installed on your system. Simulator targets, such as the ADSP-TS101 Cycle Accurate SHARC Simulator, differ from emulator targets in that the DSP exists only in software.

The Summit-ICE emulator communicates with one or more physical devices over the host PC's PCI bus. The Apex-ICE™ emulator communicates with a device through the PC's USB port.

Default directories

These intermediate and output file directories (folders) are `\Debug` (for the debug configuration) and `\Release` (for the release configuration). By default, VisualDSP++ creates these directories as children of the directory in which the `.DPJ` file is saved, which is called the project's current directory. See also "Current directory."

Glossary

Dependencies

VisualDSP++ uses dependency information to determine which files, if any, are updated during a build. If an included header file is modified, VisualDSP++ builds the source files that include (`#include`) the header file, regardless of whether the source files have been modified since the previous build.

Dependency files

Usually user files or system header (`*.h`) files, these files are referenced from a source file by a preprocessor `#include` command.

Device

A single DSP. With regard to JTAG emulation and the JTAG EZ-ICE Configurator, a device refers to any physical chip in the JTAG chain.

Device driver

A user-written model that abstracts the hardware implementation from the application code. User code accesses device drivers through a set of device driver APIs.

DWARF-2

Format for debugging source-level assembly code via improved line and symbol information

Editor window

A document window that displays a source file for editing. When an editor window is active, you can move about within the window and perform typical text editing activities such as searching, replacing, copying, cutting, pasting, and so on.

ELF

Executable and Linking Format

Emulator

Hardware used to connect a PC to a DSP target board to allow application software to be downloaded and debugged from within the VisualDSP++ environment. Emulator software performs the communications that enable you to see how your DSP code affects DSP performance.

Event

A signal (similar to a semaphore or message) used to synchronize multiple threads in a system. An event is a logical switch, having two binary states (available/true and unavailable/false) that control thread execution. When an event becomes available, all pending (waiting) threads in the wait list are set to be ready-to-run. When an event is available and a thread pends on it, the thread continues running and the event remains available.

To facilitate error handling, threads can specify a timeout period when pending on an event.

An event is a code object of global scope, so any thread can pend on any event. Event properties include the EventBit mask, EventBit value, and combination type. Events are statically allocated and enumerated at runtime. An event cannot be destroyed, but its properties can be changed (see Blueelem text).

Glossary

Event bit

A flag set or cleared to post the event. The event is posted (available) when the current values of the system Event Bits match the event bit's mask and event bits' values defined by the event's combination type.

A system has one and only one Event Bits word, the size of a data word minus one: fifteen bits for ADSP-219x DSPs; thirty-one bits for ADSP-21xxx, ADSP-2153x, and ADSP-TSxxx DSPs.

Executable file

A file or program that has been written and built in VisualDSP++

Focus

Refers to the active processor in an MP session that you are debugging

Interrupts

An external or internal condition detected by the hardware interrupt controller. In response to an interrupt, the kernel processes a subroutine call to a predefined Interrupt Service Routine (ISR).

Interrupts have the following specifications.

Latency – interrupt disable time. The period between the interrupt occurrence and the first ISR's executed instruction.

Response – interrupt response time. The period between the interrupt occurrence and a context switch.

Recovery – interrupt recovery time. The period needed to restore the processor's context and to start the return-from-interrupt (RTI) routine.

Interrupt Service Routine (ISR)

A routine executed as a response to a software interrupt or hardware interrupt. VDK supports nested interrupts, which means that the kernel recognizes other interrupts, services interrupts, or both with higher priorities while executing the current ISR. VDK ISRs are written in assembly language. VDK reserves the timer and the lowest priority (reschedule) interrupt.

JTAG

Joint Test Action Group. This committee is responsible for implementing the IEEE boundary scan specification, enabling in-circuit emulation of ICs.

kernel

The main module of a real-time operating system. The kernel loads first and permanently resides in the main memory and manages other modules of the real-time operation system. Typical services include context switching and communication management between OS modules.

Keyboard shortcuts

The keyboard provides a quick means of running the commands that are used most often, such as simultaneously typing the keyboard's **Ctrl** and **G** keys (indicated with the symbols **Ctrl+G**) to go to a line in a file.

Librarian

A utility that groups object files into library files. When you link your program, you can specify a library file and the linker automatically links any file in the library that contains a label used in your program. Source code is provided so you can adapt the routines to your needs.

Library files

The VisualDSP++ archiver, `elfar.exe`, combines object files (`.DOJ`) into library files (`.DLB`), which serve as a reusable resource for project development. The linker searches library files for routines (library members) that are referred to from other objects, and links them into your executable program.

Linear profiling

A debugging feature that samples the target's PC register at every instruction cycle. Linear profiling gives an accurate picture of where instructions were executed, since every PC value is collected. The trade-off, however, is that linear profiling is much slower than statistical profiling. A display of the resulting samples appears in the **Linear Profiling Results** window, which graphically indicates where the application is spending its time. Simulator targets support linear profiling. See also "Statistical profiling."

Linker

The linker creates executable files, shared memory files, and overlay files from separately assembled object and library files. It assigns memory locations to code and data in accordance with a user-defined `.LDF` file, which describes the memory configuration of the target system.

Loader

A utility that transforms an executable file into a boot file. The loader creates a small kernel, which is booted into internal memory at chip reset to enable a program of arbitrary size to be loaded into the DSP's internal and external memory.

Makefile

VisualDSP++ can export a makefile (make rule file), based on your project options. Use a makefile (.MAK) to automate builds outside of VisualDSP++. The output make rule is compatible with the gnu-make utility (GNU Make V3.77 or higher) or other make utilities.

Memory pool

An area of memory containing a specified number of uniformly sized blocks of memory available for allocation and subsequent use in an application. The number and size of the blocks in a particular memory pool are defined at pool creation.

Message

A signal (similar to an event or semaphore) used to synchronize two threads in a system or to communicate information between threads. A message is sent to a specified channel on the recipient thread (and can optionally pass a reference to a payload to facilitate the transfer of data between threads). Posting a message takes a deterministic amount of time and may incur a context switch.

Mixed mode

One of the two editor window display formats (the other being source mode). Mixed mode displays assembled code after the line of the corresponding C code.

Multiprocessor system

A system built with multiple DSPs. Often, performance-based products require two or more DSPs. A system built with a single DSP is called a *single-processor system*. Debugging a multiprocessor system requires that you synchronously run, step, halt, and observe program execution operations in all the processors at once. The SHARC simulator does not support this capability.

Outdated file

A file that has been edited since the last time it was built

Payload

An arbitrary amount of data associated with a message. A reference to the payload can be passed between threads as part of a message to enable the recipient thread to access the data buffer that contains the payload.

Pinning a window

A technique that statically associates a window to a specific processor

Platform

A configuration of DSPs with which a target communicates. For simulation, a platform is typically one or more DSPs of the same type. For emulation, you specify the platform using the JTAG EZ-ICE Configurator, and the platform can be any combination of devices.

The platform represents the hardware upon which one or more devices reside. You typically define a platform for a particular target. For example, if three emulators are installed on your system, a platform selection might be emulator two.

Several platforms may exist for a given debug target. For a simulator, the platform defaults to the identical DSP simulator. When the debug target is a JTAG emulator, the platforms are the individual JTAG chains. When the debug target is an EZ-ICE board, the platform is the board in the system on which you wish to focus.

Preemptive kernel

A priority-based kernel in which the currently running thread of the highest priority is preempted, or suspended, to give system resources to the new highest-priority thread.

Processor

An individual chip contained on a specific platform within a target. When you create the executable file, the processor is specified in the Linker Description File (.LDF) and other source files.

Project

This term refers to the collection of source files and tool configurations used to create a DSP program. Through a project, you can add source files, define dependencies, and specify build options related to producing your output executable program. A project file (.DPJ) stores your program's build information.

VisualDSP++ enables you to manage projects from start to finish in an integrated user interface. Within the context of a DSP project, you define project and tool configurations, specify project-wide and individual file options for debug or release modes of project builds, and create source files. VisualDSP++ facilitates easy movement among editing, building, and debugging activities.

Project configuration

This configuration includes all of the settings (options) for the tools used to build a project.

Project file tree

See “Project window.”

Project window

This window displays your project's files in a "tree view," which can include folders to organize your project files. Right-clicking on an icon (the project itself, a folder, or a file) opens a menu, providing actions you can perform on the selected item. Double-clicking on the project icon or a folder icon opens or closes the tree list. Double-clicking a file icon opens the file in an editor window.

Real-time operating system (RTOS)

A software executive that handles DSP algorithms, peripherals, and control logic. The RTOS comprises the following components: kernel, communication manager, support library, and device drivers. An RTOS enables structured, scalable, and expandable DSP application development while hiding OS complexity.

Rebuild all

See "Build."

Registers

For information on available registers, see the corresponding processor documentation or view the associated online Help.

Release configuration

You can accept the default set of options, or you can specify the options you want and save them. The configuration refers to the specified options for all the tools in the tool chain. See also "Configuration."

Reset

This command resets the processor to a known state, clearing processor memory.

Restart

This command sets your program to the first address of the interrupt vector table. Unlike a reset, you do not need to reload memory.

Right-click

This action opens a right-click menu (sometimes called a context menu, pop-up menu, or shortcut menu). The commands that appear depend on the context (what you are doing). Right-click menus provide access to many commonly used commands.

Round-robin scheduling

A scheduling scheme whereby all threads at a given priority are given processor time automatically in fixed duration intervals. Round-robin priorities are specified at build time.

Scheduler

A kernel component responsible for scheduling system threads and interrupt service routines. VDK is a priority-based kernel in which the highest-priority thread is executed first.

Semaphore

A signal (similar to an event or message) used to synchronize multiple threads in a system. A semaphore is a data object whose value is zero or a positive integer (limited by the maximum set up at creation time). The two states (available/greater than zero and unavailable/zero) control thread execution. Unlike an event, whose

Glossary

state is automatically calculated, a semaphore is directly manipulated. Posting a semaphore takes a deterministic amount of time and may incur a context switch.

Serial port data

You can automatically transfer serial port (SPORT) data to and from on-chip memory using DMA block transfers. Each serial port offers a time division multiplexed (TDM) multichannel mode.

Session

See “Debug session.”

Session name

Although the choice of target, platform, and processor define the session, you may want to further identify the session. You can modify the default session name when you first create the debug session to prevent confusion later. A session name can be any string and can include space characters. There is no limit to the number of characters in a session name, but the **Session List** dialog box can display about 32 characters.

Shortcuts

See “Keyboard shortcuts.”

Signal

A method of communicating between multiple threads. VDK supports four types of signals: semaphores, events, messages, and device flags.

Simulator

The simulator is software that mimics the behavior of a DSP chip. Simulators are often used to test and debug DSP code before the DSP chip is manufactured.

The simulator runs an executable program in software similar to the way a processor does in hardware. The simulator also simulates the memory and I/O devices specified in the `.LDF` file. VisualDSP++ lets you interactively observe and alter the data in the processor and in memory. The simulator reads executable files. A simulator's response time is slower than that of an emulator.

Source files

The C/C++ language and assembly language files that make up your project. Other source files that a project uses, such as the `.LDF` file, contain command input for the linker, and dependency files (data files and header files). View source files in editor windows.

Source mode

One of the two editor window display formats (the other being mixed mode). Source mode displays C code only.

Glossary

Statistical profiling

A debugging feature that provides a more generalized form of profiling that is well suited to JTAG emulator debug targets. With statistical profiling, VisualDSP++ randomly samples the target processor's program counter (PC) and presents a graphical display of the resulting samples in the **Statistical Profiling Results** window. This window graphically indicates where the application is spending time.

JTAG sampling is completely non-intrusive so the process does not incur additional runtime overhead. See also "Linear Profiling."

Stepping

A technique for moving through source or assembly code to observe instruction execution

Symbols

Labels for sections, subroutines, variables, data buffers, constants, or port names. For more information, refer to the related build tool documentation.

System configurator

The system configuration control is accessible from the **Kernel** page of the **Project** window. The **Kernel** page provides a graphical representation of the data contained in the `vdk.h` and `vdk.cpp` files.

Target

See "Debug target."

Tcl Scripting

VisualDSP++ includes an interpreter for the Tcl (Tool Command Language) scripting language. Analog Devices has extended Tcl version 8.3 with several procedures to access key debugging features. The power of the Tcl language, coupled with Analog Devices extensions, allows you to extensively script your work. Tcl command output displays in the **Output** window's **Console** page. The output is also logged to `VisualDSP_log.txt`.

Threads

A kernel system component that performs a predetermined function and has its own share of system resources. VDK supports multithreading, a run-time environment with concurrently executed independent threads.

Threads are dynamic objects that can be created and destroyed at runtime. Thread objects can be implemented in C, C++, or assembly language. A thread's properties include an ID, priority, and current state (wait, ready, run, or interrupted). Each thread maintains its own C/C++ stack.

