VisualDSP++™ 3.5 User's Guide for 16-Bit Processors

Revision 1.0, October 2003

Part Number
82-000035-06

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

CONTENTS

PREFACE

REFERENCE INFORMATION

SIMULATION OF BLACKFIN PROCESSORS

PREFACE

Thank you for purchasing VisualDSP++™, the development software for Analog Devices processors.

Purpose of This Manual

The *VisualDSP++ 3.5 User's Guide for 16-Bit Processors* describes the features, components, and functions of VisualDSP++. Use this guide as a reference for developing programs for Blackfin® and ADSP-21xx processors.

The User's Guide does not include detailed procedures for building and debugging projects. For how-to information, refer to the VisualDSP++ online Help and the *VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors*.

Intended Audience

This manual is primarily intended for digital signal processing (DSP) programmers who are familiar with Analog Devices processors, but are unfamiliar with the VisualDSP++ environment. The manual assumes that you have a working knowledge of your processor's architecture and instruction set. If you are unfamiliar with Analog Devices processors, you should supplement this manual with other texts (such as the Hardware Reference and Instruction Set Reference manuals that describe your target's architecture and instruction set).

Manual Contents

This manual consists of:

- Chapter 1, “Introduction” – describes VisualDSP++ features, license management, project development, code development tools, VCSE, and DSP projects; also provides a brief introduction to the VisualDSP++ Help system
- 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, plots, and Flash Programmer
- Appendix A, “Reference Information” – provides a glossary and information about file types, keyboard shortcuts, command-line parameters, scripting, toolbar buttons, and text operations; also provides details about online Help features and operations
- Appendix B, “Simulation of Blackfin Processors” – provides an overview of peripheral support in the Blackfin simulators and describes limitations of the simulation software models, simulator instruction timing analysis, and compiled simulation
- Appendix C, “Simulation of ADSP-21xx Processors” – provides an overview of peripheral support in the ADSP-21xx simulators and information about simulating the General Purpose I/O, Host Port Interface, Serial Peripheral Interface, Serial Port, UART Port, Timer, and Memory DMA; also describes simulator instruction timing analysis and boot simulation

What's New in This Manual

The *VisualDSP++ 3.5 User's Guide for 16-Bit Processors* supports all Blackfin and ADSP-21xx processors, including the new ADSP-BF561 Blackfin processor. This edition also documents new features and enhancements.

Technical or Customer Support

You can reach DSP Tools Support in the following ways.

- Visit the DSP Development Tools Web site 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.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA
  ```

Supported Processors

The name "*Blackfin*" refers to the family of Analog Devices 16-bit, fixed-point digital signal processors. VisualDSP++ currently supports the following Blackfin processors.

ADSP-BF531	ADSP-BF532 (formerly ADSP-21532)
ADSP-BF533	ADSP-BF535 (formerly ADSP-21535)
ADSP-BF561	AD6532

VisualDSP++ currently supports the following ADSP-21xx processors.

ADSP-2181	ADSP-2191
ADSP-2183	ADSP-2192-12
ADSP-2184/84L/84N	ADSP-2195
ADSP-2185/85L/85M/85N	ADSP-2196
ADSP-2186/86L/86M/86N	ADSP-21990
ADSP-2187L/87N	ADSP-21991
ADSP-2188L/88N	ADSP-21992
ADSP-2189M/89N	

Product Information

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

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

MyAnalog.com

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

Registration:

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

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

DSP Product Information

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

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

- Email questions or requests for information to dsp.support@analog.com
- Fax questions or requests for information to **1-781-461-3010** (North America) **+49 (0) 089 76 903 157** (Europe)
- Access the Digital Signal Processing Division's FTP Web site at ftp ftp.analog.com or **ftp 137.71.23.21** ftp://ftp.analog.com

Related Documents

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

VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors

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

VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x Processors

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for ADSP-219x Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for Blackfin Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for ADSP-21xx Processors

VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors

VisualDSP++ 3.5 Loader Manual for 16-Bit Processors

VisualDSP++ 3.5 Product Bulletin

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

For hardware information, refer to your processor'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.

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 VisualDSP\Help 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 Docs folder. Viewing and printing a .PDF file requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). Running setup.exe 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 Docs\Reference 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.

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

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 **Contents**, **Search**, or **Index** from the VisualDSP++ **Help** 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.

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.

Using 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.

Using the Windows Start Button

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

From the Web

To download the tools manuals, point your browser at:

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.

Contacting DSP Publications

Please send your comments and recommendations for improving our manuals and online Help. You can contact us by sending an email to:

```
dsp.techpubs@analog.com
```

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 `this` or `that`.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, code examples, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	A note providing information of special interest or identifying a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	A caution providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word **Caution** appears instead of this symbol.

1 INTRODUCTION TO VISUALDSP++

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

VisualDSP++ includes:

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

This chapter contains the following topics.

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 that you purchased, VisualDSP++ includes one or more of the following components.

- C/C++ compiler with run-time library
- Assembler, linker, preprocessor, and archiver
- Loader and splitter
- Simulator
- EZ-KIT Lite development system (must be purchased separately)
- Emulator (must be purchased separately)

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, or linker 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 1-33.

- **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.
- **View error details and offending code.** From the **Output** window's **Build** view, display error details by highlighting the error code (such as cc0251) and pressing the **F1** key. Double-click an error line 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 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. It also provides a source code control (SCC) interface, which enables you to access SCC applications without leaving the IDDE.

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 command-line scripts.** Use scripts to customize key debugging features.
- **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.
- **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.
- **Generate interrupts using streaming I/O.** Set up serial port (SPORT) or memory-mapped I/O.
- **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.

- **Trace program execution history.** Trace how your program arrives at a certain point and show reads, writes, and symbolic names.
- **View pipeline depth of assembly instructions**. Display the pipeline stage by querying the target processor or processors through the pipeline interface (not supported on the ADSP-218x processor).

For details, see the *VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors*.

VDK Features

The VisualDSP++ Kernel (VDK) is a scalable software executive specially developed for effective operations on Analog Devices 16-bit processors and tightly integrated with VisualDSP++.

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

The kernel provides all the basic building blocks required for application development. Properties of the kernel can be characterized as follows.

- **Automatic**. VisualDSP++ automatically generates source code framework for each user-requested object in the user-specified language.
- **Deterministic**. VisualDSP++ specifies whether the execution time of a VDK API is deterministic.
- **Multitasking.** Kernel tasks (threads) are independent of one another. Each thread has its own stack.
- **Modular**. The kernel comprises various components. Future releases may offer additional functionality.

- **Portable**. Most of the kernel components can be written in ANSI Standard C or C++ and are portable to other Analog Devices processors.
- **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**. The kernel provides run-time error checking.
- **Scalable**. If a project does not include a kernel feature, the support code is not included in the target system.

VisualDSP++ 3.5 Features

VisualDSP++ 3.5 includes the following new features and enhancements.

- **New processor support**. The ADSP-BF561 processor is supported by this software version. Refer to the *ADSP-BF561 Blackfin Hardware Reference* and chip data sheet for details.
- **Multiple project support**. VisualDSP++ provides the ability to switch among multiple open projects in the same IDDE session. The **Project** window displays active projects.
- **Data streaming and logging**. VisualDSP++ now offers the ability to stream and log data from a target DSP without halting the DSP. The IDDE takes advantage of this capability in plot windows. If the target supports background telemetry channel (BTC), the plot window is updated while the target is running.

- **License management in the IDDE**. License management (installation and validation) has been integrated into the VisualDSP++ IDDE. Installing a FlexLM license server is still handled by the separate installation application.

- **Profile-guided optimization (PGO) in the IDDE**. The VisualDSP++ IDDE includes facilities to run common PGO scenarios simply and also provides a mechanism for advanced applications that require more control over the profiling process via scripting. The PGO process involves setting up and executing data sets to produce an optimized application. A *data set* is the association of zero or more input streams with one `.pgo` output file.

  The most common scenario for collecting PGO data is setting up one or more simple *File to Device* streams. The *File* is a standard ASCII stream input file, and the *Device* is any stream device (such as memory or a peripheral) supported by the simulator target. You create, edit, and delete data sets in VisualDSP++ and then run them to optimize your application. For PGO operations, the **Tools** menu provides a new **PGO** submenu and a **Manage Data Sets** option.

  Note that each compiler manual associated with VisualDSP++ 3.5 has a new chapter called "Achieving Optimal Performance from C/C++ Source Code." Its focus is to help you maximize code performance from the compiler while you minimize code size. The chapter starts by discussing some general optimization principles and how the compiler can most help your optimization effort. Optimal coding styles are then considered in detail. Special features such as compiler switches, built-in functions, and pragmas are also discussed. The chapter ends with a short example that demonstrates how the optimizer works.

  Additional information about PGO is in the *VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors* and in the online Help.

- **Integrated Source Code Control (SCC)**. VisualDSP++ uses the Microsoft Common Source Code Control (MCSCC) interface to provide a connection from the IDDE to SCC applications (such as Visual SourceSafe, PVCS Version Manager, and ClearCase) installed on your machine. You can now conveniently access commonly-used SCC features from VisualDSP++ without leaving the IDDE. Advanced and application-specific SCC features not available from the IDDE must be run directly from the SCC applications.

- **Automation aware scripting engine**. VisualDSP++ includes a scripting engine that uses the Microsoft ActiveX script host framework. The engine enables you to use multiple scripting languages (such as VBScript, JavaScript, and so on) to access the VisualDSP++ Automation API.

  You can interact with the IDDE by using a single command or a script file similar to the Tcl scripting functionality, which was available in previous versions of VisualDSP++.

- **Profiling code with Expert Linker**. You can use Expert Linker to profile object sections in a program. When the program halts, Expert Linker graphically displays how much time was spent in each object section. You can use this display to locate code "hotspots" and then move that code to faster, internal memory.

- **Address bar in Disassembly and Memory windows**. When enabled, an address bar is displayed in **Disassembly** windows and memory windows. You can use the address bar to navigate by address, symbol, or expression. The address bar maintains a most recently used history of visited locations.

- **Menus with Icons**. Icons now appear beside menu commands that have corresponding toolbar buttons.

License Management

VisualDSP++ is a licensed software product. This section describes licensing options, license status, license installation, software registration, validation codes, product updates, and product serial numbers.

Licensing Options

Two licensing options are available: *single-user* and *client*. A *server* license is required before you can install a client license (see Table 1-1).

Table 1-1. VisualDSP++ Licenses

License	Description
Single-User	Also called *node-locked* or *per-user* licenses, they are locked to the machine ID of the host computer. Once installed, these tools will run only on that one machine. You may install and register the software up to three times (for example, at work, at home, and on a laptop computer). Use, however, is restricted to one installed software at a time.
Client	Client licenses are a client/server-based application. The server manages a pool of licenses installed on the server. One license is installed on the server for each purchased copy of VisualDSP++. In this model, you can have as many client installations as desired. When a client starts the software, it checks out a license from the server. When the software exits, the license is returned to the server. As long as licenses are available on the server, clients can access VisualDSP++. **Example:** Assume a license server has been set up with 10 licenses, and 20 client machines are installed in three labs. Ten simultaneous developers (any combination) can use the software. When the 11th client tries to use VisualDSP++, a message appears, stating that no more licenses are available. This allows sharing of the software resources in an environment that needs more locations than developers.
Server	Allows multiple users to access VisualDSP++ on computers sharing "client licenses" across a network. A server license must be installed prior to installing client licenses.

Server-based *floating licenses* consist of two parts: server and client. The server manages the license pool that is stored on the server. The clients "check-out" licenses when the software is started and return licenses to the server when developers exit VisualDSP++.

License Status

The **Licenses** page of the **About VisualDSP++** dialog box displays the status of all recognized licenses.

The status for each serial numbered product can be:

- Validated (Permanent)
- Not Validated
- Test Drive
- For EZ-KIT Lite: Permanent, Restricted; Expiring in X days; or Expired

Temporary Licenses

A temporary license indicates the number of days remaining before you can no longer use the product. Test drive versions of VisualDSP++ (serial number beginning with "TST") carry temporary licenses.

An unrestricted version of VisualDSP++ includes its permanent license. If you do not install the validation code after purchasing a full (unrestricted) license, the status of the license is marked "Not Validated (Expiring in X days)." Install the validation code to change the status to "Permanent."

Valid vs. Expired Licenses