Ticks

The system level timing mechanism. Every system tick is a timer interrupt.

Tool chain

The collection of tools (utilities) used to build a project configuration

Trace

Provides a history of program execution. A trace is sometimes called an execution trace or a program trace. Trace results show how the program arrived at a certain point and show program reads, writes, and memory fetches. SHARC DSPs do not support traces.

Unscheduled regions

A sequence of instructions whose execution can be interrupted, but cannot be swapped out. The kernel acknowledges and services interrupts when an unscheduled region routine is running.

VisualDSP++

An Integrated Development and Debugging Environment (IDDE) for Analog Devices DSP development tools.

VisualDSP++ Kernel (VDK)

RTOS kernel from Analog Devices. VDK is part of VisualDSP++. The kernel is integrated with the Integrated Development and Debugging Environment (IDDE), assembler, compiler, and linker programs into the DSP development tool chain.

The VDK is supported on the ADSP-219x, ADSP-21xxx, ADSP-TSxxx, and Blackfin processors. Refer to the *VisualDSP++ Kernel (VDK) User's Guide* for details.

Watchpoints

For simulation only. Similar to breakpoints, watchpoints stop program execution. Watchpoints, however, allow you to set up conditions, such as a memory read or stack pop. Unlike breakpoints, watchpoints are not attached to a specific address. The program halts when a watchpoint's conditions are met. SHARC DSPs do not support watchpoints.

Workspace

You can open multiple windows and place them anywhere you want. After you open and arrange your windows, you can save the layout (configuration) as a workspace setting, which you can recall (load) at a later time. Each debug session's default workspace is automatically saved when you close the debug session and is automatically restored when you load that session.

File Types

Table A-1 describes the files used to build a project.

Table A-1. Files Used with VisualDSP++

Extension	Name	Purpose
.ASM	Assembly source file	Source file comprising assembly language instructions
.C	C source file	Source file comprising ANSI standard C code and Analog Devices extensions
.CPP .CXX .HPP .HXX	C++ source file	Preprocessed compiler files that are inputs to the C/C++ compiler. These files comprise ANSI standard C++ code.
.DPJ	Project file	Contains a description of how your source files combine to build an executable program
.LDF	Linker Description File	This linker command source file is a text file that contains commands for the linker in the linker's scripting language
.S .PP .IS	Intermediate files	Preprocessed assembly files generated by the preprocessor
.DOJ	Assembler Object file	Binary output of the assembler

Table A-1. Files Used with VisualDSP++ (Cont'd)

Extension	Name	Purpose
.DLB	Archiver file	The archiver's binary output in ELF format
.H	Header file	A dependency file used by the preprocessor, and a source file for the assembler and compiler
.DAT	Data file	A dependency file used by the assembler for data initialization
.DXE .SM .OVL .DLO	Debugging files	Binary output files from the linker in ELF/DWARF format
.MAP	Linker Memory Map file	An optional output for the linker. This text file contains memory and symbol information for executable files.
.TCL	Tool Command Language file	A Tcl scripting language file used to script work
.OBJ	Assembled Object file	(Previous releases only, replaced by .DOJ) The output of the assembler
.LST	Listing file	An optional file output by the assembler
.ACH	Architecture file	(Previous releases only, replaced by .LDF)
.TXT	Linker Command-Line file	(Previous releases only, replaced by .LDF) This ASCII text file contains command line input for the linker
.EXE	Debugging file	(Used in previous releases, replaced by .DXE)
.VDK	VisualDSP++ Kernel Support file	Enables VDK support

Keyboard Shortcuts

VisualDSP++ includes keyboard shortcuts (also called shortcut keys) for the operations that you use most often. These keyboard shortcuts appear in the tables below. You can also run commands by:

- Choosing a command from a drop-down menu on the menu bar
- Clicking a toolbar button
- Right-clicking from a particular context, such as from the **Project** window
- Clicking a configured user tool 
- Clicking a button within a dialog box
- Running a Tcl script (from the **File** menu or **Output window**)
- Choosing a command from the application's control menu

Working with Files

When working with files, use the keyboard shortcuts listed in [Table A-2](#).

Table A-2. Keyboard Shortcuts for Working with Files

Action	Key(s)
Open a new file	Ctrl+N
Open an existing file	Ctrl+O
Save a file	Ctrl+S
Print a file	Ctrl+P
Go to the next window	F6
Go to the previous window	Shift+F6

Moving within a File

To move within a file, use the keyboard shortcuts listed in [Table A-3](#).

Table A-3. Keyboard Shortcuts for Moving within a File

Action	Key(s)
Move the cursor to the left one character	Left Arrow (←)
Move the cursor to the right one character	Right Arrow (→)
Move the cursor to the beginning of the file	Ctrl+Home
Move the cursor to the end of the file	Ctrl+End
Move the cursor to the beginning of the line	Home
Move the cursor to the end of the line	End
Move the cursor down one line	Down Arrow (↓)
Move the cursor up one line	Up Arrow (↑)
Move the cursor one page down	Page Down
Move the cursor one page up	Page Up
Move the cursor right one tab	Shift
Move the cursor left one tab	Shift+Tab
Move the cursor left one word	Ctrl+Left Arrow (←)
Move the cursor right one word	Ctrl+Right Arrow (→)
Move to the matching brace character within a file	Ctrl+B
Go to the next bookmark	F2
Go to a line	Ctrl+G
Find text	Ctrl+F
Find the next occurrence of text	F3

Cutting, Copying, Pasting, Moving Text

To edit text, use the keyboard shortcuts listed in [Table A-4](#).

Table A-4. Keyboard Shortcuts for Editing Text

Action	Key(s)
Copy text	Ctrl+C or Ctrl+Insert
Copy text	Select with cursor and Ctrl+drag
Cut text	Ctrl+X or Shift+Delete
Delete text	Delete (selection or forward)
Delete text	Backspace (selection or backward)
Move text	Select with cursor and drag
Move selected text right one tab	Tab
Move selected text left one tab	Shift+Tab
Paste text	Ctrl+V or Shift+Insert
Undo the last edit	Ctrl+Z or Alt+Backspace
Redo an edit command	Shift+Ctrl+Z
Replace text	Ctrl+H or Ctrl+R

Selecting Text within a File

To select text within a file, use the keyboard shortcuts listed in [Table A-5](#).

Table A-5. Selecting text within a File

Action	Key(s)
Select all text in a file	Ctrl+A
Select the character on the left	Shift+Left Arrow (←)
Select the character on the right	Shift+Right Arrow (→)

Keyboard Shortcuts

Table A-5. Selecting text within a File (Cont'd)

Action	Key(s)
Select all text to the beginning of the file	Shift+Ctrl+Home
Select all text to the end of the file	Shift+Ctrl+End
Select all text to the beginning of the line	Shift+Home
Select all text to the end of the line	Shift+End
Select all text to the line below	Shift+Down Arrow (↓)
Select all text to the line above	Shift+Up Arrow (↑)
Select all text to the next page	Shift+PgDn
Select all text to the above page	Shift+PgUp
Select the word on the left	Shift+Ctrl+Left Arrow (←)
Select the word on the right	Shift+Ctrl+Right Arrow (→)
Select by column	Place cursor, press and hold down Alt and drag the cursor (selects by column-character instead of by line-character)

Working with Bookmarks in an Editor Window

When working with bookmarks in an editor window, use the keyboard shortcuts listed in [Table A-6](#).

Table A-6. Keyboard Shortcuts for Bookmarks

Action	Key(s)
Toggle a bookmark	Ctrl+F2
Go to next bookmark	F2

Building Projects

To build projects, use the keyboard shortcuts listed in [Table A-7](#).

Table A-7. Keyboard Shortcuts for Building Projects

Action	Key(s)
Build the current project	F7
Build only the current source file	Ctrl+F7

Using Keyboard Shortcuts for Program Execution

For program execution, use the keyboard shortcuts listed in [Table A-8](#).

Table A-8. Keyboard Shortcuts for Program Execution

Action	Key(s)
Load a program	Ctrl+L
Reload a program	Ctrl+R
Dump to file	Ctrl+D
Run	F5
Multiprocessor run	Ctrl+F5
Run to cursor	Ctrl+F10
Halt	Shift+F5
Step over	F10
Step into	F11
Multiprocessor step	Ctrl+F11
Step out of	Alt+F11
Halt a Tcl script	Ctrl+H

Working with Breakpoints

When working with breakpoints, use the keyboard shortcuts listed in [Table A-9](#).

Table A-9. Keyboard Shortcuts for Breakpoints

Action	Key(s)
Open the Breakpoints dialog box	Alt+F9
Enable/disable a breakpoint	Ctrl+F9
Toggle (add or remove) a breakpoint	F9

Working with Multiprocessor Systems

Use the keyboard shortcuts listed in [Table A-10](#) when working with multiprocessor systems.

Table A-10. Keyboard Shortcuts for Multiprocessing

Action	Key(s)
Move focus to the next processor in the processor list	Ctrl+Up Arrow (!)
Move focus to the previous processor in the processor list	Ctrl+Down Arrow (Ø)
Move focus to the Nth processor in the processor list	Ctrl+number key (1–9), for example Ctrl+3

 The SHARC simulator does not support multiprocessor debugging.

Obtaining Online Help

To obtain online Help, use the keyboard shortcuts listed in [Table A-11](#).

Table A-11. Keyboard Shortcuts for Obtaining Online Help

Action	Key(s)
View online Help for the selected object	F1
Obtain context-sensitive Help for controls (buttons, fields, menu items)	Shift+F1

Miscellaneous

For windows and workspaces, use the keyboard shortcuts listed in [Table A-12](#).

Table A-12. Miscellaneous Keyboard Shortcuts

Action	Key(s)
Refresh all windows	F12
Select workspace 1 through 10	Alt+1 ... Alt+0

IDDE Command Line Parameters

You can invoke VisualDSP++ from a DOS command line.

Syntax:

```
idde.exe [-f script_name]
         [-s session_name]
         [-p project_name]
```



Specify the full path to `idde.exe`.

Toolbar Buttons

[Table A-13](#) describes the `idde.exe` command line parameters.

Table A-13. `idde.exe` Command Line Parameters

Item	Description
<code>-f script_name</code>	Loads and executes the Tcl script specified by <code>script_name</code> . Use this parameter to automate regression tests. You can also manipulate VisualDSP++ by running a Tcl script from a library of common Tcl commands that you create. If an error is encountered while executing this script, VisualDSP++ automatically exits.
<code>-s session_name</code>	Specifies the session to which VisualDSP++ connects when it starts. The session must already exist. This parameter is useful when you are debugging more than one target board. Having multiple shortcuts to <code>idde.exe</code> allows you to run a different session. This overrides VisualDSP++'s default behavior of always connecting to the last session.
<code>-p project_name</code>	Specifies the project to load at startup. The project must already exist.

Examples:

```
idde.exe -f "c:\\scripts\\myscript.tcl"
```

```
idde.exe -s "My 21160 JTAG Emulator Session"
```

```
idde.exe -p "c:\\projects\\myproject.dpj"
```

Toolbar Buttons

The toolbar, which comprises separate toolbars, provides quick mouse access to commands.

The toolbar is a Windows docking bar. You can move it to different areas of the screen by dragging it to the selected location.

[Table A-14 on page A-31](#) describes the toolbar buttons.

Table A-14. Toolbar Buttons

Button	Purpose
	Creates a new document
	Opens an existing document
	Saves the active document or template with the same name
	Prints the active document
	Loads a program into the target
	Reloads the most recent program into the target
	Cuts selected data from the document and store it on the clipboard
	Copies the selection to the clipboard
	Pastes the contents of the clipboard at the insertion point
	Undoes previous edit command (multilevel undo)
	Redoes the command undone by the previous Undo command (multilevel redo)
	Finds a text block in an editor window

Toolbar Buttons

Table A-14. Toolbar Buttons (Cont'd)

Button	Purpose
	Finds again or repeats the previous find command
	Replaces the selected text with other text
	Searches through files for text or regular expressions
	Goes to or moves to the specified location
	Displays the current source file
	Toggles the bookmark at selected line in the active editor window
	Goes to the next bookmarked line in the editor window
	Goes to the previous bookmarked line in the editor window
	Clears all bookmarks in the editor window
	Opens the online Help to the Search page
	Provides context-sensitive Help for a button command or portion of VisualDSP++
	Opens the About VisualDSP++ dialog box

Table A-14. Toolbar Buttons (Cont'd)

Button	Purpose
	Adds a source file to the project
	Removes a selection from the project
	Opens an existing project
	Saves the open project
	Opens the Project Options dialog box, where you specify project options
	Builds the selected source file
	Builds the project (update outdated files)
	Builds all files in the project
	Stops the current project build
	Arranges windows as tall non-overlapping tiles
	Arranges windows as wide non-overlapping tiles
	Arranges windows so they overlap

Toolbar Buttons

Table A-14. Toolbar Buttons (Cont'd)

Button	Purpose
	Closes all open windows
	Refreshes all the debugging windows
	Runs (starts or continues) the current program
	Restarts the current program
	Stops the current program
	Resets the target
	Toggles a breakpoint for the current line
	Clears all current breakpoints
	Enables or disables one breakpoint
	Disables all breakpoints
	Steps one line
	Steps over the current statement

Table A-14. Toolbar Buttons (Cont'd)

Button	Purpose
	Steps out of the current function
	Runs the program to the line containing the cursor
	Opens the Expressions window
	Opens the Locals window
	Opens the Call Stack window
	Opens the Disassembly window
	Runs a multiprocessor group
	Restarts a multiprocessor group
	Halts a multiprocessor group
	Steps a multiprocessor group by one assembly instruction
	Runs the command associated with the user tool (one of ten)
	Opens the associated workspace (one of ten)

Text Operations

VisualDSP++ allows you to use regular expressions and tagged expressions in find/replace operations and comments in your code.

Regular Expressions vs. Normal Searches

Normally, when you search for text, the search mechanism scans for an exact, character-by-character match of the search string, which does not have to be an entire word. Every character in the search string is examined. If there are embedded spaces, for instance, the exact number is matched.

Regular expression matching provides much more flexibility and power than a normal search. A regular expression can be a simple string, which yields the same matches as normal searches. Some characters in a regular expression string, however, have special interpretations, which provide greater flexibility.

For example, with regular expression matching, you can find the following.

- All occurrences of either `hot` or `cold`
- Occurrences of `for` followed by a left parenthesis, with any number of intervening spaces
- A `;` (semicolon) only when it is the last character on a line
- The string `ADSP` followed by a sequence of digits

You can use a regular expression as the search pattern for replacement. In that case, there are ways to identify and recover the variable portions of the matched strings.

Specific Special Characters

Regular expressions assign special meaning to the following characters.

If you need to match on one of these characters, you must escape it by preceding it with a backslash (\). Thus, \^ matches the ^ character, yet ^ matches the beginning of the line.

Table A-15. Special Search Characters

Character	Description
^	A caret matches the beginning of the line
\$	A dollar sign matches the end of the line
.	A period (.) matches any character
[abc]	A bracketed sequence of characters matches one character, which may be any of the characters inside the brackets. Thus, [abc] matches an a, b, or c.
[0-9]	This shorthand form is valid within the sequence brackets. It specifies a range of characters, from first through last, exactly as if they had been written explicitly. Ranges may be combined with explicit single characters and other ranges within the sequence. Thus, [-+.0-9] matches any constituent character of a signed decimal number; and [a-zA-Z0-9_] matches a valid identifier character, either lowercase or uppercase. Ranges follow the ordering of the ASCII character set.
[^abc] [^0-9]	A caret (^) that is the first character of a sequence matches all characters except for the characters specified after the caret.
(<i>material</i>)	The material inside the parentheses can be any regular expression. It is treated as a unit, which can be used in combination with other expressions. Parenthesized material is also assigned a numerical tag, which may be referenced by a replace operation.

Special Rules for Sequences

The normal special character rules of regular expressions do not apply within a bracketed sequence. Thus, [`*&`] matches an asterisk or ampersand.

Certain characters have special meaning within a sequence. These include `^` (not), `-` (range), and `]` (end of sequence). By placing these characters appropriately, you can specify these characters to be part of the sequence.

To search for a right bracket character, place `]` as the first character of the search string. To search for a hyphen character, place `-` as the first character of the search string after `]`, if present. Place a caret anywhere in the search string except at the front, where it means “not.”

Repetition and Combination Characters

The characters described in [Table A-16](#) extend the meaning of the immediately preceding item. This item may be a single character, a sequence in braces, or an entire regular expression in parentheses.

Table A-16. Match Characters

Character	Description
*	An asterisk matches the preceding any number of times, including none at all. Thus, <code>ap*le</code> matches <code>apple</code> , <code>aple</code> , <code>apppple</code> and <code>ale</code> . For example, <code>^ *void</code> matches only when <code>void</code> occurs at the beginning of a line and is preceded by zero or more spaces.
+	A plus character matches the preceding any number of times, but at least one time. Thus, <code>ap+le</code> matches <code>apple</code> and <code>aple</code> , but does not match <code>ale</code> .

Table A-16. Match Characters (Cont'd)

Character	Description
?	A question mark matches the preceding either zero or one time, but not more. Thus, ap?le matches ale and aple, but nothing else.
	The pipe character () matches either the preceding or following item. For example, (hot) (cold) matches either hot or cold. Note: Spaces are characters. Thus, (hot) (cold) matches “hot “or” cold”.

Match Rules

If multiple matches are possible, the *, +, and ? characters match the longest candidates. The | character matches the left-hand alternative first.

For more information, see the many reference texts available on this topic, such as *Mastering Regular Expressions*, *Powerful Techniques for Perl and Other Tools* by Jeffrey E. F. Friedl, (c) 1997 O'Reilly & Associates, Inc.

Tagged Expressions in Replace Operations

Use a tagged expression as part of the string in the **Replace** field for a replace operation.

You must enclose a tagged expression between parentheses characters.

In the **Replace** field, the operators in [Table A-17 on page A-40](#) represent tagged expressions from the **Find** field.

Text Operations

Table A-17. Using Tagged Expressions in Replace Operations

Find field	Replace field
Entire matched sub string	\0
Tagged expressions within parentheses () from left to right	\1 \2 \3 \4 \5 \6 \7 \8 \9
Entire match expression	&

The replace expression can specify an ampersand (&) character, meaning that the & represents the substring that was found. For example, if the substring that matched the regular expression is “abcd”, a replace expression of “xyz&xyz” changes it to “xyzabcdxyz”. The replace expression can also be expressed as “xyz\0xyz”, where the “\0” indicates a tagged expression representing the entire substring that was matched. Similarly, you can have another tagged expression represented by “\1”, “\2”.



Although the tagged expression 0 is always defined, the tagged expressions 1, 2, and so on, are defined only when the regular expression used in the search has enough sets of parenthesis. Some examples are shown in [Table A-18](#).

Table A-18. Examples of Replace Operations

String	Search	Replace	Result
Mr.	(Mr)(\.)	\1s\2	Mrs.
abc	(a)b(c)	&\1-\2	abc-a-c
bcd	(a b)c*d	&\1	bcd-b
abcde	(.*)c(.*)	&\1-\2	abcde-ab-de
cde	(ab cd)e	&\1	cde-cd

Comment Start and Stop Strings

You use start comment strings and stop comment strings for comment highlighting colors. [Table A-19](#) and [Table A-20](#) describe the two types of comment strings that you can set for each file type.

Table A-19. Start Comment Strings

String	Purpose
!	Starts an assembly style, single-line comment
/*	Starts a C/C++ style, multi-line comment
//	Starts a C/C++ style, single-line comment

Table A-20. Stop Comment Strings

String	Purpose
Carriage return	Ends a single-line comment (C and Assembly)
*/	Ends a C/C++ style, multi-line comment
(blank)	Ends a C/C++ style, single-line comment

B TCL SCRIPTING

This appendix describes how you can use Tcl scripting to perform repeated sequences of debugging operations. The information is organized under the following topics.

- [“Overview of Tcl Scripting” on page B-1](#)
- [“Tcl Command Issuance” on page B-3](#)
- [“Examples of Tcl Scripts” on page B-6](#)
- [“Types of Tcl Commands” on page B-14](#)
- [“Tcl Command Reference” on page B-19](#)

Overview of Tcl Scripting

VisualDSP++ includes an interpreter for the Tool Command Language (Tcl) scripting language. This well-documented C-like language, developed by UC Berkeley researchers, provides an excellent means of scripting repeated sequences of debugging operations. Use this powerful language to develop full-blown test applications of DSP systems.

Analog Devices Tcl Commands

Analog Devices has extended Tcl version 8.3 with several procedures to access key debugging features. Use the power of the Tcl language, coupled with Analog Devices extensions to extensively script your work in VisualDSP++.

Overview of Tcl Scripting

VisualDSP++ provides these groups of Tcl commands:

- Target query and manipulation commands
- GUI manipulation commands
- Project build and maintenance commands

Additional Tcl Resources

The following resources can aid you in using Tcl to enhance your debug sessions.

- www.scriptics.com is a resource for Tcl programmers
- *Practical Programming in Tcl & Tk*, by Brent B. Welch (ISBN 0136168302)

Tcl Output

Tcl output is logged to `VisualDSP_log.txt`, which, by default, is located in the directory:

You can view the output of Tcl commands in the **Output** window's **Console** page. Tcl output is logged to `VisualDSP_log.txt`, which by default, is located in the following directory.

```
C:\Program Files\Analog Devices\VisualDSP\Data\
```

View this file to analyze the Tcl output.

Tcl Command Issuance

You can issue Tcl commands in various ways: from a command line, from the **Output** window, from the **File** menu, from an editor window, or from a user tool.

Issuing Commands from a Command Line

To load a script from a DOS command window, type this command:

```
idde -f filename
```

Optionally, add `-s` and the session name to specify a previously created session. When no session name is specified, the last session is used.

If the script encounters an error during execution, VisualDSP++ automatically exits.

Issuing Commands from the Output Window

In the **Output** window, you can type a Tcl command from the **Console** page, as shown in [Figure B-1](#).

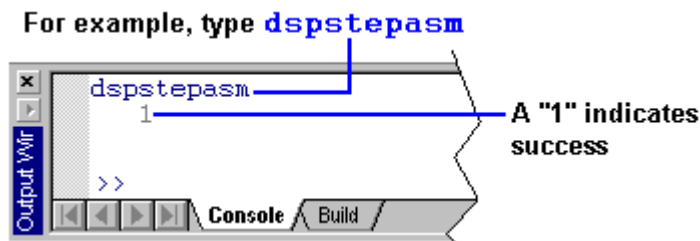


Figure B-1. Typing a Tcl Command from the Output Window

Tcl Command Issuance

Load a script from the **Output** window's **Console** page by typing:

```
source filename
```

Similar to C/C++, Tcl uses a backslash (\) as its escape character. When you specify paths in the Windows environment, you must escape the escape character. For example:

```
source c:\\my_dir\\my_subdir\\my_file.tcl
```

You can also use forward slashes to delimit directories in a path, for example:

```
source c:/my_dir/my_subdir/my_file.tcl
```

Command execution is deferred until a line is typed without a trailing backslash. This feature permits the entry of an entire block of code (or entire Tcl procedures) for the Tcl interpreter to evaluate at once.

Issuing Commands from the File Menu

You can quickly issue Tcl scripts of frequent use. From the **File** menu, choose **Recent Tcl Scripts**, and then select the Tcl script.

Issuing Commands from an Editor Window

In an open editor window that contains a Tcl script, right-click and choose **Source Tcl Script**, as shown in [Figure B-2](#).

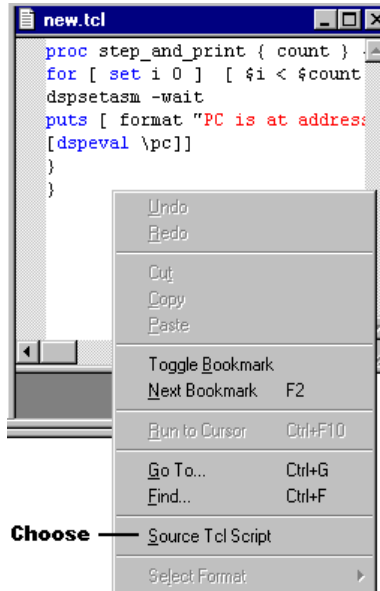


Figure B-2. Issuing a Tcl Script from an Editor Window

Issuing Commands from a User Tool

Click a user tool in the toolbar, or choose a user tool from the **Tools** menu.

Examples of Tcl Scripts

This section provides examples that show how to create and run a Tcl script.

Step and Print Example

This example shows how to create a Tcl script and run it from the **Output** window's **Console** page.

This Tcl procedure, named `step_and_print`, single steps through your assembly code a specified number of times. When you run the script, you supply the number of steps (the count), and the output is printed to a log file.

Creating the Tcl Script

1. Use a text editor to type the following code.

```
proc step_and_print { count } {  
    for { set i 0 } { $i < $count } { incr i } {  
        dspstepasm -wait  
        puts [ format "PC is at address 0x%x\n" [dspval \ $pc]]  
    }  
}
```

2. Save to `C:\Temp\test.tcl`.

Running the New Tcl Script

1. From the **Output** window's **Console** page, type the Tcl `source` command followed by the path and file name of the new Tcl script.

```
>source C:\\Temp\\test.tcl
```

Use double backslash characters.

2. Press **Enter**.

The Tcl script is loaded.