An expired license is indicated by "Expired". A valid license is indicated by "Permanent" unless it is for an EZ-KIT. A valid license for an EZ-KIT is indicated by "Permanent, Restricted".

Client Licenses

When a client license is installed, the `server_name` appears under **Serial Number**, "client" appears under **Family**, and "`use_server`" appears under **Status**.

License Installation

After installing VisualDSP++, you have to license the software. Licensing involves these three tasks:

- Installing the single-user or client serial number

  Note that a server license must be installed before you can install client licenses.

- Registering products
- Entering validation codes

You perform license management activities within VisualDSP++ by using the **About VisualDSP++** dialog box. See the online Help for detailed installation, registration, and validation procedures.

Test drives require online registration to receive a "TST" serial number, which expires 90 days after installation. Test drives do not require a validation code.

As of VisualDSP++ 3.5, EZ-KIT Lite versions of VisualDSP++ require online registration and a validation code. The "KIT" serial number is on the label on the back of the CD wallet.

"KIT" serial numbers impose restrictions on VisualDSP++. These limitations do not prevent processor evaluation on the EZ-KIT Lite evaluation board, but they encourage the purchase of a full (unrestricted) VisualDSP++ license.

Installing a Single-User License

VisualDSP++ must be licensed. A single-user (per-user, node-locked) license allows VisualDSP++ to be used on one computer. Installing the serial number creates a `license.dat` file.

This procedure requires a serial number, which is located on the software registration card or on the CD's sleeve.

This procedure assumes you have installed VisualDSP++ from the CD and have installed the latest service pack (if applicable). Service packs are available from the Analog Devices Web site at:

`http://www.analog.com/dsp/tools/fixes.html`

If the installation is successful, a "success" message appears. Additional information appears, depending on the installed license. At this point, the software has a temporary, 30-day license.

For serial numbers that begin with "ADI", "ENG", or "KIT", a message instructs you to register the serial number to receive a validation code. Note that "ENG" licenses are for Analog Devices use only.

For serial numbers that begin with "TST", a message informs you that the license can be installed only once.

You install a single-user license from the **Install New License** dialog box, accessed from the **Licenses** page of the **About VisualDSP++** dialog box.

Installing a Server License

Installing a server license allows multiple users to use VisualDSP++ on computers using "client licenses" across a network. You must install a server license before installing client licenses.

This procedure requires the entry of a server node name, which you can obtain from the **Identification** page of the **Network** applet in Windows **Control Panel**.

The installation procedure uses the FlexLM (Flexible License Manager) network license manager included with VisualDSP++. Flex-LM installs the NT service and path information. Note that when you add a seat or a permanent license, you must restart/reread the FlexLM license manager each time that you update your license file.

You install a server license by running **Install License** and **Install Server License** from the VisualDSP++ CD ROM.

Installing a Client License

A client license allows access to VisualDSP++ over a network.

A server license allows multiple users to use VisualDSP++ on computers using "client licenses" across a network. A server license must be installed before you can install client licenses.

If the installation is successful, a "success" message is displayed.

If a `license.dat` file already exists on the client machine (`VisualDSP\system`), a message indicates that the current `license.dat` file will be renamed to `license.bak` and the server license will be copied to the client machine.

You install a client license from the **Install New License** dialog box, accessed from the **Licenses** page of the **About VisualDSP++** dialog box.

Software Registration

After you install or float a single-user license, you must register the product (via the Analog Devices Web site) within 30 days to obtain a validation code.

"ENG," "ADI," and "KIT" serial numbers require registration. "TST" serial numbers do not require registration.

You register a license from the **Licenses** page of the **About VisualDSP++** dialog box, accessed from the **Help** menu.

Validation Codes

After you successfully register an "ENG," "ADI," or "KIT" serial number on the Analog Devices Web site, you will receive a validation code that you must enter to create a permanent license. If you enter this code successfully, a message indicates that a permanent license has been created.

You enter a validation code from the **Enter Validation Code** dialog box, accessed from the **Licenses** page of the **About VisualDSP++** dialog box.

Product Upgrades

From time to time, Analog Devices releases new software versions.

The upgrade procedure does not change the previous version's folder structure or license file. The new installation process uses the previous version's path and license.

Check the Analog Devices Web site to ensure you have the latest software version.

Product Serial Numbers

Product serial numbers are located on product CD sleeves. You can also view a product's serial number from within VisualDSP++.

If you cannot locate a serial number, contact your local sales representative or Analog Devices sales.

- Send e-mail to: `dsptools.support@analog.com`
- Phone: 1-800-ANALOGD (1-800-262-5643)

Provide details about the exact products, versions, and operating system being used.

Within VisualDSP++, you view product serial numbers from the **Licenses** page of the **About VisualDSP++** dialog box, accessed from the **Help** menu.

Project Development

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

Overview of Programming with VisualDSP++

Programming effectively with VisualDSP++ depends on how well you master a four-step process. You must learn how to:

1. Work with VisualDSP++
2. Implement structured software design with VisualDSP++
3. Optimize performance with VisualDSP++
4. Test and debug your programs with VisualDSP++

Working With VisualDSP++: You should have a working knowledge of VisualDSP++, the front end for all available targets and platforms. You should know how and when to use its various features and have a firm foundation in these project basics:

- Working with property pages. These pages are analogous to command-line switches.
- Setting up debug sessions. Know the distinctions between the three development stages: evaluation (via an EZ-Kit Lite development system), simulation, and emulation.
- Understanding how program sections and memory segments relate to physical DSP memory. Become familiar with the Expert Linker.
- Accessing peripherals. This task includes setting up and handling interrupts in both C and assembly.

Designing Structured Software With VisualDSP++: You should consider elements of software design, code reuse, and interoperability. If you are new to embedded systems, try to acquire a clear understanding of:

- The role of and motivation behind component software
- How to create and use a VCSE component
- The role of an RTOS
- How to use VDK to manage multiple threads of execution and the communication between those threads

Optimizing Performance With VisualDSP++: At this stage, you should understand how to access the features of the DSP and how to use a structured approach to develop software. Next you should optimize your software to take full advantage of the DSP's computational power. This step entails:

- Understanding the compiler optimizer
- Writing mixed C and assembly programs
- Accessing C/C++ data structures in assembly
- Harnessing the power of C++
- Setting up and using overlays
- Configuring emulation L1 memory (Blackfin processors only) for cache vs. SRAM with cache visualization
- Using statistical profiling

Testing and Debugging With VisualDSP++: At this stage, you should have a good understanding of the various facilities available for producing optimal software. The last step is applying software testing and debugging techniques, which include:

- Collecting data and using the advanced plot windows
- Using compiled simulation
- Using ActiveX and COM Automation to create regression test environments and to take advantage of interoperability with other applications

DSP Project Development Stages

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

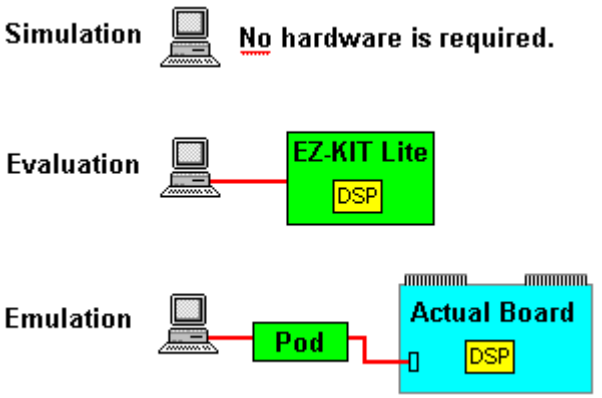

Figure 1-1. Project Development Stages

You use VisualDSP++ during both simulation and emulation.

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. A *simulator* is software that mimics the behavior of a DSP chip. Running VisualDSP++ with a simulation target without a physical processor enables you to build, edit, and debug your DSP program before a DSP chip is manufactured.

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*, the hardware that connects your PC to the actual DSP target board. An emulator enables application software to be downloaded and debugged from within 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 processor (or group of processors). A target can be a simulator, EZ-KIT Lite evaluation board, or an emulator. Your system can include multiple targets.

For example, the JTAG emulator communicates with one or more physical devices over the host PC's PCI bus, and the Apex-ICE™ emulator communicates with a device via the PC's USB port.

Simulation Targets

A *simulation target*, such as the ADSP-BF535 Family Simulator, is a pure software module and does **not** require the presence of a processor for debugging.

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

Compiled simulation is an optional process that converts a .DXE file into an .EXE file, which executes directly on the system hosting VisualDSP++ to increase speed. For details, see "Compiled Simulation" on page B-41.

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 processors 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 1-2. 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
- Data transfer through the processor's I/O pins
- 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 the following table, the check mark ✔ indicates the various debugging tools that you can use while building and debugging your DSP program.

Table 1-3. Tools Used for Simulation and Emulation

Tool	Simulation	Evaluation	Emulation
Linear profiles	✔		
Interrupts	✔		
Streams	✔		
Watchpoints	✔		
Pipelining	✔		

Table 1-3. Tools Used for Simulation and Emulation (Cont'd)

Tool	Simulation	Evaluation	Emulation
Breakpoints	✔	✔	✔
Plotting	✔	✔	✔
Statistical profiles			✔
Hardware breakpoints			✔

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

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 and watchpoints
- View the state of the processor's memory, registers, and stacks
- Perform a cycle-accurate statistical profile or linear profile

VisualDSP++ Kernel

A 16-bit project can optionally include the VisualDSP++ Kernel (VDK), which is a software executive between DSP algorithms, peripherals, and control logic.

The **Project** window's **Kernel** tab accesses a tree control for structuring and scaling application development. From 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
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 "Basic Tutorial" chapter of the *VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors*.

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 you build 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.

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 on 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 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 (`.DXE`) 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

This section describes the following DSP development tools.

- C/C++ compiler with run-time libraries
- Assembler and preprocessor
- Linker
- Expert Linker
- Archiver
- Loader
- Splitter

Available code development tools differ, depending on your processor. The options available on the tab 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 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 `.DOJ` extension.

You specify compiler options for your build by selecting **Project**, **Project Options**, and the **Compile** tab. On the **Compile** page, you then select a **Category** of options.

Compiler options are grouped into the categories described in Table 1-4.

Table 1-4. Groups of Compiler Options

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

The available compile options depend on your target DSP and your code development tools.

For more information, refer to the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual* for your processor.

C++ Run-Time Libraries


You must run VisualDSP++ to use the C++ run-time libraries.

The C and C++ run-time 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. The run-time 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. This release of the compiler software includes both the Standard C Library and 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.

You specify assembler options by selecting **Project**, **Project Options**, and the **Assemble** tab.

Assembler terms are defined as follows.

Instruction set

The set of assembly instructions that pertain to a specific DSP. For information about the instruction set, refer to your processor'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++ 3.5 Assembler and Preprocessor Manual* for your processor.

Linker

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

The linker's output files (.DXE, .SM, .OVL) are binary, executable, and linkable files (ELF). To make an executable file, the linker processes data from a Linker Description File (.LDF) and one or more object files (.DOJ). The executable files contain program code and debugging information. The linker fully resolves addresses in executable files.

You specify linker options by selecting **Project**, **Project Options**, and the **Link** tab. On the **Link** page, you then select a **Category** of options. Linker options are grouped into the following categories.

- General
- LDF Preprocessing
- Elimination
- Processor

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 (.DXE) link against. Through this process, a shared item is available to multiple executables without being duplicated.

Link objects

Object files (.DOJ) that become linked and other items, such as executables (.DXE, .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 run-time 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++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

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
- Defined 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 commands – 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 statement specifies the object file that the linker uses to resolve the mapping.

For details, refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

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
- View how much stack or heap has been used after you run a DSP program

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.

The Expert Linker works with the linker. For more information about linking, refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

Expert Linker graphically displays the available project information in an `.LDF` file as input. This information includes object files, LDF macros, libraries, and target memory descriptions. You can then use the drag-and-drop function to arrange the object files in a graphical memory mapping representation. When you are satisfied with the memory layout, you can generate the executable file (`.DXE`) via VisualDSP++ project options.

You can use default `.LDF` files that come with VisualDSP++, or you can 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 on page 1-38). Use this display 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.

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.

You can launch the Expert Linker from VisualDSP++ in three ways:

- Double-click the `.LDF` file in the **Project** window.
- Right-click the `.LDF` file in the **Project** window to display a menu and then choose **Open in Expert Linker**.
- From the VisualDSP++ main menu, choose **Tools**, **Expert Linker**, and **Create LDF**.

The **Expert Linker** window (Figure 1-2 on page 1-38) is displayed.

Expert Linker Window

The Expert Linker window (Figure 1-2) enables you to modify the memory map or the object mappings. You can specify a color for each type of object (internal memory, external memory, unused memory, reserved memory, output sections, object sections, overlays in live space, and overlays in run space). The objects are displayed in color when you view the **Memory Map** pane in graphical memory map mode. When the project is about to be built, Expert Linker saves the changes to the `.LDF` file.

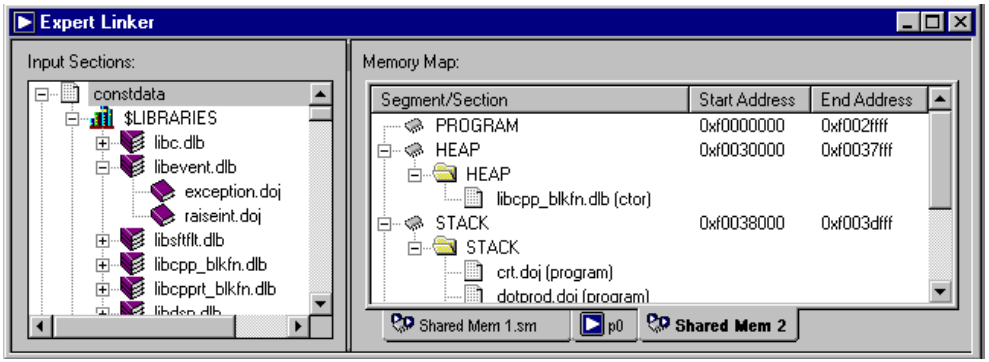

Figure 1-2. Expert Linker Window

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.

Memory Map Pane Right-Click Menu

Table 1-5 describes the commands on the Memory Map pane right-click menu.

Table 1-5. Memory Map Pane Right-Click Menu

Command	Purpose
View Mode→Memory Map Tree	Displays the memory map in tree mode
View Mode→Graphical Memory Map	Displays the memory map in graphical blocks
View→Mapping Strategy (Pre-Link)	Displays the memory map, which shows where you intended to place object sections
View→Link Results (Post-Link)	Displays the memory map, which shows where the object sections are actually placed
New→Memory Segment	Opens the **Memory Segment Properties** dialog box, from which you specify the name, address range, type, width, and so on of the memory segment that you want to add
New→Output Section	Adds an output section to the selected memory segment. **Note**: Right-click on a memory segment to access this command.
New→Shared Memory	Opens the **Shared Memory Properties** dialog box, from which you specify the name of the shared memory output file and processors that share the memory. This command is not available on single-processor systems.
New→Overlay	Opens the **Overlay Properties** dialog box, from which you add a new overlay to the selected output section or memory segment. **Note**: The new overlay's run space is in the selected output section.
Delete	Deletes the selected object
Pin to Output Section	Pins an object section to an output section to prevent it from overflowing to another output section. This command is available only after you right-click on an object section that is part of an output section set to overflow to another section.

Table 1-5. Memory Map Pane Right-Click Menu (Cont'd)

Command	Purpose
View Section Contents	Opens the **Section Contents** dialog box, which displays the contents of the input or output section. This command is available only after you link or build the project and then right-click on an input or object section.
Add Hardware Page Overlay Support	Sets up hardware overlay live and run spaces for all available hardware pages by: a) Checking if memory segments are currently defined in all hardware pages. If memory segments are located, you are queried about whether to delete those segments. b) Creating a memory segment containing an overlay (live space) in each hardware page c) Creating a memory segment containing all overlay run spaces in hardware page 0 d) Creating a default mapping for each overlay. The default mapping maps objects containing the section, "`pmpage0`" to the hardware overlay on PM page 0, "`pmpage1`" to PM page1, "`dmpage0`" to DM page 0, and so on.
View Symbols	Opens the **View Symbols** dialog box and displays the symbols for the project, overlay, or input section. This command is available after you link the project and then right-click on the **Memory Map** pane for a processor, memory segment, output section, or input section.
Expand All	Expands all items in the memory map tree to make their contents visible
View Legend	Opens the **Legend** dialog box, which shows all possible icons in the tree window, with a brief description of each icon. The **Colors** page displays a list of colors used in the graphical memory map. You can specify each object's color.
View Global Properties	Opens the **Global Properties** dialog box for the selected object. The dialog box's title and content depend on the selected object.

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.

For Blackfin DSPs, be aware of the following stack and heap restrictions.

- The heap, stack, and system stack must be defined in output sections named `HEAP`, `STACK`, and `SYSSTACK`, respectively.
- The heap, stack, and system stack must be the only items in those output sections. You cannot place other objects in those output sections.

For other processor families, the restrictions on memory segment names differ according to what is used in the default `.LDF` files. If you do not heed these restrictions, you will not be able to view stack and heap usage after running your program.

Figure 1-3 shows an example memory map after you run a Blackfin program.

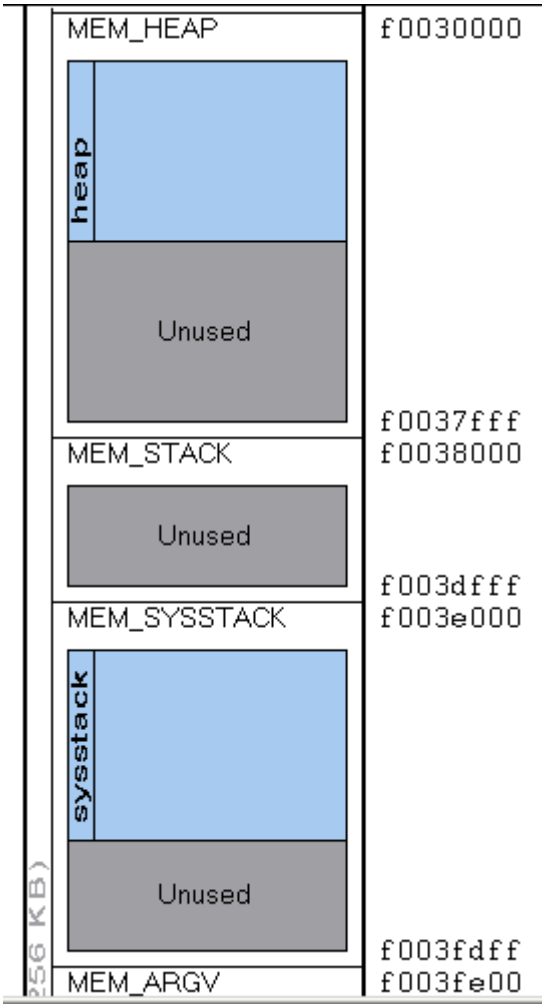

Figure 1-3. Memory Map Example After Running a Blackfin Program

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. For details, refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

Splitter

The splitter processes an executable file to generate a series of non-bootable programmable read-only memory (PROM) files. These files execute from the processor's external memory.

The splitter's primary output is a PROM file with these extensions:

- `.LDR` (Blackfin processors)
- `.S_#`, `.H_#`, and `.STK` (ADSP-219x processors)
- `.BNM`, `.BNU`, and `.BNL` (ADSP-218x processors)

The ADSP-218x splitter (`elfspl21.exe`) can prepare non-bootable PROM image files that execute from DSP external memory. In most instances, use the loader instead of the splitter.