3. On the **Console** page, call the function and supply the step count, for example:

```
step_and_print 10
```

4. Press **Enter**.

The function executes. The program counter single steps ten times (the step count in the example shown in [Figure B-3](#)) and halts.

Output resulting from commands entered in the **Output** window's **Console** page is saved to `VisualDSP_log.txt`.

```

Loading C:\Program Files\Analog
Load complete.
> source c:\\temp\\test.tcl
> step and print 10
PC is at address 0x20005
PC is at address 0x20005
PC is at address 0x20006
PC is at address 0x20007

```

Source the Tcl command

Type the function call and the count

View the output

Figure B-3. Resulting Output

Regression Test Example

You can use Tcl with Analog Devices extensions to build sophisticated behaviors and boost productivity. The regression test below is used by Analog Devices software developers to test VisualDSP++'s debugging capability. This particular test ensures that function parameters evaluate correctly.

Note: At one time, developers performed this test manually with the GUI. The process took several minutes. With a Tcl script, the test takes seconds.

Two procedures are defined to help implement this test. The `goto_line` procedure runs the program to a certain line number. The `assert` procedure trips an error if a given expression evaluates to zero (false).

The regression test example follows.

```
proc assert { e } {  
    if { 0 == $e } {  
        error "Assertion Failed!"  
    }  
}  
  
proc goto_line { file line } {  
  
    # Lookup the address corresponding to {file, line} pair.  
    # The return value from dsplookupline is a list of start  
    # and end address (we use lindex to extract the car of  
    # the list).  
  
    dspsetbreak [ lindex [ dsplookupline $file $line ] 0 ] \  
        -temporary  
    dsprun  
    dspwaitforhalt  
}  
  
dspload argtest.dxe
```

```
goto_line main.c 126
```

```
assert [ expr 0x56          == [ dspeval "cNum" ] ]
assert [ expr 0x7890        == [ dspeval "sNum" ] ]
assert [ expr 0x1234        == [ dspeval "iNum" ] ]
assert [ expr 0xdeaddead    == [ dspeval "lNum" ] ]
assert [ expr 1.234         == [ dspeval "fNum" float ] ]
assert [ expr 5.678         == [ dspeval "dNum" float ] ]
```

```
goto_line main.c 58
```

```
assert [ expr 0x56          == [ dspeval "cNum" ] ]
assert [ expr 0x7890        == [ dspeval "sNum" ] ]
assert [ expr 0x1234        == [ dspeval "iNum" ] ]
assert [ expr 0xdeaddead    == [ dspeval "lNum" ] ]
assert [ expr 1.234         == [ dspeval "fNum" float ] ]
assert [ expr 5.678         == [ dspeval "dNum" float ] ]
```

```
assert [ expr 0x56          == [ dspeval "g_cNum" ] ]
assert [ expr 0x7890        == [ dspeval "g_sNum" ] ]
assert [ expr 0x1234        == [ dspeval "g_iNum" ] ]
assert [ expr 0xdeaddead    == [ dspeval "g_lNum" ] ]
assert [ expr 1.234         == [ dspeval "g_fNum" float ] ]
assert [ expr 5.678         == [ dspeval "g_dNum" float ] ]
```

```
assert [ expr 0x56          == [ dspeval "c" ] ]
assert [ expr 0x7890        == [ dspeval "s" ] ]
assert [ expr 0x1234        == [ dspeval "i" ] ]
assert [ expr 0xdeaddead    == [ dspeval "l" ] ]
assert [ expr 1.234         == [ dspeval "f" float ] ]
assert [ expr 5.678         == [ dspeval "d" float ] ]
```

```
goto_line main.c 142
```

```
assert [ expr 0xdeaf == [ dspeval "iNum" ] ]
```

Examples of Tcl Scripts

```
dsprun
dspwaitforhalt

exit
```

For reference, the source code for `main.c` is presented here.

Note: `ScareCompiler()` is an auxiliary stub function used to (intentionally) suppress compiler optimizations.

```
#include <stdio.h>

extern void ScareCompiler(void *pv);

char   g_cNum;
short  g_sNum;
int     g_iNum;
long    g_lNum;
float   g_fNum;
double  g_dNum;

int LotsOfArgs(char c, short s, int i, long l, float f, double d)
{
    char   cNum;
    short  sNum;
    int     iNum;
    long    lNum;
    float   fNum;
    double  dNum;

    cNum = c;
    sNum = s;
    iNum = i;
```



```

lNum = l;
fNum = f;
dNum = d;

ScareCompiler((void*)&cNum);
ScareCompiler((void*)&sNum);
ScareCompiler((void*)&iNum);
ScareCompiler((void*)&lNum);
ScareCompiler((void*)&fNum);
ScareCompiler((void*)&dNum);

g_cNum = cNum;
g_sNum = sNum;
g_iNum = iNum;
g_lNum = lNum;
g_fNum = fNum;
g_dNum = dNum;

ScareCompiler((void*)&g_cNum);
ScareCompiler((void*)&g_sNum);
ScareCompiler((void*)&g_iNum);
ScareCompiler((void*)&g_lNum);
ScareCompiler((void*)&g_fNum);
ScareCompiler((void*)&g_dNum);

/*****
Set a break here verify that the locals, arguments and globals
look ok. They should have the same values as the locals in
main().
Examples:
    g_cNum, cNum, & c should all equal 0x56
    g_sNum, sNum, & s should all equal 0x7890
*****/

```

Examples of Tcl Scripts

```
if(cNum != 0x56)
    printf("As If!\n");
if(sNum != 0x7890)
    printf("As If!\n");
if(iNum != 0x1234)
    printf("As If!\n");
if(lNum != 0xdeaddead)
    printf("As If!\n");
if(fNum != 1.234)
    printf("As If!\n");
if(dNum != 5.678)
    printf("As If!\n");
if(cNum && sNum && iNum && lNum && fNum && dNum)
    return (0xdeaf);
else
    return (0xbad);
}

/*****
TestArg test program
This tests the debugger and the debug information
generated by the compiler for the basic types. It tests
that the information is ok regardless of whether the
variable is a local or an argument to a function.
To use this file:
1. Load the argtest executable
2. Load the argtest layout file
3. Deposit break points at the specified locations
4. Enter the following into the expressions window
    g_cNum
    g_sNum
    g_iNum
    g_lNum
    g_fNum
```

g_dNum

5. Run evaluating correctness based on the comments in the code.

```

*****/

void main(void)
{
    char    cNum;
    short   sNum;
    int     iNum;
    long    lNum;
    float   fNum;
    double  dNum;

    cNum = 0x56;
    sNum = 0x7890;
    iNum = 0x1234;
    lNum = 0xdeaddead;
    fNum = 1.234;
    dNum = 5.678;

    ScareCompiler((void*)&cNum);
    ScareCompiler((void*)&sNum);
    ScareCompiler((void*)&iNum);
    ScareCompiler((void*)&lNum);
    ScareCompiler((void*)&fNum);
    ScareCompiler((void*)&dNum);

    /*****
        Set a break here
        verify that the locals look exactly
        as they were initialized. The ScareCompiler
        function does nothing to the variables.
    *****/
}

```

Types of Tcl Commands

```
if(cNum > 0x56)
    printf("No way!\n");
if(sNum > 0x7890)
    printf("No way!\n");
if(iNum > 0x1234)
    printf("No way!\n");
if(lNum > 0xdeaddead)
    printf("No way!\n");
if(fNum > 1.234)
    printf("No way!\n");
if(dNum > 5.678)
    printf("No way!\n");

/* This should return 0xdead */
iNum = LotsOfArgs(cNum, sNum, iNum, lNum, fNum, dNum);
// Set a break here check that iNum is 0xdead
if(iNum == 0xbad)
    printf("That's just wrong\n");
else
    printf("No problems\n");
}
```

Types of Tcl Commands

The three types of Tcl commands are:

- GUI manipulation commands
- Target query and manipulation commands
- Project build and maintenance commands

This section describes each group of Tcl commands.

GUI Manipulation Commands

Use the Graphic User Interface (GUI) manipulation commands in [Table B-1](#) to create windows and menu items without using the GUI.

Table B-1. GUI Manipulation Command Summary

Command	Description
“dspaddmenuitem” on page B-20	Adds a menu item (command) to the menu bar
“dspcheckmenuitem” on page B-25	Queries or sets the “checked” attribute for a menu item
“dspclickmenuitem” on page B-26	Simulates a mouse click of a menu item
“dspdeletemenuitem” on page B-28	Deletes a menu item
“dspenablemenuitem” on page B-30	Queries or sets the “enabled” attribute of a menu item
“dspmemorywin” on page B-46	Displays a memory window
“dspregisterwin” on page B-61	Creates a custom register window

Types of Tcl Commands

Target Query and Manipulation Commands

Use the Tcl commands in [Table B-2](#) to query the target and manipulate values.

Table B-2. Target Query and Manipulation Command Summary

Command	Description
“dspsetbreak” on page B-66	Sets (inserts) a breakpoint
“dpscancelbreak” on page B-24	Cancels (deletes) a breakpoint
“dspgetbreak” on page B-33	Returns information about the breakpoint
“dspeval” on page B-31	Evaluates an expression
“dspgetmemblock” on page B-35	Fetches a block of memory
“dspgetmeminfo” on page B-37	Gets information about types of memory. This information is used by other commands.
“dspgetprocessors” on page B-38	Gets the names of the processors in a multiprocessor debug session
“dspgetstate” on page B-39	Gets the current state of a processor
“dspgetswstack” on page B-40	Gets the C-language software stack for a processor
“dsphalt” on page B-41	Requests a halt of the processor
“dspload” on page B-43	Loads a file to the target
“dsplookupline” on page B-44	Looks up the start and end address corresponding to a line in a file
“dsplookupsymbol” on page B-45	Looks up the address of a symbol identified by a label

Table B-2. Target Query and Manipulation Command Summary (Cont'd)

Command	Description
“dspplotrotate” on page B-48	Rotates a waterfall plot by azimuth and elevation
“dspplotwin” on page B-49	Configures and displays a plot in a plot window
“dspreset” on page B-62	Sends a reset message to the target
“dsprestart” on page B-63	Sends a restart message to the target
“dsprun” on page B-64	Sends a run message to the target
“dspset” on page B-65	Evaluates an expression and assigns the value to another expression
“dspgetmemblock” on page B-35	Sets a block of memory
“dspsetswstack” on page B-70	Changes the debug focus
“dspstepasm” on page B-71	Steps the target a single disassembly step
“dspstepin” on page B-72	Steps the target a single source language step
“dspstepout” on page B-73	Steps the target out of the current subroutine in a source language
“dspstepover” on page B-74	Steps the target a single source language step, skipping over any subroutine calls
“dspwaitforhalt” on page B-75	Delays further execution until the target halts
“dspaddstream” on page B-22	Adds a stream to the debug session
“dspdeletestream” on page B-29	Deletes the stream from the debug session

Types of Tcl Commands

Table B-2. Target Query and Manipulation Command Summary (Cont'd)

Command	Description
“dspdeleteallstream” on page B-27	Deletes all streams from the debug session
“dspliststream” on page B-42	Lists all the streams in the debug session

Project Build and Maintenance Commands

Use the Tcl commands in [Table B-3](#) to operate at the project level.

Table B-3. Project Build and Maintenance Command Summary

Command	Description
“dspprojectload” on page B-58	Loads the project into VisualDSP++
“dspprojectbuild” on page B-55	Builds the currently loaded project configuration
“dspprojectinfo” on page B-57	Returns information about the currently loaded project
“dspprojectaddfile” on page B-53	Adds a file to the current project
“dspprojectaddfolder” on page B-54	Adds a folder to the current project
“dspprojectremovefile” on page B-59	Removes a file from the current project
“dspprojectremovefolder” on page B-60	Removes a folder from the current project
“dspsetbreak” on page B-66	Closes the currently loaded project

Tcl Command Reference

This section provides descriptions of the syntax, purpose, and arguments for each Tcl command. Examples are included for some commands.

In the syntax statements, optional Tcl command parameters are indicated with question-mark characters (?). For example, in the following syntax statement, `-all` and `-wait` are optional parameters.

```
dspprojectbuild projconfig ?-all? ?-wait?
```

dspaddmenuitem

Syntax

```
dspaddmenuitem menuItem callback  
?-head? ?-info value? ?-help value?
```

Purpose

This command adds a menu item to the menu bar.

Returns: a cookie representing the identifier for the menu item. The cookie is used in calls to `dspcheckmenuitem`, `dspenablemenuitem`, and `dspdeletemenuitem`. The cookie is passed to the callback function, allowing a single callback function to service multiple menu items.

Arguments

`menuItem`

Specifies the path to the menu item. It is of the format `MENU:SUBMENU:SUBMENU:ITEM`. This specifies the menu tree that needs to be traversed to access the menu item.

`callback`

Specifies the Tcl procedure called when the menu item is clicked. This function must be of the form `proc function_name { id } { body }`. The `id` parameter to this callback function is the cookie for the menu item (see “Returns” above).