The ADSP-218x splitter must be run from a command line. For details, refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

The splitter generates images for the external PMOVLAY and DMOVLAY memory pages. If you use the splitter to produce ROM images (for example, ADSP-2181 program memory pages 1 and 2), the generated code must target ROM. Define appropriate ROM segments in your `.LDF` file.

The ADSP-219x splitter utility (`elfloader.exe`) can prepare non-bootable PROM image files that execute from DSP external memory. In most instances, use the loader instead of the splitter.


Splitting does not apply to ADSP-2192-12 DSPs.

Splitter capability for Blackfin processors is available from the **Load** page of the **Project Options** dialog box.

For more information about the splitter and options used to generate loader files, refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

Loader

The loader (`elfloader.exe`) generates a boot-loadable file for 16-bit processors by processing executable files. To generate a loadable file, the loader processes data from a boot kernel file (`.DXE`), the linker's executable file (`.DXE`), and in some cases overlay files (`.OVL`). The ADSP-BF532 and ADSP-21xx processors use one on-chip ROM bootstrap kernel for automatic booting from an external memory device. The ADSP-BF535 and ADSP-21xx processors are supported by two booting kernels. Boot loading through the boot kernel is currently supported only on the ADSP-BF535 and ADSP-21xx processors.

Once you have fully debugged your program, use the loader to generate a set of boot-loadable files for your target system. The loader produces one output file with an `.LDR` extension (`.BNM` for ADSP-218x processors) or two output files (an `.LDR` boot kernel file and a `.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.

You specify loader options by selecting **Project**, **Project Options**, and the **Load** tab. On the **Load** page, you then select a **Category** of options.

Loader options are grouped into the following categories.

- Loader options
- Boot kernel options
- ROM splitter options

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

VCSE

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
- Provide support for testing of components as they are developed or used by an application
- Enable the use of components to be optimized automatically
- Encourage third party developers to provide the implementation of algorithms as easily used components

For more information, refer to the *VisualDSP++ 3.5 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 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 in 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 following.

- Relevant object and header files to the project
- 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. It also 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 convenient form 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 supported by a component and generates the framework code needed to implement the component. The developer of the component can thus concentrate on providing the implementation of the methods that the component will 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 five possible output items. These items can be emitted for each interface:

- An interface header file, which 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 run-time 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
- A test shell component, which can be used to do the following automatically: validate the behavior of a component or ensure that an application is using a component properly. In addition to generating checks automatically, you can add additional validation for each method in an interface.

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.

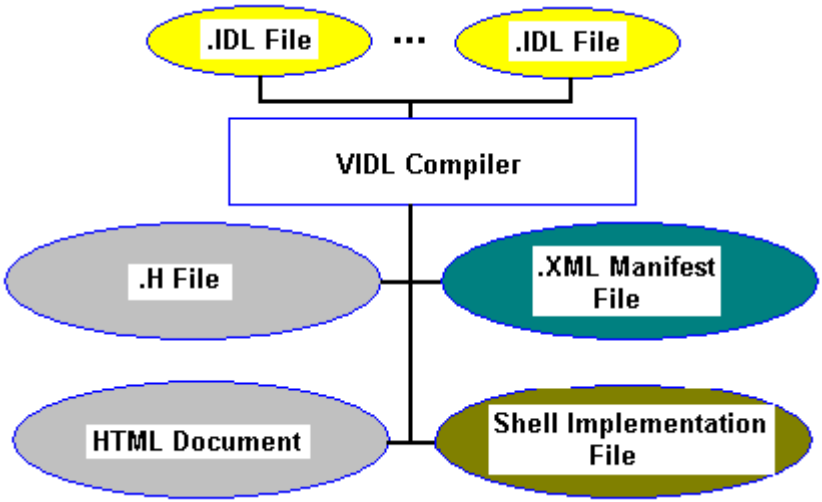

Figure 1-4. Files Produced by the VIDL Compiler

DSP Projects

The project is the structure in which you build the DSP program. VisualDSP++ provides flexibility in how you set up projects. You configure settings for DSP code development tools and configurations, and you 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.

What is a Project?

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

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.

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.

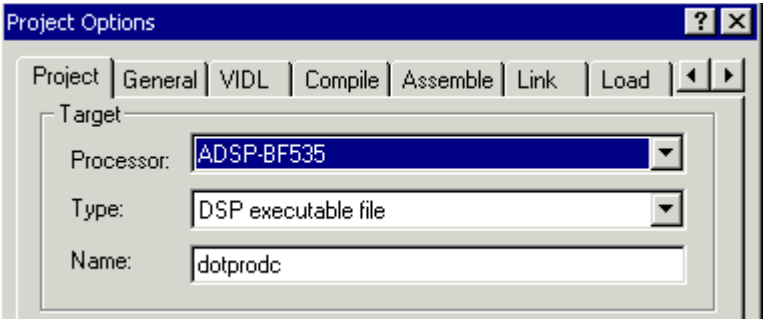

Figure 1-5. Top Portion of the Project Options Dialog Box

For each code development tool (compiler, assembler, linker, splitter 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

Project Groups

Project groups enable you to work with a number of projects at once. A project group can be empty or contain any number of projects. Opening a project adds it to the project group. Closing a project removes it from the project group.

Similar functionality is found in Microsoft Visual Studio. The capabilities provided by project groups are particularly useful in VCSE development, which involves multiple projects.

The **Project** window (Figure 1-6) displays the project group icon and the projects opened in that workspace.

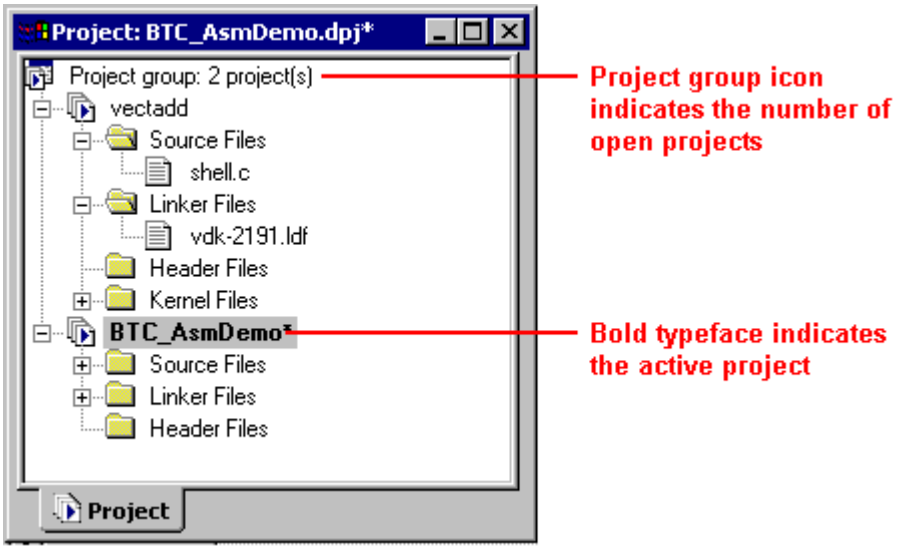

Figure 1-6. Project Window

Each workspace has one project group. When you switch among workspaces, the project group is loaded and the same set of projects are opened just as when you last closed the workspace.

One project is active at a time. The active project responds to commands and messages from menus and tool bars. The **Project** window displays the active project with bold typeface. A **Project** box, located by default with the toolbar buttons, displays the name of the active project (see Figure 1-7).

Figure 1-7. Project Box Showing the Active Project

Though commands are sent to the active project, they may also be carried out by a project on which the active project depends. For example, assume that project A is active and depends on project B. Performing a **Rebuild All** command on project A builds project B first. The same applies to the clean command. Exporting a makefile exports one makefile for each open project. In the makefile of a project depending on another project, one sub-target is created for each project on which it depends. Thus, building a project builds all dependent projects first.

Source Code Control (SCC)

VisualDSP++ includes Source Code Control (SCC), which enables you to use the Microsoft Common Source Code Control (MCSCC) interface to connect between the VisualDSP++ IDDE and the SCC applications installed on your machine.

Various SCC products (such as Visual SourceSafe, PVCS Version Manager, and ClearCase) support the MCSCC interface. Using the convenient VisualDSP++ interface, you can access the commonly used features of these applications without leaving the IDDE. You can launch the SCC application from the plugin menu to use non-supported features.

When you create a project, you are prompted to add the project to SCC. When you open a project in the IDDE, the SCC plugin connects to the selected SCC application and locates a controlled copy of the project and its source files. If a controlled copy is not located, the SCC application must locate it. Typically, you are queried to browse for it. If the controlled copy is successfully found or added, the plugin keeps its application-specific path in the project file and reconnects with this path in the future. You can subsequently reconnect to the controlled copy without having to browse for it.

Operations executed on large number of files tend to take longer to finish. A message box provides status information by displaying the operation currently executing. A button on the message box enables you to cancel the operation. The **Output** window's **Console** view displays finished operations. Messages indicate what has been done. Warnings and error messages may also appear in the **Output** window.

SCC applications provide dialog boxes and GUI displays for some file operations such as show history, show difference, and show properties. You can run these operations from VisualDSP++.

Makefiles

You can use a makefile (`.MAK`) to automate builds within VisualDSP++. The output make rule is compatible with the gnumake 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 1-8 includes an active makefile (indicated by ).

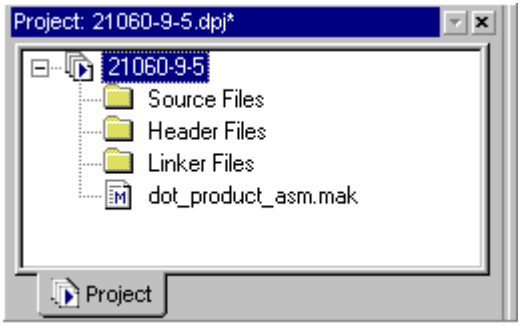

Figure 1-8. Enabled Makefile dot_product_asm.mak

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.

You can view a makefile's command line. To change the makefile's target, use the **Configuration** box, shown in Figure 1-9.

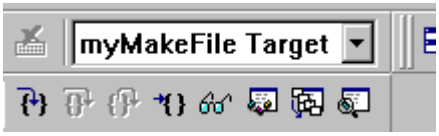

Figure 1-9. Configuration Box

When you close a project, the make commands and the target list associated with each makefile are serialized in 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 appears 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.
# 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
# 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 \,/,$(ADI_DSP))
VDSP=$(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/dot_product_asm.dxe
```

```
./debug/dotprod.doj : ./dotprod.c
$(VDSP)/ccblkfn
-c .\dotprod.c
-g -BF535
-o .\Debug\dotprod.doj -MM
./debug/dotprod_func.doj : ./dotprod_func.asm
$(VDSP)/easmBLKFN.exe -BF535
-o .\Debug\dotprod_func.doj
.\dotprod_func.asm -MM
./debug/dotprod_main.doj : ./dotprod_main.c
$(VDSP)/blackfin/include/stdio.h
$(VDSP)/blackfin/include/yvals.h
$(VDSP)/blackfin/include/stdlib.h
$(VDSP)/blackfin/include/math.h
$(VDSP)/blackfin/include/ymath.h
$(VDSP)/blackfin/include/ccblkfn.h
$(VDSP)/ccblkfn -c .\dotprod_main.c
-g -BF535
-o .\Debug\dotprod_main.doj -MM
./debug/dot_product_asm.dxe :
./debug/dotprod.doj ./debug/dotprod_func.doj
./debug/dotprod_main.doj ./dotprodasm.ldf
$(VDSP)/ccblkfn.exe .\Debug\dotprod.doj
.\Debug\dotprod_func.doj
\Debug\dotprod_main.doj
-T .\dotprodasm.ldf -BF535
-L .\Debug -o .\Debug\dot_product_asm.dxe
-flags-link -MM
endif
ifeq ($(MAKECMDGOALS),Debug_clean)
Debug_clean:
$(RM) ".\Debug\dotprod.doj"
$(RM) ".\Debug\dotprod_func.doj"
$(RM) ".\Debug\dotprod_main.doj"
$(RM) ".\Debug\dot_product_asm.dxe"
$(RM) ".\Debug\*.ipa"
$(RM) ".\Debug\*.opa"
$(RM) ".\Debug\*.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 1-6. 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 1-10.

Figure 1-10. 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 1-11 shows a customized configuration (**Version2**) listed in the configuration box.

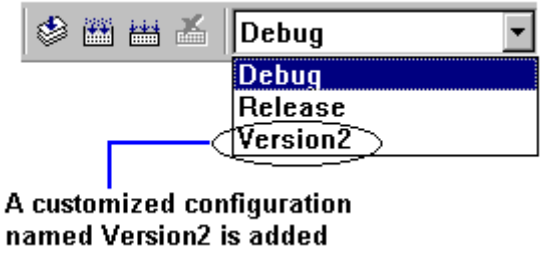

Figure 1-11. Customized Configuration Version2

Project Build

The term *build* refers to the process of performing 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 of 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 1-7 describes these build options.

Table 1-7. 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 1-8. The "`C`" after the slash in the commands must be uppercase.

Table 1-8. 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 are examples of dependency information.