`-head`

Specifies that the menu item is to be prepended to the menu. Otherwise, the menu item is appended to the menu.

-info value

Specifies an information string to be displayed in the application's status bar when the menu item has focus.

-help value

Specifies a callback to a Help function. This callback has the same format as the function callback described above.

Example

The following script creates a menu item that outputs a message when you choose the menu item.

```
proc callback { id } {  
    puts [ format "Menu ID %d clicked.\n" $id ]  
}  
  
set id [ dspaddmenuitem "Custom:Menu #1" callback \  
    -info "Demo menu item" ]  
puts [ format "Menu ID %d installed.\n" $id ]
```

dspaddstream

Syntax

```
dspaddstream -source value1
              ?-sourceproc value2?
              ?-sourceaddress value3?
              ?-dest value4?
              ?-destproc value5?
              ?-destaddress value6?
              ?-format value7?
              ?-circular?
              ?-norewind?
```

Purpose

This command adds a stream to the debug session.

Returns: stream *id* if successful or 0 otherwise

Arguments

-source *value1*

value1 specifies the source of the stream connection (a device or a file)

-sourceproc *value2*

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. *value2* specifies the processor to which the source device is attached. Omitting this argument steers the command toward the currently focused processor.

-sourceaddress *value3*

Addresses for devices such as memory mapped I/O ports

`-destproc value5`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. `Value5` specifies the processor to which the destination device is attached. Omitting this argument steers the command toward the currently focused processor.

`-destaddress value6`

Addresses for devices such as memory mapped I/O ports

`-format value7`

`value7` specifies the format of the file being used, for example: hexadecimal (default), integer, unsigned integer, float, and octal.

`-circular`

Specifies whether the source file is circular. If it is, after the last data is read from the file, the data from the beginning of the file is read. By default, circular is disabled.

`-norewind`

By default, on reset or restart, the file pointer is rewound to the start of the file. When this option is specified, the file is not rewound. Use this option during a regression test to continue testing.

Example

```
dspaddstream -source "TX1" -dest "d:/transmit.dat"
              -format "Unsigned Integer"
```

This command specifies the source as TX1 and the destination as a file (d:/transmit.dat) in unsigned-integer format.

dspcancelbreak

Syntax

```
dspcancelbreak ?-processor value? id
```

Purpose

This command cancels (deletes) the breakpoint identified by `id`.

Returns: `id`

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. `value` specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`id`

An identifier from a previous call to [“dspsetbreak” on page B-66](#). Using [“dspgetbreak” on page B-33](#) can also determine a breakpoint’s identifier.

Example

```
# Cancel all breakpoints
foreach bp [dspgetbreak] { dspcancelbreak [lindex $bp 0] }
```

dspcheckmenuitem

Syntax

```
dspcheckmenuitem id ?value?
```

Purpose

This command queries or sets the checked attribute of the menu item identified by `id`.

Returns: the string “checked” or “unchecked”

Arguments

`id`

Obtained from a previous call to `dspaddmenuitem`

`value`

Specifies the new value for this attribute. Valid values are `checked` and `unchecked`. If `value` is not specified, the value of the attribute is not altered.

dspclickmenuitem

Syntax

```
dspclickmenuitem menuItem
```

Purpose

This Tcl command simulates a mouse click on a menu item.

Returns: 1 if successful or 0 otherwise (when the menu item does not exist)

Arguments

menuItem

Specifies the menu item to click. The format, MENU:SUBMEU:SUBMENU:ITEM, specifies the menu tree to be traversed to access the menu item.

Example

This command refreshes the window:

```
> dspclickmenuitem Window:Refresh
```


dspdeleteallstream

Syntax

```
dspdeleteallstream
```

Purpose

This command deletes all the streams in the debug session.

Returns: nothing

dspdeletemenuitem

Syntax

```
dspdeletemenuitem id
```

Purpose

This command deletes the menu item specified by `id`.

Returns: nothing

Arguments

`id`

Obtained from a previous call to `dspaddmenuitem` (see [“dspaddmenuitem”](#) on page B-20)

dspdeletestream

Syntax

```
dspdeletestream id
```

Purpose

This command deletes the stream identified by `id`.

Returns: `id` of the stream if successful or 0 otherwise

Example

```
dspdeletestream 1
```

dspenablemenuitem

Syntax

```
dspenablemenuitem id ?value?
```

Purpose

This command queries or sets the enabled attribute of the menu item identified by `id`.

Returns: the string `enabled`, `disabled`, or `grayed`

Arguments

`id`

Obtained from a previous call to `dspaddmenuitem` (see [“dspaddmenuitem” on page B-20](#))

`value`

Specifies the new value for this attribute. Valid values are `enabled`, `disabled`, or `grayed`. If `value` is not specified, the value of the attribute is not altered.

dspeval

Syntax

```
dspeval [-processor value] expr ?format?
```

Purpose

This command evaluates an expression specified by `expr`. Valid forms of expressions are the expressions that can be accepted by the **Expressions** window.

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. `value` specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`expr`

The expression to be evaluated

`format`

Specifies the format used to format the output. Valid values are hexadecimal (default), integer, unsigned integer, float, and octal.

Escape Character

When using the dollar sign (\$) character in an expression, escape this character with a backslash (\).



The \$ is the variable signifier in Tcl.

Tcl Command Reference

Examples

- The following example returns the value of the expression or an error message if a problem was encountered.

```
> dspeval "\$PC"
```

- The following example evaluates a C expression.

```
> dspeval "&my_array[5]"
```

- The following example fetches the opcode at the PC.

```
> dspeval "\$PM (\$PC)"
```

dspgetbreak

Syntax

```
dspgetbreak ?-processor value? ?id?
```

Purpose

This command returns a list containing information about the breakpoint identified by `id`.

The list consists of these elements:

- The `id` of the breakpoint
- The software overlay or hardware page in which the breakpoint is located (-1, if none)
- The address of the breakpoint
- The source file of the breakpoint ("", if unknown)
- The line number of the breakpoint (zero if unknown)
- “temporary” or “permanent”
- “enabled” or “disabled”
- “placed” or “unplaced”
- The skip count
- The test expression ("", if unknown)

If `id` is omitted, a list of *all* breakpoint information is returned.

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. `value` specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`id`

A value previously returned by [“dspsetbreak” on page B-66](#)

dspgetmemblock

Syntax

```
dspgetmemblock ?-processor value?
                memory
                start
                count
                ?-stride value?
                ?-format value?
```

Purpose

This command fetches a block of memory and returns a list comprised of the memory fetched. Each element of the list represents the value of a single address.

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. *value* specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

memory

Specifies the memory from which to fetch the block. This value is one of the strings outputted by [“dspgetmeminfo” on page B-37](#).

start

Specifies the first address to fetch in the block

count

Specifies the total number of addresses to fetch

Tcl Command Reference

`-stride value`

Specifies the distance between memory locations. If `value` is omitted, 1 is used.

`-format value`

Specifies the format used to format the output. Valid values are hexadecimal (default), integer, unsigned integer, float, and octal. Values vary from target to target. The default is hexadecimal.

Example

```
> set pm [ lindex [ lindex [dspgetmeminfo] 0 ] 0 ]  
      Program(PM) Memory  
> dspgetmemblock $pm 0x20004 3 -format "Assembly"  
      {nop;} {jump __lib_start;} {nop;}
```

dspgetmeminfo

Syntax

```
dspgetmeminfo ?-processor value?
```

Purpose

This command returns information about the types of memory within the target and returns a list of memories. Each element of the list is itself a list comprised of two elements:

- An ASCII string representing the canonical name of the memory
- The width of the memory in bits

Returns: information used by other commands to identify a particular memory by its name

Arguments

```
-processor value
```

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. `value` specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

Example

The SHARC simulator's output:

```
> dspgetmeminfo

{ "Program(PM) Memory" 48 } { "Data(DM) Memory" 32 } { "Data(DM)
Short Word Memory" 16 } { "Data(DM) 40-bit Memory" 48 }
{ "External Data(DM) Byte Memory" 8 } { "Boot (PROM) Memory" 8 }
```

dspgetprocessors

Syntax

```
dspgetprocessors
```

Purpose

This command returns the names of the processors in a multiprocessor debug session. For a single processor session, an empty list is returned.

These names are used in the `-processor` argument of other Tcl commands.

dspgetstate

Syntax

```
dspgetstate ?-processor value?
```

Purpose

This command returns the current state of a processor.

Returns: one of these strings: reset, running, stepping, halted, loaded, or unknown

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

dspgetswstack

Syntax

```
dspgetswstack ?-processor value?
```

Purpose

This command returns the C-language software stack for a processor.

 Do not confuse this stack with the internal hardware stack found on some targets.

Returns: a list of frames. Each frame is a list comprised of a name (C function) and a cookie value. The cookie value changes the debug focus to a different frame (see [“dspsetswstack” on page B-70](#)). If a frame currently has the debug focus, a third item in the list is the string focus. One frame only may have focus at a given time.

If the stack cannot be found or identified because the program is in an assembly language module (or no debug information is present), a null list is returned.

Arguments

```
-processor value
```

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

dsphalt

Syntax

```
dsphalt ?-processor value? ?-wait?
```

Purpose

This command sends a halt request message to the target.

This command ensures only that a message is sent. It does not guarantee that the target will halt.

Returns: 1

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

-wait

Prevents further execution of a Tcl script until the target has halted

dspliststream

Syntax

```
dspliststream
```

Purpose

This command lists all the streams in the debug session.

Returns: the list of streams. The list consists of the `id` of the stream, source device or a file, destination device or a file.

dspload

Syntax

```
dspload ?-processor value? fileName ?-symbols? ?-wait?
```

Purpose

This command loads a file to the target. Use this command when you debug a ROM target.

Returns: 1 if successful or an error message otherwise

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`fileName`

Specifies the file to be loaded to the target

`-symbols`

Indicates that symbolic debugging information only be loaded from the file. The target itself is not loaded with the binary's image. Use this option to debug a ROM target.

`-wait`

Prevents further execution of a Tcl script until the target has halted

dsplookupline

Syntax

```
dsplookupline ?-processor value? file line
```

Purpose

This command looks up the start and end address for the `file line`

Returns: a list comprising two elements, representing the start and end addresses. If line number information cannot be determined, an error is returned.

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`file`

Specifies the name of the file

`line`

Specifies the line in the file

Example

This command sets a breakpoint at a line:

```
> dspsetbreak [lindex [dsplookupline foo.c 100] 0]
```

This example uses Tcl's `lindex` command to access the first element in the two-element list.

dsplookupsymbol

Syntax

```
dsplookupsymbol ?-processor value? ?-memory value? label
```

Purpose

This command looks up the address of a symbol.

Returns: the label's address

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`-memory value`

Specifies the memory in which the look up is to be performed. The form of this argument is one of the strings returned by the `dspget-meminfo` command (see [“dspgetmeminfo” on page B-37](#)).

`label`

Specifies the symbol

dspmemorywin

Syntax

```
dspmemorywin memory addr
               ?-rect { left top right bottom }?
               ?-title name? ?-track expr?
```

Purpose

This command opens a memory window for the memory identified by `memory` at the address specified by `addr`.

Returns: 1

Arguments

`memory`

One of the strings returned by `dspgetmeminfo` (see [“dspgetmeminfo” on page B-37](#)).

`addr`

Specifies the address

```
-rect { left top right bottom }
```

Specifies the coordinates of the rectangle enclosing the window. This list has four integer values. If the rectangle size is not specified, the MS Windows system library sets the size.

```
-title name
```

Assigns the specified name to the window

`-track expr`

Focuses the window on a specific activity. The expression `expr` is evaluated at every processor halt, and the window moves to the address based on the evaluation of the expression. Use this option to follow pointer movement.

dspplotrotate

Syntax

```
dspplotrotate title az el
```

Purpose

This command rotates a waterfall plot. You choose the degree of azimuth rotation and elevation rotation.

Returns: an error message in the **Output** window if the command fails

Arguments

title

This title must match a title of a previously defined waterfall plot.

az

The azimuth rotation angle (from 0 to 360 degrees)

el

The elevation viewpoint (from -90 to +90 degrees)

dspplotwin

Syntax

```
dspplotwin memtype addr count
        ?-stride value?
        ?-datatype value?
        ?-plottype value?
        ?-title value?
        ?-setname value?
        ?-xmemtype value?
        ?-xaddr value?
        ?-xstride value?
        ?-xdatatype value?
        ?-columncount value?
        ?-add?
```

Purpose

This Tcl command creates and displays a plot in a plot window.

Returns: on failure, an error message in the **Output** window

Arguments

memtype

The Y-axis or Z-axis memory type, such as \$dm or \$pm, depends on the plot type.

Tcl Command Reference

`addr`

The Y-axis or Z-axis address depends on the plot type.

`count`

The Y-axis or Z-axis row count depends on the plot type.

`-stride value`

The Y-axis or Z-axis stride depends on the plot type.

`-datatype value`

The Y-axis or Z-axis data type, such as float, depends on the plot type.

`-plottype value`

Specifies the desired plot type. Choose: line, xy, constellation, eyediagram, waterfall, or image.

`-title value`

The plot's title. Surround the text with double-quote characters (").

`-setname value`

The name of this data set. Surround the text with double-quote characters (").

`-xmemtype value`

The X-axis memory type, such as \$dm, depends on the plot type.

`-xaddr value`

The X-axis address depends on the plot type.

`-xstride value`

The X-axis stride depends on the plot type.

`-xdatatype value`

The X-axis data type, such as float, depends on the plot type.

`-columncount value`

The Z-axis column count

`-add`

Specifies that you are adding data to a previously defined plot

Examples

The following commands configure and display various plots.

Eye Diagram Plot

```
dspplotwin
    $dm eyedata 100
    -datatype float
    -plottype eyediagram
    -title "Eye Diagram"
    -setname "Input"
```

Constellation Plot

```
dspplotwin
    $dm ydata 100
    -datatype float
    -plottype constellation
    -xmementype $dm
    -xaddr xdata
    -xdatatype float
    -title "Constellation Plot"
    -setname "Input"
```

Waterfall Plot

```
dspplotwin
    $dm zdata 64
    -datatype float
    -plottype waterfall
    -title "Waterfall Plot"
    -setname "Input"
    -columncount 20
```

dspprojectaddfile

Syntax

```
dspprojectaddfile ?-folder name? filename
```

Purpose

This command adds the file's `filename` to the currently loaded project.

Returns: an error condition when:

- No project is loaded
- `filename` does not exist
- `filename` is already in the project

Arguments

`-folder name`

Specifies the folder in which to place the file. If the folder does not exist, it is created. If no folder is specified, the file is inserted at the root of the project file hierarchy.

`filename`

Specifies the name of the file to be added

dspprojectaddfolder

Syntax

```
dspprojectaddfolder ?-ext extensions? foldername
```

Purpose

This command adds the folder's `foldername` (for example, `folder1` or `folder1/folder2`) to the currently loaded project.

Returns: an error condition occurs when no project is loaded. An attempt to add a folder currently in the project results in success.

Arguments

`-ext extensions`

Specifies extensions for the folder

`Foldername`

Specifies the name of the folder. A parent folder is created if it is specified in the path name but does not currently exist.

dspprojectbuild

Syntax

```
dspprojectbuild projconfig ?-all? ?-wait?
```

Purpose

This command builds the configuration `projconfig` of the currently loaded project.

Returns:

- When the `-wait` parameter is used, 1 to indicate that the project build was successful, and 0 to indicate that the build was unsuccessful
- When the `-wait` parameter is not used, 1 to indicate that the call was successful. Otherwise, the call was not successful.
- An error condition in response to any of the following: `projconfig` does not exist, no project is loaded, or the build fails for any reason

Arguments

`projconfig`

Identifies the project configuration

`-all`

Performs a “rebuild all.” If you do not specify this argument, an increment build is preformed.

`-wait`

Prevents completion of the call until after the project build is completed. Without this switch, the call returns right away.

dspprojectclose

Syntax

```
dspprojectclose ?-save? ?filename?
```

Purpose

This command closes the currently loaded project.

Returns: an error condition when:

- No project is loaded
- The project cannot be saved (for example, when the project file is read only)
- `filename` is specified and `-save` is not

Arguments

`-save`

Specifies that changes to the project be saved. If the flag is omitted, the project is closed and changes are not saved.

`filename`

Specifies the file name to save to (that is, “**Save As**”). If the file name is not specified, the project is saved to the location from which it was loaded.

dspprojectinfo

Syntax

```
dspprojectinfo
```

Purpose

This command returns information about the currently loaded project.

The return value is a list comprising these two members:

- A list of the project's configurations
- A list of the project's input files. Each element of the list is a list containing the file name and project folder. Project folders use a file hierarchy-like syntax, with "/" to indicate the root of the project's file hierarchy.

Returns: an error condition when no project is loaded

Example

```
% dspprojectinfo
{debug release} {{ "C:/ProjectA/ProjectA.c" "/" } \
 "C:/ProjectA/ProjectA.h" "/Include Files" }
```

In this example, the returned information indicates that the project has two configurations (debug and release) and comprises two input files.

dspprojectload

Syntax

```
dspprojectload projectname
```

Purpose

This command loads the project's `projectname` into VisualDSP++.

Returns: an error condition when:

- `projectname` does not exist
- Another project is already loaded into VisualDSP++

dspprojectremovefile

Syntax

```
dspprojectremovefile filename
```

Purpose

This command removes the file `filename` from the currently loaded project.

Returns: an error condition when:

- No project is loaded
- `filename` is not included in the project

dspprojectremovefolder

Syntax

```
dspprojectremovefolder foldername
```

Purpose

This command removes the folder's `foldername` from the currently loaded project.

Returns: an error condition when:

- No project is loaded
- `foldername` is not included in the project

dsregisterwin

Syntax

```
dsregisterwin { { name x y ?type? } }
              ?-title name?
              ?-format value?
              ?-rect { left top right bottom }?
```

Purpose

This command creates and displays a custom register window.

Returns: 1

Arguments

```
{ { name x y [type] } }
```

A list of registers. Each element of this list is itself a list, containing information for a single register, including: the ASCII name of the register, the X and Y coordinates that position the register in the window (measured in characters), and (optionally), the type of the register. Valid values for type are `normal` (default), `nodata`, `noLabel`, and `private`.

```
-title name
```

Assigns a title to the register window

```
-format value
```

Specifies the base format for which the X and Y position were calculated (default value is hexadecimal)

```
-rect { left top right bottom }
```

A list comprising four integers. If you do not specify the rectangle size, the MS Windows system library picks the size.

dspreset

Syntax

```
dspreset ?-processor value? ?-wait?
```

Purpose

This command sends a message to the target to reset.

Returns: 1

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`-wait`

Prevents further execution of a Tcl script until the target has halted

dsprestart

Syntax

```
dsprestart ?-processor value? ?-wait?
```

Purpose

This Tcl command sends a message to the target to restart.

Returns: 1

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

-wait

Prevents further execution of a Tcl script until the target has halted

dsprun

Syntax

```
dsprun ?-processor value? ?-wait?
```

Purpose

This Tcl command sends a message to the target to run.

Returns: 1

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`-wait`

Prevents further execution of a Tcl script until the target has halted

dspset

Syntax

```
dspset [-processor value] left_expr right_expr
```

Purpose

This command evaluates `right_expr` and assigns its value to `left_expr`.

Only rudimentary checking is performed. Modifiers like `const` are ignored.

Returns: `right_expr`

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`left_expr`

Specifies the expression to be assigned a value. This expression must be an lvalue.

`right_expr`

Specifies the expression to be evaluated

Example

Set the value of R0 to the address of a C variable.

```
> dspset \R0 "&my_variable"
```

dspsetbreak

Syntax

```
dspsetbreak ?-processor value?  
            address  
            ?-expression value?  
            ?-count value?  
            ?-temporary?  
            ?-disabled?
```

Purpose

This command sets (inserts) a breakpoint on the target at the address specified by `address`.



This functionality was provided by the `dspbreakpoint` command in earlier software releases.

Returns: a non-zero value representing the identifier of the breakpoint if successful or 0 otherwise. The returned value can be stored and used in subsequent calls to [“dspcancelbreak” on page B-24](#) and [“dspgetbreak” on page B-33](#).

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`address`

The address to which a breakpoint is specified

`-expression value`

Specifies a condition that must be evaluated to TRUE to halt execution of the debug target at the breakpoint address. If `expression` and `count` are omitted, execution of the debug target stops at the breakpoint. Valid expressions are anything that the **Expressions** window accepts.

`-count value`

Delays the setting of the breakpoint. The `value` argument specifies the number of halts that pass before setting the breakpoint

`-temporary`

Cancels the breakpoint at the next halt (implements features like run-to-cursor)

`-disabled`

Disables the breakpoint

Example

The following command sets a temporary breakpoint at `main()`.

```
dspsetbreak [dsplookupsymbol main] -temporary
```

dspsetmemblock

Syntax

```
dspsetmemblock ?-processor value? memory start count  
?-stride value? ?-format value? { fill ... }
```

Purpose

This command sets a block of memory.

Returns: 1

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

memory

Specifies the memory to set the block in and is one of the strings outputted by `dspgetmeminfo` (see [“dspgetmeminfo” on page B-37](#)).

start

Specifies the first address to set in the block.

count

Specifies the total number of addresses to set. If the length of `fill` is less than `count`, the `fill` values wrap to provide values for all `count` addresses.

`-stride value`

Specifies the distance between memory locations. If `value` is not specified, 1 is used.

`-format value`

Specifies the format used to format the data. Valid values are hexadecimal (default), integer, unsigned integer, float, and octal. Values vary from target to target.

`fill`

This list specifies the value(s) with which to fill memory. If the length of `fill` is less than the `count`, the `fill` values wrap to provide values for all `count` addresses.

Example

This example fills ten addresses in SHARC memory with a dummy fill value (as Microsoft Visual C++ does in `malloc()`).

```
> set mem [lindex [lindex [dspgetmeminfo] 1] 0]
Data(DM) Memory
> dspsetmemblock $dm 0x30000 10 0xcdcdcdcd
1
```

dspsetswstack

Syntax

```
dspsetswstack ?-processor value? cookie
```

Purpose

This command changes the debug focus to the frame identified by cookie.

Returns: 1

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

cookie

Specifies the frame. This value is determined by calling `dspgetswstack` (see [“dspreset” on page B-62](#)).

Example

Notice the movement of focus.

```
>dspgetswstack
{"foo()" 0x2fff5 focus} {"main(int, char**)" 0x2ffff}
>dspsetswstack 0x2ffff
1
>dspgetswstack
{"foo()" 0x2fff5} {"main(int, char**)" 0x2ffff focus}
```

dspstepasm

Syntax

```
dspstepasm ?-processor value? ?-wait?
```

Purpose

This command steps the target a single disassembly step.

Returns: 1

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

-wait

Prevents further execution of a Tcl script until the target has halted

dspstepin

Syntax

```
dspstepin ?-processor value? ?-wait?
```

Purpose

This command steps the target a single source language step.

Returns: 1 if successful or 0 when source stepping is not enabled at PC

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

-wait

Prevents further execution of a Tcl script until the target has halted

dspstepout

Syntax

```
dspstepout ?-processor value? ?-wait?
```

Purpose

This command steps the target out of the current subroutine in a source language.

Returns: 1 if successful or 0 otherwise (that is, source stepping not enabled at PC)

Arguments

-processor value

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

-wait

Prevents further execution of a Tcl script until the target has halted

dspstepover

Syntax

```
dspstepover ?-processor value? ?-wait?
```

Purpose

This command steps the target a single source language step, but skips over subroutine calls.

Returns: 1 if successful or 0 otherwise (source stepping not enabled at PC)

Arguments

`-processor value`

Relevant only in a multiprocessor debug session and ignored during a single processor debug session. The `value` argument specifies the processor to which the command is directed. Omitting this argument steers the command toward the currently focused processor.

`-wait`

Indicates that the script should execute and halt the target, which places the target in a known state

dspwaitforhalt

Syntax

```
dspwaitforhalt
```

Purpose

This command delays further execution of Tcl commands until the target has halted.