```
./debug/diriirc.doj : ./diriirc.dsp

./debug/setupiir.doj : ./setupiir.dsp

./debug/shell.doj : ./shell.c ./newsigc.dat ./bcoeff.dat
./acoeff.dat

./debug/mixedcandasm.dxe : $(VDSP)/BF535/ldf/adsp-BF535.ldf
./debug/diriirc.doj ./debug/setupiir.doj ./debug/shell.doj
$(VDSP)/BF535/lib/BF535_hdr.doj
$(VDSP)/BF535/lib/BF535_int_tab.doj $(VDSP)/BF535/lib/libc.dlb
$(VDSP)/BF535/lib/libdsp.dlb $(VDSP)/BF535/lib/libio.dlb
```

Project Rules

The **Project** window displays a project's files, as shown in Figure 1-12.

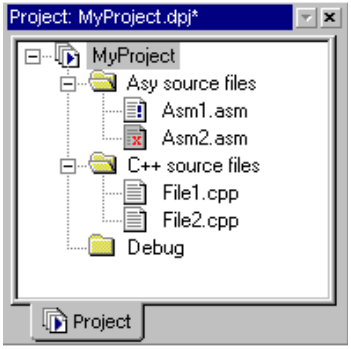

Figure 1-12. Example of Project Files

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.

For details about using the Help system, refer to "Online Help Features and Operations" on page A-48.

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

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). You can open project files by dragging and dropping them into the main window.

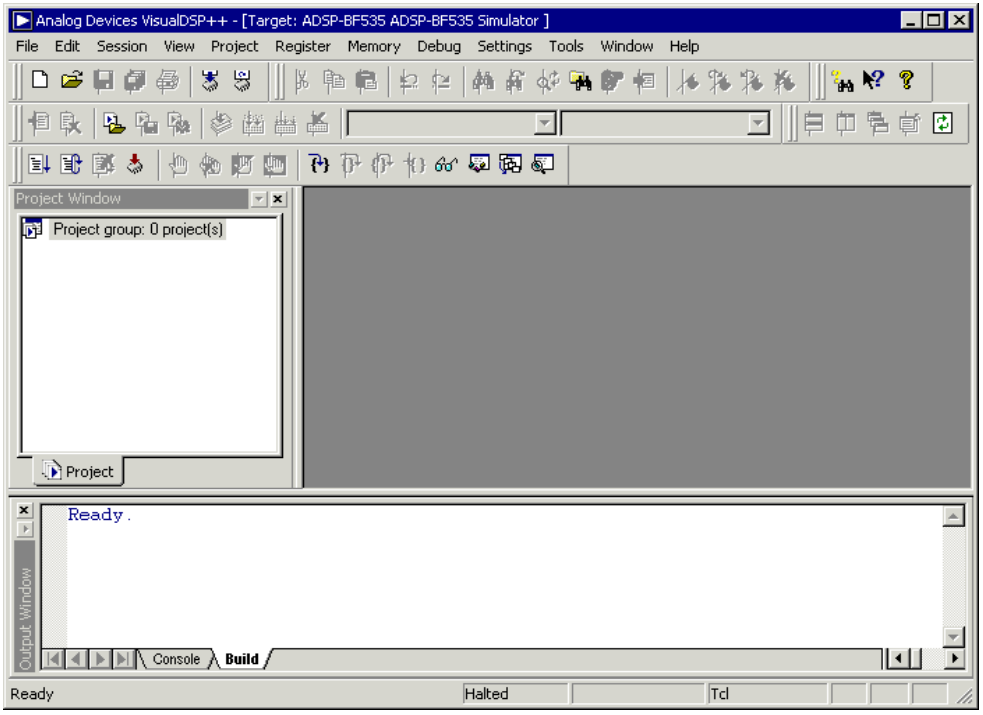

Figure 2-1. Example of VisualDSP++ Main Window

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 facilitate project development. 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.

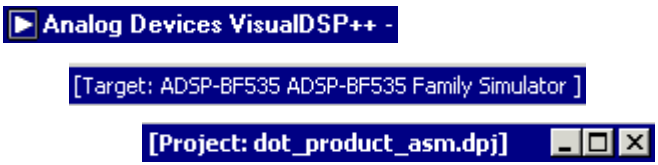

Figure 2-2. Example Title Bar (Split into Three Parts to Fit the Page)

The title bar includes these components:

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

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 2-52.

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 to display Help.

  View the description of the command in the 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.

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	
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

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 on page 2-11 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 script 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.

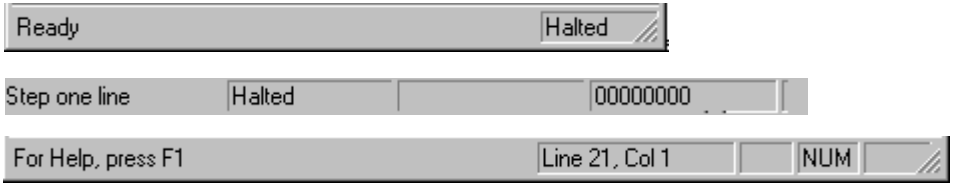

Figure 2-8. The Status Bar's Appearance Depends on Context

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.
- When you halt program operation with a **Halt** command, the address where the program halted appears.
- When you use some script commands, the status bar provides information, such as when the menu item has focus.

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 latched down
NUM	The keyboard's **Num Lock** key is latched down
SCRL	The keyboard's **Scroll Lock** key is latched down

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

To open a **Project** window, choose **View** and **Project Window**. Figure 2-9 shows a **Project** window with VDK enabled.

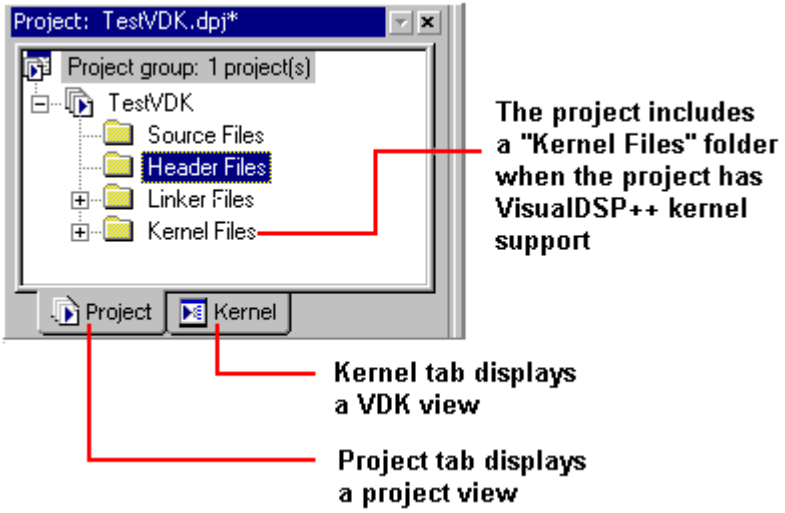

Figure 2-9. Project Window with Kernel Tab

The **Project** window can include two subtabs:

- The **Project** tab 

, which is always available, provides a hierarchal representation of a debug session's projects, folders, files, and dependencies.
- The **Kernel** tab 

appears when you enable VDK for a project.

Project View

The **Project** view displays a project group, which may contain any number of projects. Only one project, however, is active at a time. Nodes are arranged in a hierarchy similar to the file structure in Windows Explorer.

Figure 2-10 shows some of the information displayed in the **Project** view.

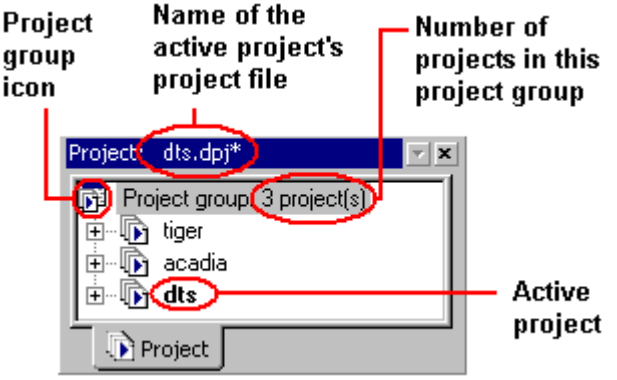

Figure 2-10. Project View

Project Dependencies

A project may depend on other projects. The icon indicates dependency and identifies the dependency. Building a project also builds the projects on which it depends.

Figure 2-11 shows how project dependencies are indicated in the **Project** view.

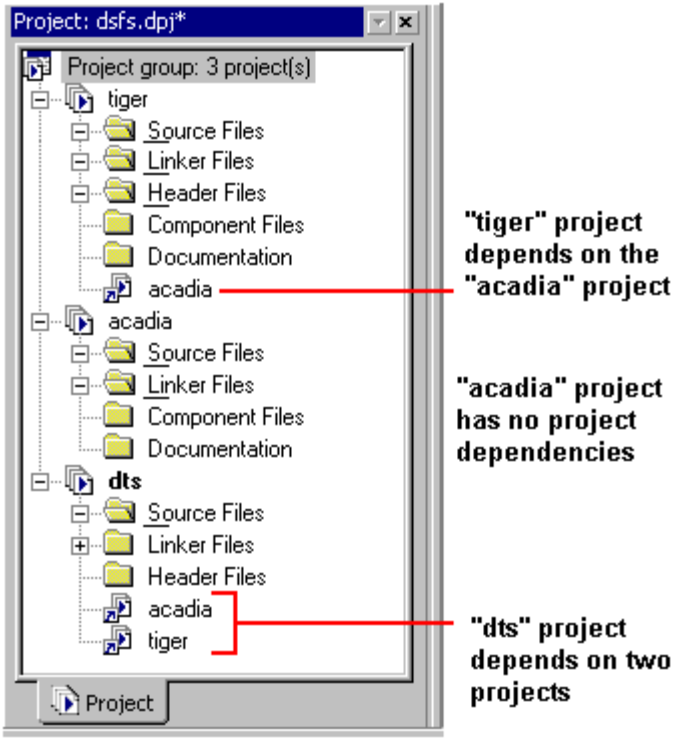

Figure 2-11. Projects Dependencies Indicated in the Project View

Project Nodes

The **Project** window comprises the types of nodes described in Table 2-5.

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

Node	Icon	Description
Project group		Only one project group permitted in a debug session
Project		Multiple projects permitted, but only one is active (indicated with bold typeface)
Folder		Closed folder
		Opened folder revealing its contents
File		File that uses project settings
		File whose options differ from the project options
		File excluded from the current configuration
		Enabled (active) makefile
Project dependency		Project on which this project depends

Project Page Right-Click Menus

Right-click menus (also called context menus or popup menus) operate on Project window objects (the project group, projects, folders, and files). These menus provide fast access to many menu bar and toolbar commands. The commands available from a right-click menu depend on context (the selected object).

Project window right-click menus offer these standard commands:

- **Allow Docking** (dock the Project window to the frame)
- **Hide** (remove the Project window from view)
- **Float in Main Window**

Project Group Icon Right-Click Menu

The project group icon ( ) right-click menu (Figure 2-12) provides a project group context from which you can:

- Create a new project
- Open a project and add it to the project group
- View the project group's properties

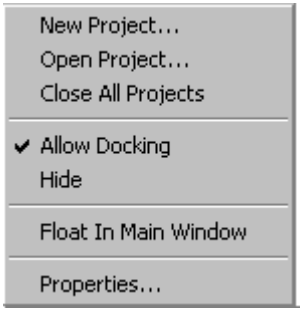

Figure 2-12. Project Group Icon's Right-Click Menu

Project Icon Right-Click Menu

The project icon ( ) right-click menu (Figure 2-13) provides a project context from which you can:

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

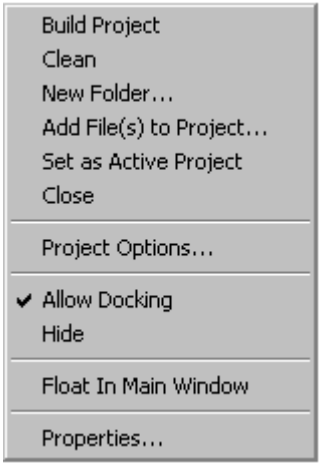

Figure 2-13. Project Icon's Right-Click Menu

Folder Icon Right-Click Menu

The selected folder icon (or) right-click menu (Figure 2-14) provides a "container" context from which you can perform these "local" operations:

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

Figure 2-14. Folder Icon Right-Click Menu

File Icon Right-Click Menu

The selected file icon (or or) right-click menu (Figure 2-15 on page 2-22) provides a file context from which you can:

- 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

Figure 2-15. File Icon Right-Click Menu

File icon commands apply to the selected file in the **Project** window, not to a source file in an editor window.

Project Folders

Project window folders (and) 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 manually move a file anywhere.

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 2-6. 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
	Enabled (active) makefile

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.

Project Window Icons for Source Code Control (SCC)