Returns: 1

I INDEX

Symbols

- .ASM files, [2-19](#), [A-21](#)
- .C files, [2-19](#), [A-21](#)
- .CPP files, [2-19](#), [A-21](#)
- .CXX files, [2-19](#), [A-21](#)
- .DAT files, [A-21](#)
- .DLB files, [2-19](#), [A-21](#)
- .DLO files, [A-21](#)
- .DOJ files, [1-23](#), [1-25](#), [1-26](#), [2-19](#), [A-21](#)
- .DPJ files, [1-45](#), [A-21](#)
- .DSP files, [2-19](#)
- .DXE files, [1-26](#), [A-21](#)
- .EXE files, [A-21](#)
- .H files, [1-32](#), [A-21](#)
- .H# files, [A-21](#)
- .HPP files, [A-21](#)
- .HXX files, [A-21](#)
- .IS files, [A-21](#)
- .LDF files, [1-25](#), [1-26](#), [1-27](#), [2-19](#), [A-21](#)
- .LDR files, [1-33](#), [3-34](#)
- .LST files, [A-21](#)
- .MAP files, [A-21](#)
- .OVL files, [1-26](#), [A-21](#)
- .PP files, [A-21](#)
- .S files, [1-32](#), [2-19](#), [A-21](#)
- .S# files, [A-21](#)
- .SM files, [1-26](#), [A-21](#)

- .STK files, [1-32](#), [A-21](#)
- .TCL files, [A-21](#)
- .VDK files, [A-21](#)

Numerics

- 3-D waterfall plots (*see* waterfall plots)

A

- adding files to your project, [1-18](#)
- anomalies
 - ADSP-2116x DSPs, [3-31](#)
 - clock doubling, [3-31](#)
 - shadow write FIFO, [3-27](#)
 - short word, [3-24](#)
 - SIMD FIFO, [3-28](#), [I-1](#)
- anomaly events, recording, [3-28](#)
- applying
 - file build options, [1-20](#)
 - project build options, [1-20](#)
- archiver, [1-32](#)
- assembler, [1-25](#)
 - about, [1-25](#)
 - input files, [2-19](#)
 - terms, [1-25](#)
- assembling language files into object files, [1-25](#)

INDEX

B

- bookmarks, [2-89](#)
- boot
 - kernel, [1-34](#)
 - loading or booting, [1-34](#), [3-34](#)
 - options, [3-33](#)
- boot-loadable files, [1-32](#), [1-33](#), [3-34](#)
- breakpoints
 - conditional, [3-21](#)
 - editor window, [2-89](#)
 - symbols, [3-20](#)
 - unconditional, [3-21](#)
- build settings, [1-54](#)
 - custom, [1-54](#)
 - individual file, [1-54](#)
 - project wide, [1-54](#)
- build type (*see* configuration)
- build, project, [1-53](#)
- buttons
 - appearance on toolbars, [2-10](#)
 - for standard Windows functions, [2-48](#)
 - on built-in toolbars, [2-8](#)

C

- C programs, compiling, [1-23](#)
- C++ programs, compiling, [1-23](#)
- C++ run-time libraries, [1-24](#)
- Call Stack window, [2-67](#)
- clock doubling, [3-31](#)
 - clock rate ratios, [3-31](#)
 - configuring the CLKOUT pin, [3-32](#)
 - Crystal Double Mode Enable pin, [3-31](#)

- code
 - development tools, [1-2](#), [2-19](#)
 - file association with tools, [2-19](#)
- command-line
 - parameters, [A-29](#)
- commands
 - on a control menu, [2-5](#)
 - program execution operation, [3-18](#)
 - single stepping, [3-18](#)
 - stepping, [3-18](#)
 - toolbar buttons, [A-30](#)
 - user tools, [2-13](#)
- comments
 - rules for, [A-41](#)
 - start and stop strings, [A-41](#)
- compiler, [1-23](#)
 - input files, [2-19](#)
 - options, [1-23](#)
- compiling, [1-23](#)
 - C programs, [1-23](#)
 - C++ programs, [1-23](#)
- conditional breakpoints, [3-21](#)
- configuration, [1-52](#)
 - plot, [2-99](#)
 - project, [1-52](#)
 - release, [1-52](#)
- Configurator, VisualDSP ICE, [3-4](#)
- Configure Simulator Event dialog box, [3-29](#), [3-30](#)
- configuring
 - Plot window, [2-99](#)
 - plots, [2-99](#)
- constellation plots, [3-12](#)
- control menu, [2-4](#), [2-5](#)

- conventions, manual, [xxx](#)
- creating
 - files to add to your project, [1-18](#)
 - new plot window, [2-99](#)
- custom build
 - options, [1-20](#)
 - settings, [1-54](#)
- custom register windows, [2-77](#)
- customer support, [xxiii](#)
- customizing
 - plot window, [2-99](#)
 - toolbar, [2-9](#)
- D**
- data
 - files, [1-25](#)
 - input and output simulation, [3-22](#)
 - sets, defined, [2-99](#)
 - transfers, simulating, [1-15](#)
- debug configuration, [1-52](#)
- debug sessions
 - managing, [3-3](#)
 - multiprocessor, [3-4](#)
 - running multiple, [3-3](#)
 - selecting at startup, [3-17](#)
 - setting up, [3-2](#)
 - switching, [3-3](#)
 - viewing list of, [3-3](#)
- debugging
 - features of VisualDSP++, [1-5](#)
 - IDDE features, [1-5](#)
 - multiple processors, [3-3](#)
 - overview of, [1-15](#)
 - windows used while debugging, [2-49](#)
- declarations, [1-27](#)
- dependencies, project, [1-53](#)
- developing, setting custom build project
 - options, [1-20](#)
- development tools, [1-2](#)
- Disassembly windows, [2-51](#), [2-52](#), [2-53](#)
 - examples, [2-51](#)
 - features, [2-53](#)
 - right-click menu, [2-53](#)
 - symbols, [2-54](#)
- docking, [2-43](#)
 - toolbars, [2-10](#)
 - windows, [2-43](#)
- dotprodc.dxe, automatically loading, [1-21](#)
- DSP
 - development tools, [1-2](#)
 - plotting memory, [3-9](#)
- E**
- editing
 - features, [1-3](#)
 - files, [1-19](#)
- editor files, comments, [A-41](#)
- Editor Tab mode, [2-30](#)
- editor windows
 - bookmarks, [2-89](#)
 - Editor Tab mode, [2-30](#)
 - expression evaluation, [2-89](#)
 - features, [2-87](#)
 - source mode vs. mixed mode, [2-90](#)
 - symbols, [2-89](#)
- elfloader.exe, [1-33](#), [1-34](#)
- emulation, [1-16](#), [3-7](#), [3-8](#)

INDEX

- debug session management, [3-3](#)
- restarting the program, [3-19](#)
- statistical profiling, [3-7](#)
- environment, simulating hardware, [1-15](#)
- error messages, [2-31](#), [2-40](#), [2-49](#)
 - in the Output window, [2-31](#)
 - log file, [2-40](#), [2-41](#), [2-49](#)
- evaluating expressions, [2-89](#)
- events
 - thread, [2-85](#)
 - using the data cursor, [2-85](#)
 - viewing details and thread status, [2-85](#)
- executable, loading, [3-18](#)
- Expert Linker, [1-28](#)
 - overview, [1-28](#)
 - stack and heap usage, [1-30](#)
 - window, [1-29](#)
- expressions
 - about, [2-93](#)
 - C expressions, [2-93](#)
 - context-sensitive evaluation, [2-89](#)
 - evaluating, [2-89](#)
 - in an Expressions window, [2-92](#)
 - register, [2-93](#)
 - regular, [A-36](#), [A-37](#), [A-38](#), [A-39](#)
 - tagged, [A-39](#), [A-40](#)
 - types of, [2-92](#)
 - use of, [2-93](#)
 - viewing value of, [2-89](#)
 - window, [2-55](#)
- Expressions window, [2-55](#)
- extensions, DSP project file, [A-21](#)
- external interrupts, generating, [1-15](#)
- eye diagrams, [3-13](#)
 - example of, [3-13](#)
 - FIFO, [3-13](#)
- EZ-ICE target, [3-2](#)
- F
- features
 - project build, [1-4](#)
 - project management, [1-4](#)
- file and tool options, [1-20](#)
- file building options, [1-20](#)
- file tree, [2-15](#)
 - icons, [2-15](#)
 - Project window, [2-15](#)
- file types, [A-21](#)
- files, [2-22](#)
 - .ASM, [2-20](#)
 - .C, [2-20](#)
 - .CPP, [2-20](#)
 - .CXX, [2-20](#)
 - .DLB, [2-20](#)
 - .DOJ, [1-23](#), [1-25](#), [1-26](#), [2-20](#)
 - .DPJ, [1-45](#)
 - .DSP, [2-20](#)
 - .DXE, [1-26](#)
 - .H, [1-32](#)
 - .LDF, [1-25](#), [1-26](#), [1-27](#), [1-34](#), [2-20](#)
 - .S, [1-32](#), [2-20](#)
 - .VPS, [2-95](#)
- assembler, [1-25](#)
- associations with tools, [2-19](#)
- automatic placement, [2-20](#)
- boot-loadable, [1-32](#), [1-33](#)
- building, [1-55](#)

- compiler, [1-23](#)
 - data, [1-25](#)
 - executable, [1-27](#)
 - extensions, [A-21](#)
 - header, [1-25](#)
 - in a project, [2-19](#)
 - language, [1-25](#)
 - linker, [1-26](#), [1-27](#), [1-34](#)
 - log, [2-40](#), [2-41](#)
 - nested folders in Project window, [2-18](#)
 - object, [1-25](#), [1-26](#)
 - overlay, [1-26](#)
 - placement rules, [2-21](#)
 - placing into folders automatically, [2-18](#)
 - PROM, [1-32](#), [1-33](#)
 - specifying build settings, [1-52](#)
 - used by DSP projects, [A-21](#)
 - vd_k_config.cpp, [2-22](#)
 - vd_k_config.h, [2-22](#)
 - VisualDSP_Log.txt, [2-41](#)
 - finding
 - (and replacing) tagged expressions, [A-40](#)
 - regular expressions in find/replace operations, [A-36](#)
 - Flash Programmer
 - flash devices, [3-35](#)
 - flash driver, [3-37](#)
 - functions, [3-36](#)
 - interface window, [3-37](#)
 - window controls, [3-39](#)
 - Flash Programmer window, [3-37](#)
 - floating, [2-46](#)
 - toolbars, [2-10](#)
 - window commands, [2-43](#)
 - windows, [2-44](#), [2-46](#), [2-47](#)
 - focus
 - multiprocessor debug session, [2-78](#)
 - pinning, [3-5](#)
 - window, [3-5](#), [3-6](#)
 - folders
 - automatic file placement, [2-20](#)
 - in the Project window, [2-15](#), [2-18](#)
 - project, [2-18](#)
 - functions, displaying local variables, [2-57](#)
- ## G
- General page, [1-21](#)
 - generating external interrupts, [1-15](#)
 - global build options, [1-20](#)
 - glossary, [A-2](#)
 - groups, multiprocessor, [2-78](#), [2-81](#)
- ## H
- hardware breakpoints, [3-21](#)
 - hardware simulation, [1-15](#)
 - header files, [1-25](#)
 - heaps, usage in Expert Linker, [1-30](#)
- ## I
- icons
 - Project window, [2-15](#), [2-24](#)
 - icons, Project window, [2-24](#)
 - idde.exe, command-line parameters, [A-29](#)

INDEX

- IDL (*see* Interface Definition Language)
- Image Viewer, [2-103](#), [3-35](#)
 - Export Image dialog box, [2-108](#)
 - Gamma Correction dialog box, [2-107](#)
 - Image Configuration dialog box, [2-106](#)
 - right-click menu, [2-105](#)
- Interface Definition Language (IDL), [1-42](#)
- interrupts, [3-22](#)
 - generating, [1-15](#)
 - hardware simulation, [1-15](#)
- issuing, Tcl commands, [B-3](#)
- J**
- JTAG emulator, [3-2](#)
 - breakpoints, [3-20](#)
 - debug session management, [3-3](#)
 - debug sessions, [3-3](#)
 - platforms, [3-2](#)
 - sampling, [3-8](#)
 - statistical profiling, [2-58](#), [3-7](#)
- K**
- kernel (*see* VDK)
- Kernel tab, [2-22](#)
- keyboard shortcuts, [A-23](#)
- L**
- libraries, C++ run-time, [1-24](#)
- lindex command, [B-44](#)
- line plots, [3-10](#)
- linear profiling, [3-7](#)
- Linear Profiling Results window, [2-58](#)
- linker
 - input files, [2-19](#)
 - overview, [1-26](#)
- Linker Description File, [1-27](#)
- linking, object files, [1-26](#)
- loader, [1-33](#)
- loading, programs, [3-18](#)
- local build options, [1-20](#)
- Locals window, [2-57](#)
- locating, text using regular expressions, [A-36](#)
- log file, [2-40](#)
 - error messages, [2-49](#)
 - Tcl output, [B-2](#)
- logging error messages, [2-49](#)
- M**
- makefiles, [1-47](#)
 - example makefile, [1-49](#)
 - Output window, [1-48](#)
 - rules, [1-48](#)
- managing
 - debug sessions, [3-3](#)
 - projects, [1-4](#)
 - source files, [1-4](#)
- MDI child windows, [2-43](#)
- memory
 - plots from, [3-9](#)
 - windows, [2-68](#)
- Memory Map window, [2-74](#)
- memory windows, [2-68](#), [2-70](#)
 - customization, [2-73](#)
 - examples, [2-68](#)
- menu bar, [2-6](#)

- menus
 - application menu bar, 2-6
 - control, 2-4
 - control menu, 2-5
 - messages written to VisualDSP_Log.txt
 - file, 2-40
 - mixed mode, 2-91
 - editor window, 2-90
 - examples, 2-91
 - vs. source mode, 2-90
 - modes, 2-90
 - mixed, 2-90, 2-91
 - source, 2-90
 - multiprocessor, 3-3
 - (*see* multiprocessor debug sessions)
 - groups, 2-80, 2-81
 - window tabs, 2-80
 - multiprocessor debug sessions, 3-3
 - debugging, 3-4, 3-5
 - focus, 3-5, 3-6
 - Multiprocessor window, 2-78
 - Multiprocessor window
 - Groups tab, 2-80
 - Status tab, 2-80
- N**
- nested folders, 2-18
 - nodes, in Project window, 2-15
- O**
- object files, 1-25
 - operations
 - program execution, 3-18
 - program execution commands, 3-18
 - options
 - compiler, 1-23
 - file and tool, 1-20
 - file building, 1-20
 - project building, 1-20
 - Output window, 2-31
 - Build tab, 2-32
 - capture of all messages, 2-40
 - Console tab, 2-32
 - customization, 2-41
 - right-click menu, 2-42
 - overlays, files, 1-27
 - overriding, project-wide options, 1-54
- P**
- platforms, DSP configuration, 1-14
 - plot windows, 2-94, 2-95, 2-99
 - capabilities, 2-95
 - creating a new window, 2-99
 - features, 2-96
 - presentation of, 2-99
 - right-click menu, 2-95, 2-96
 - See also* plots
 - status bar, 2-95, 2-97
 - plots
 - 3-D waterfall, 2-99, 3-14
 - configuration of, 2-99
 - constellation, 3-12
 - data sets, 2-99
 - DSP memory, 3-9
 - eye diagram, 3-13
 - line, 3-10
 - presentation options, 2-102
 - See also* plot windows

INDEX

- spectrogram, [3-16](#)
 - types of, [3-9](#)
 - waterfall, [3-14](#)
 - plotting, DSP memory, [3-9](#)
 - position rules, in a window, [2-47](#)
 - positioning, windows, [2-47](#)
 - post-build options, [1-55](#)
 - preferences
 - IDL font and color for editing, [1-38](#)
 - load file and advance to main, [1-21](#)
 - VisualDSP++ and tool output color, [2-32](#)
 - Preferences dialog box, [1-21](#), [2-32](#)
 - presentation, of plot windows, [2-99](#)
 - procedures for development, setting
 - custom build project options, [1-20](#)
 - profiles
 - code analysis, [3-7](#)
 - statistical, [3-7](#)
 - profiling, [3-7](#), [3-8](#)
 - linear, [3-7](#), [3-8](#)
 - statistical, [3-7](#), [3-8](#)
 - program
 - execution commands, [3-18](#)
 - execution operations, [3-18](#)
 - restart, [3-19](#)
 - project
 - build, [1-4](#), [1-46](#)
 - build settings, [1-54](#)
 - building options, [1-20](#)
 - configurations, [1-52](#)
 - debugging, [1-5](#), [1-15](#)
 - defined, [1-45](#)
 - dependencies, [1-19](#)
 - files, [2-19](#)
 - folders, [2-15](#)
 - management, [1-4](#)
 - nodes, [2-15](#)
 - options, [1-46](#)
 - subfolders, [2-15](#)
 - VisualDSP++, [1-45](#)
 - window, [2-15](#), [2-18](#)
 - Project window, [2-15](#), [2-22](#)
 - about, [2-15](#)
 - files, [2-15](#)
 - icon right-click menus, [2-24](#)
 - Kernel tab, [2-22](#)
 - Kernel view, [1-17](#)
 - nodes, [2-16](#), [2-17](#)
 - Project tab, [2-16](#)
 - rules, [1-57](#)
 - use of folders, [2-18](#)
 - project-wide file and tool options, [1-20](#)
 - PROM files, [1-32](#), [1-33](#)
 - pull-tabs, [2-44](#)
- ## R
- register windows
 - custom, [2-77](#)
 - regular expressions, [A-36](#), [A-37](#), [A-38](#), [A-39](#)
 - release configuration, [1-52](#)
 - restarting
 - program during emulation, [3-19](#)
 - program during simulation, [3-19](#)
 - right-click menus, [2-43](#)
 - commands, [2-43](#)
 - in plot windows, [2-95](#)

S

scripting, Tcl, [B-1](#)

scroll bars, descriptions of, [2-44](#)

searches

normal, [A-36](#)

regular expressions vs. normal, [A-36](#)

special character rules, [A-38](#)

searches, normal, [A-36](#)

sessions

debug, [3-3](#)

selecting at startup, [3-17](#)

setting

build options, [1-54](#)

custom build options, [1-20](#)

shadow write FIFO anomaly, [3-27](#)

short word anomaly, [3-24](#)

shortcut keys (*see* keyboard shortcuts)

simulating

data I/O streams, [3-22](#)

hardware, [1-15](#)

input/output data, [3-22](#)

interrupts, [1-15](#)

simulation, [3-7](#)

debug session management, [3-3](#)

linear profiling, [3-7](#), [3-8](#)

options, [3-23](#)

platforms, [1-14](#)

restarting the program, [3-19](#)

SPI in slave mode, [3-32](#)

tools, [3-22](#)

single stepping, available commands, [3-18](#)

source files, [1-3](#)

comments, [A-41](#)

editing features, [1-3](#)

in a project, [2-19](#)

management, [1-4](#)

source mode, editor windows, [2-90](#)

source windows (*see* editor windows)

spectrogram plots, [3-16](#)

example of, [3-16](#)

FFT output, [3-16](#)

SPI simulation in slave mode, [3-32](#)

splitter, [1-32](#)

Stack windows, [2-77](#)

stacks, usage in Expert Linker, [1-30](#)

Statistical Profiling Results window, [2-58](#), [2-59](#), [2-60](#)

statistical profiling, vs. linear profiling, [3-7](#)

status bar, [2-13](#), [2-14](#), [2-95](#)

examples, [2-13](#)

in plot windows, [2-95](#)

status icons, editor window, [2-89](#)

status messages, log file, [2-40](#)

stepping, available commands, [3-18](#), [3-19](#)

steps, development

add and edit project source files, [1-18](#)

build a debug version of the project, [1-21](#)

build a release version of the project, [1-21](#)

create a project, [1-18](#)

set project options, [1-18](#)

streams, [3-22](#)

subfolders, in the project tree, [2-15](#)

INDEX

symbols

Disassembly window, [2-54](#)

editor window, [2-89](#)

T

Target Load window, [2-86](#)

Tcl, [B-1](#), [B-2](#), [B-3](#)

about, [B-1](#)

commands, [B-1](#)

issuing commands, [B-3](#)

from a command line, [B-3](#)

from a user tool, [B-5](#)

from an editor window, [B-5](#)

from the File menu, [B-4](#)

from the Output window, [B-3](#)

output, viewing, [B-2](#)

overview of, [B-1](#)

scripting, [B-1](#)

See also Tcl commands

Tcl commands, [B-1](#)

dspaddmenuitem, [B-20](#)

dspaddstream, [B-22](#)

dspcancelbreak, [B-24](#)

dspcheckmenuitem, [B-25](#)

dspclickmenuitem, [B-26](#)

dspdeleteallstream, [B-27](#)

dspdeletemenuitem, [B-28](#)

dspdeletestream, [B-29](#)

dspenablemenuitem, [B-30](#)

dspeval, [B-31](#)

dspgetbreak, [B-33](#)

dspgetmemblock, [B-35](#)

dspgetmeminfo, [B-37](#)

dspgetprocessors, [B-38](#)

dspgetstate, [B-39](#)

dspgetswstack, [B-40](#), [B-62](#)

dsphalt, [B-41](#)

dspliststream, [B-42](#)

dspload, [B-43](#)

dsplookupline, [B-44](#)

dsplookupsymbol, [B-45](#)

dspmemorywin, [B-46](#)

dspplotrotate, [B-48](#)

dspplotwin, [B-49](#)

dspprojectaddfile, [B-53](#)

dspprojectaddfolder, [B-54](#)

dspprojectbuild, [B-55](#)

dspprojectclose, [B-56](#), [B-66](#)

dspprojectinfo, [B-57](#)

dspprojectload, [B-58](#)

dspprojectremovefile, [B-59](#)

dspprojectremovefolder, [B-60](#)

dspregisterwin, [B-61](#)

dspreset, [B-62](#)

dsprestart, [B-63](#)

dsprun, [B-64](#)

dspset, [B-65](#)

dspsetbreak, [B-66](#)

dspsetmemblock, [B-68](#)

dspsetswstack, [B-70](#)

dspstepasm, [B-71](#)

dspstepin, [B-72](#)

dspstepout, [B-73](#)

dspstepover, [B-74](#)

dspwaitforhalt, [B-75](#)

See also Tcl

technical support, [xxiii](#)

terms, VisualDSP++, [A-2](#)

- threads, [2-82](#)
 - idle, [2-86](#)
 - status, [2-82](#), [2-85](#)
- title bar, [2-4](#)
 - components, [2-3](#)
 - right-click menu commands, [2-43](#)
- Tool Command Language (Tcl). *See* Tcl
- toolbars, [2-7](#), [A-30](#)
 - built-in, [2-8](#)
 - button appearance, [2-10](#)
 - customization, [2-9](#)
 - docked versus floating, [2-10](#)
 - shape, [2-12](#)
- tools
 - access to, [1-3](#)
 - code development, [1-2](#), [2-19](#)
 - command line access, [1-46](#)
 - input files, [2-19](#)
 - project options, [1-46](#)
 - third-party, [1-2](#)
 - user configured, [2-13](#)
- Tools menu, user tools, [2-13](#)
- traces, [2-56](#)
- U
 - unconditional breakpoints, [3-21](#)
 - user interface, parts of, [2-1](#)
- V
 - variables, global vs. local, [2-93](#)
 - VCSE, [1-35](#)
 - component manager, [1-39](#)
 - component model, [1-36](#)
 - components, [1-35](#), [1-37](#)
 - overview, [1-35](#)
 - structure of, [1-40](#)
 - tool chain integration, [1-38](#)
 - tools, [1-37](#)
 - user interface, [1-38](#)
 - wizards, [1-39](#)
 - VDK, [1-17](#), [2-22](#), [2-82](#), [2-87](#)
 - about, [1-17](#)
 - features, [1-6](#)
 - Kernel view in Project window, [2-22](#)
 - overview of, [1-6](#)
 - State History window, [2-84](#)
 - VDK Status window, [2-82](#)
 - VDK State History window, [2-84](#)
 - VDK Status window, [2-82](#)
 - vd_k_config.cpp, [2-22](#)
 - vd_k_config.h, [2-22](#)
 - VisualDSP ICE Configurator, [3-4](#)
 - VisualDSP++
 - control menu, [2-5](#), [2-6](#)
 - debugging, [1-5](#), [1-15](#)
 - editing features, [1-3](#)
 - editor, [2-28](#)
 - editor windows, [2-28](#)
 - environment, [1-2](#)
 - file association for tools, [2-19](#)
 - files, [A-21](#), [A-22](#)
 - glossary, [A-2](#)
 - kernel, [1-17](#)
 - keyboard shortcuts, [A-23](#)
 - log file, [2-40](#)
 - menu bar, [2-6](#), [2-7](#)
 - overview, [1-1](#)
 - parts of, [2-2](#)

INDEX

- parts of the user interface, [2-1](#)
- project, [1-45](#)
- project build features, [1-3](#)
- Project window, [2-15](#), [2-17](#)
- purpose, [1-3](#)
- source file editing features, [1-3](#)
- toolbar, [A-30](#)
- tools - file association, [2-19](#)
- windows, [2-15](#)

W

- watchpoints, [3-21](#)
- waterfall plots, [3-14](#)
 - grid of sampled data, [3-15](#)
 - rotating, [3-14](#)
- windows
 - Call Stack, [2-67](#)
 - debugging, [2-49](#)
 - Disassembly, [2-51](#), [2-52](#), [2-53](#)
 - docked, [2-44](#), [2-45](#)
 - editor, [2-87](#)
 - Expert Linker, [1-29](#)
 - Expressions, [2-55](#)
 - Flash Programmer, [3-37](#)
 - Image Viewer, [2-104](#)
 - Linear Profiling Results, [2-58](#)
 - Locals, [2-57](#)

- manipulation of, [2-43](#)
- MDI, [2-43](#)
- memory, [2-68](#)
- Memory Map, [2-74](#)
- Multiprocessor, [2-78](#)
- Output, [2-31](#), [2-42](#)
- parts of the user interface, [2-1](#)
- plot, [2-94](#)
- Project, [2-15](#), [2-16](#), [2-18](#)
- pull-tabs, [2-44](#)
- Register, [2-75](#)
- right-click menu commands, [2-43](#)
- rules for positions, [2-47](#)
- scroll bars, [2-44](#)
- See also* VisualDSP++
- source, [2-28](#)
- stack, [2-77](#)
- Statistical Profiling Results, [2-58](#)
- Target Load, [2-86](#)
- Trace, [2-56](#)
- VDK State History, [2-84](#), [2-86](#)
- VDK Status, [2-82](#)
- Windows buttons, [2-48](#)

X

- X-Y plots, [3-11](#)