Icons in the **Project** window indicate source code control (SCC) status. Files with a green check mark (✔) are under SCC and are checked in. Files with a red check mark (✔) are checked out of SCC. When a file is not connected to a controlled copy under SCC, the file icon has no check mark.

The following file icons indicate SCC status.

Table 2-7. SCC Status Icons

Icon	Description
	File is under SCC and is checked in
	File is under SCC and is checked out
	Project file is checked out
	File includes a file-specific build command and is checked out
	Makefile is checked out
	File is excluded from the build and is checked out

File Associations

VisualDSP++ associates these file extensions 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

Automatic file placement enables you to drag and drop files into designated folders on the **Project** page 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 onto the **Project** page, the file is added to the first folder associated with the file's extension. The **Project** page accepts dragged files only when a project is opened.

- Using menu commands to add files

  Files are added to the folders that you select on the **Project** page. 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.

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

Kernel Page

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

From the **Kernel** page, you can 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 from the **Kernel** page.

The example in Figure 2-16 on page 2-28 shows an expanded view of the elements on the **Kernel** page for a VDK-enabled project.

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

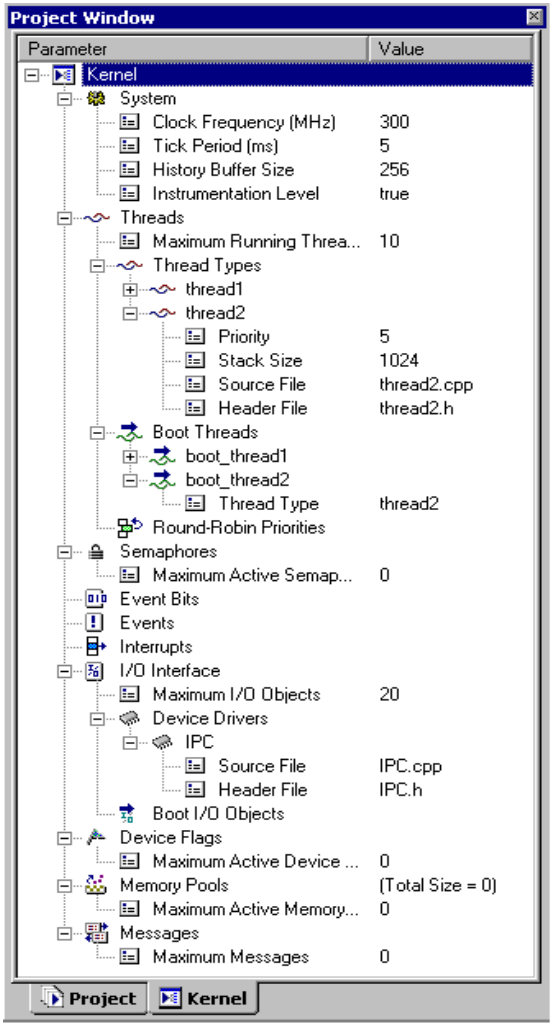

Figure 2-16. Expanded View of Elements on the Kernel Page

Editor Windows

Use editor windows to view and edit project files. You can open an editor window from the **Project** window by double-clicking on a file or by choosing **Open File** from a file's right-click menu. Figure 2-17 shows items that you can customize in editor windows.

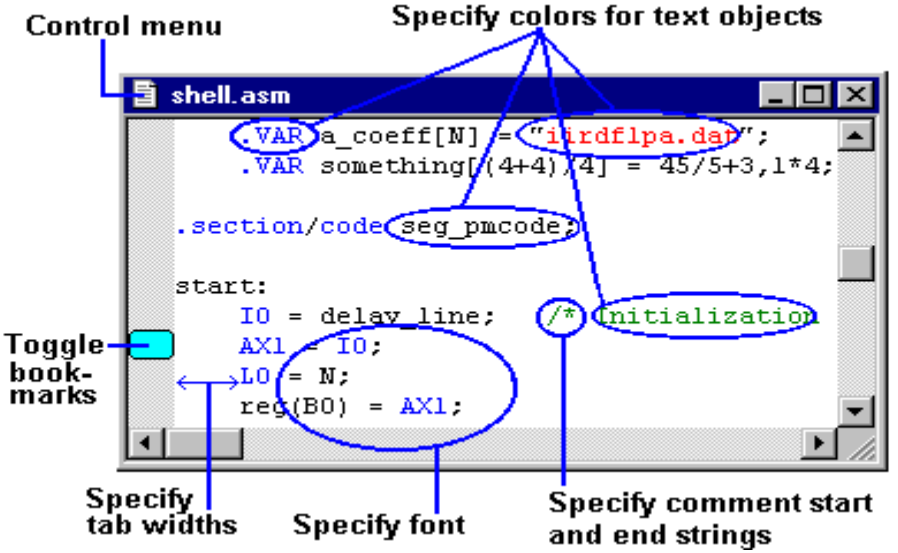

Figure 2-17. Items that can be Customized

You can open as many editor windows as you like and do the following.

- Define color-coded comments, strings, keywords, and tabs
- Preview and print window data
- Load a script
- Define headers and footers
- Set bookmarks

- Find, replace, use wrap-around search and 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 **Editor Tab** mode to switch quickly between source files (see "Editor Tab Mode" on page 2-31).
- 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

The editor window's right-click menu provides these commands:

- **Undo** or **Redo** the last edit
- **Cut**, **Copy**, or **Paste** text
- **Toggle Bookmark** or go to the **Next Bookmark**
- Display **Line Numbers** or **Go To** a specified line number
- **Run to Cursor**
- Locate **Match Brace** characters

- **Find** a specified value by indicating various search parameters
- **Select Format** (Hex, Float, Unsigned Integer, Integer, Octal)

Editor Tab Mode

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

Figure 2-18 shows an editor window with the **Editor Tab** option enabled.

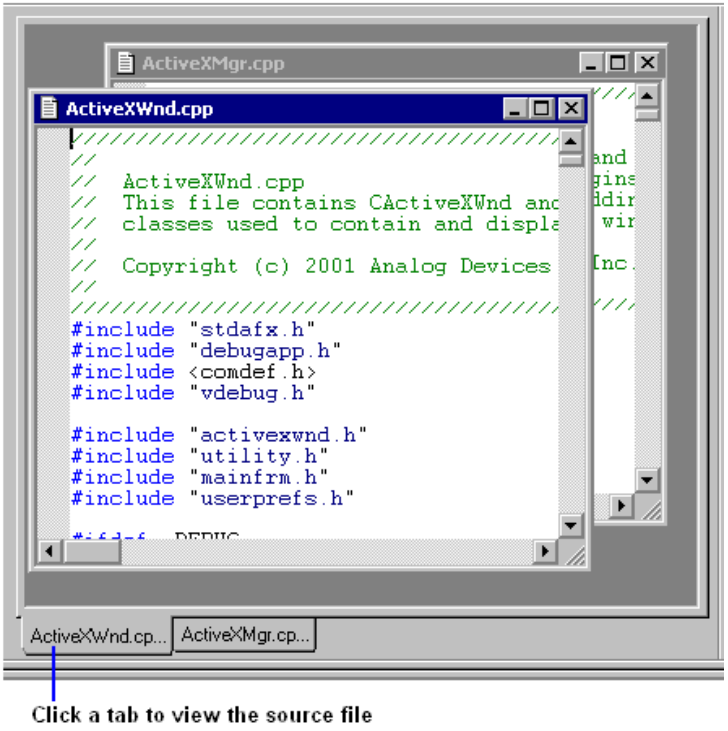

Figure 2-18. Editor Tab Mode Enabled

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 a scripting interface

The **Output** window shown in Figure 2-19 contains build status information.

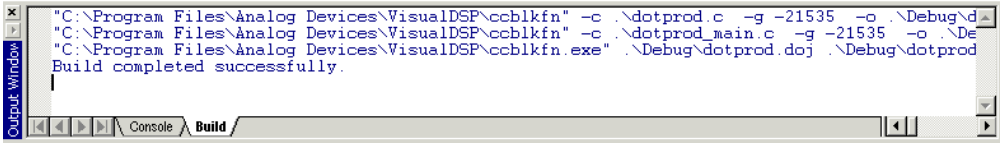

Figure 2-19. Build Status Information in the Output Window

Display the **Output** window by choosing **View** and **Output Window**.

Output Window Tabs

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

Build Page

The **Build** page (Figure 2-20) displays error messages generated during a build. Double-click on an error message to jump to the offending code in an editor window.

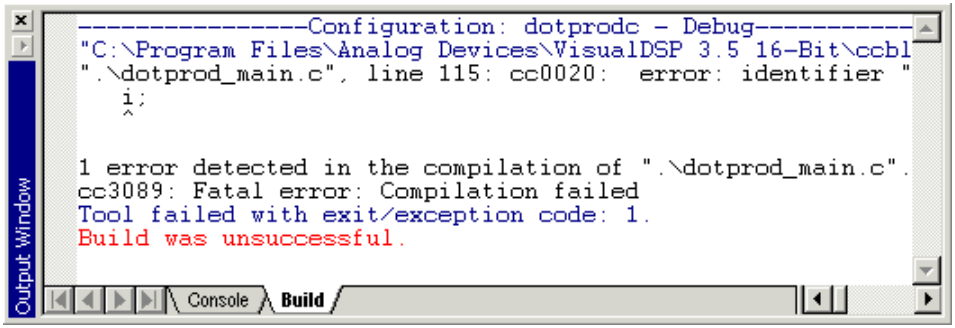

Figure 2-20. Error Messages in the Output Window

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 Page

From the **Output** window's **Console** page (Figure 2-22 on page 2-42), you can:

- View VisualDSP++ or target status error messages
- View STDIO output from C/C++ programs
- View I/O (streams) messages
- Scroll through previous commands by pressing the keyboard's up arrow (↑) and down arrow (↓) keys

- Perform multi-line selection, copy, paste, and clear
- Issue script commands and view script command output
- Auto-complete script commands
- Execute a previously issued script command by double-clicking on the command
- Enter multi-line script 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 on the **Console** page 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 on the **Build** page in the **Output** window.

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	Identifies errors so severe that further processing of the input is suspended. Fatal errors are sometimes called catastrophic errors.
Error	Identifies 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	Identifies situations that do not prevent the tool from processing the input, but may indicate potential problems
Remark	Provides information of possible interest

You can change the severity level of an error marked “discretionary.” You cannot change the severity level of an error marked “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.

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: • ar (archiver) • cc (compiler) • ea (assembler) • el (expert linker) • li (linker) • pp (preprocessor) • vc (VIDL compiler) • vu (VCSE)
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 the severity level of 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 Error Messages

You can change the severity level of an error marked "discretionary." Refer to the tools documentation for command-line switches that override error message severity. The VisualDSP++ environment' **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.

For example, if a condition in the input crashes the tool, you can restrict the severity level of the problem to ensure that an error (instead of a fatal error) is reported.

Another way to suppress the reporting of an individual error message is to use pragmas in the input source via the tool's command line. For more information about pragmas, refer to your processor's *VisualDSP++ 3.5 C/C++ Compiler and Library Manual*.

The following examples demonstrate how you can promote, demote, and suppress messages. The 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:

```
"test.c", line 5: cc0117: {D} warning: non-void function
"foo" should return a value
}
^
"test.c", line 10: cc0549: {D} warning: variable "x" is used
before its value is set
printf("x = %d\n", x);
^
build completed successfully
```

  Note that the compiler appended `D` to each of the warning messages (cc0117 and cc0549) to indicate that the message is discretionary.

- Example 2: Promoting Warnings to Errors

  Typing `$ ccblkfn -c test.c -Werror 549` in a command window promotes one of the two warnings to an error.

```
“test.c”, line 5: error cc0117: {D} warning:  non-void function
"foo" should return a value
}
^
“test.c”, line 10: error cc0549: {D} error: variable “x” is used
before its value is set
printf(“x = %d\n”, x);
^

1 error detected in the compilation of “test.c”.
cc219x: 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 additional remarks.

  ```
  $ cc219x -c test.c -Wremarks
  ```

  The `-Wremark 549,117` 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.

  ```
  $ cc219x -c test.c -Wremark 549,117
  ```

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

  ```
  $ cc219x -c test.c -Wremark 549,117 -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.

  ```
  $ cc219x -c test.c -Wsuppress 549,117 -Wremarks
  ```

How to Suppress the Reporting of Compiler Warnings and Remarks

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

You cannot suppress compiler warnings without also suppressing remarks.

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

From the **Compile** page (**Warning** 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	Issues 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`) on the **Build** page 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** page.

Figure 2-21 shows a sample log file.

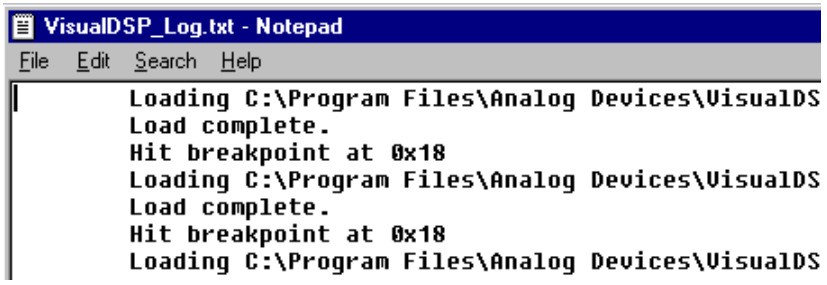

Figure 2-21. Example – Portion of a Log File

The file path specified in the log file assumes that you installed VisualDSP++ by accepting the 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 to a different portion of the screen by dragging it to the selected location. You can dock, hide, or float the window.

The **Output** window's **Console** page can interact with script engines. All script input and output is sent to the **Console** page, shown in Figure 2-22.

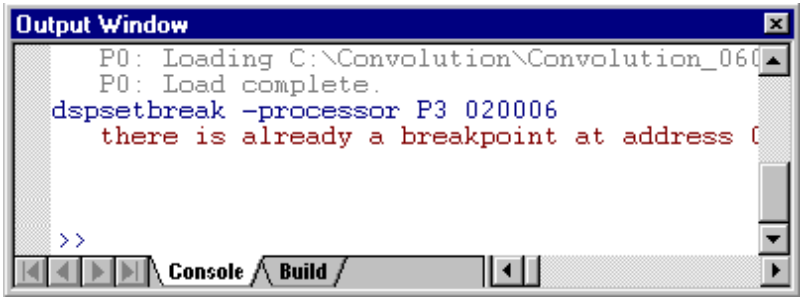

Figure 2-22. Messages in the Project Window's Console Page

These messages are saved to the log file `VisualDSP_Log.txt`, which is located in your installation's `Data` directory.

Right-Click Menu

The **Output** window's right-click menu is shown in Figure 2-23.

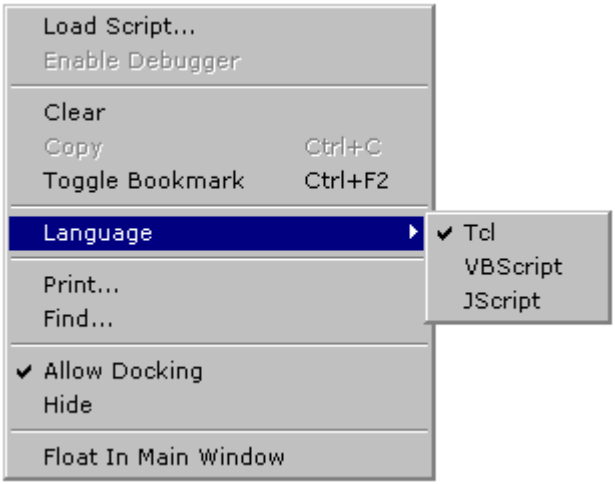

Figure 2-23. Output Window's Right-Click Menu

This menu enables you to:

- Load a script or enable the debugger
- Clear the text in the window or copy selected text
- Toggle bookmarks
- Choose a scripting language
- Print or find text in the window
- Dock, hide, or float the window. (To display the hidden window choose **Output Window** from the **View** menu.)

Script Command Output

Scripts provide a powerful means of developing full-blown test applications of DSP systems. VisualDSP++ 3.5 (and higher) includes a language-independent scripting host that uses the Microsoft ActiveX® script host framework. This scripting host lets you use multiple scripting languages that conform to the Microsoft ActiveX script engine.

The main benefit of calling scripts in these languages is that they have support for COM scripting, which allows access to the VisualDSP++ Automation API. VisualDSP++ supports the following Microsoft ActiveX script engines (languages):

- Visual Basic® (Scripting Edition)
- JScript®

The Tool Command Language (Tcl) interpreter included with VisualDSP++ is not a Microsoft ActiveX script engine. VisualDSP++ permits the use of other script engines (languages) that are not supported by Analog Devices technical support.

Script output is logged to `VisualDSP_log.txt` for viewing and analysis. By default, this file is located in the installation's `Data` directory.

In the **Output** window's **Console** view, you can:

- Issue script commands and view script command output

  For more information about issuing script commands, refer to "Extensive Scripting" on page A-34.

- Enable the Microsoft Script Debugger

  Right-click in the **Output** window and choose **Enable Debugger**. The debugger steps through code, set breakpoints, and so on. Once enabled, the debugger stops on the first error encountered in the script.

Note that although most script engines (languages) support this option, some may not. Consult the script engine's documentation for further details on whether it supports the debugging interfaces within the Microsoft ActiveX script engine framework.

- Specify the scripting language

  Right-click in the **Console** view and select a language from a list of scripting languages installed on your machine.

  The name of the current scripting language appears in the status bar at the bottom of the VisualDSP++ main window, as shown in Figure 2-24.

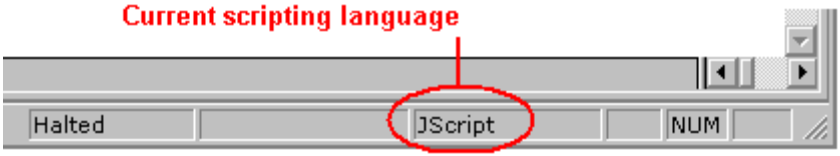

Figure 2-24. Scripting Language Displayed in Status Bar

- Load a script

  You can load a script by selecting **Load Script** from the **File** menu, from the **Console** view's right-click menu, or from the editor window's right-click menu. The script loads and runs until the script finishes running or until you halt the script by choosing **Halt Script** from the **Debug** menu.

The **Console** view supports script command auto-completion if you enable this feature on the **General** page in the **Preferences** dialog box, accessed from the **Settings** menu.

The VisualDSP++ installation directory includes example scripts in the "`Scripting Examples`" folder located under the DSP family name (for example, `Blackfin`) and the `Examples` folder.

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 2-25, enable you to manipulate your windows display and update windows during program execution. Refer to your Windows documentation for more information.

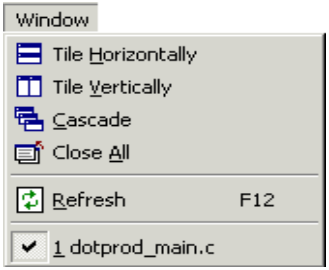

Figure 2-25. Window Menu Commands

Right-Click Menu Options

A menu appears when you right-click in a 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 2-26.

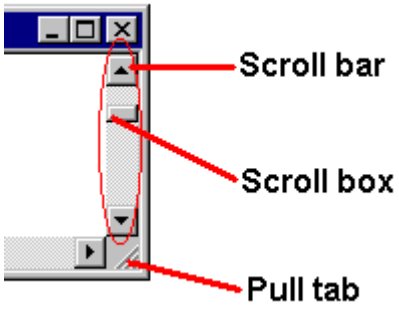

Figure 2-26. Scrolling to Move the Viewing Area

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 2-27 is docked (**Allow Docking** is selected).

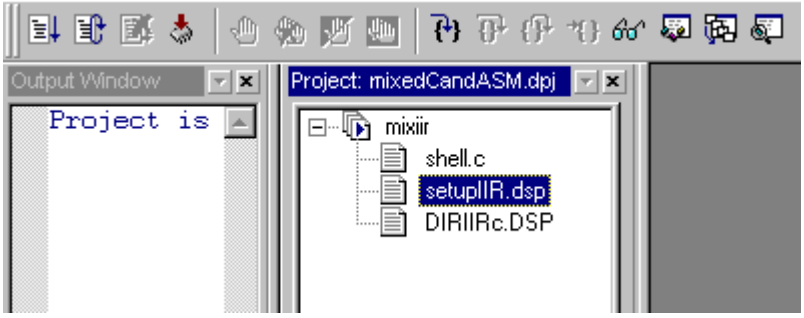

Figure 2-27. Example of a Docked Project Window

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

Examples of Floating Windows

The **Project** window in Figure 2-28 is floating in the main window (**Float In Main Window** is selected).

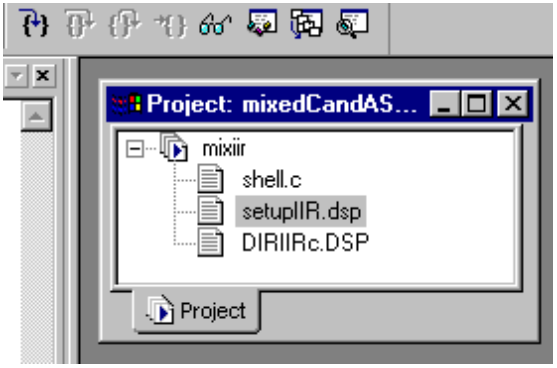

Figure 2-28. Project Window Floating in Main Window (1 of 2)

The **Project** window in Figure 2-29 is also floating in the main window (**Float In Main Window** is selected).

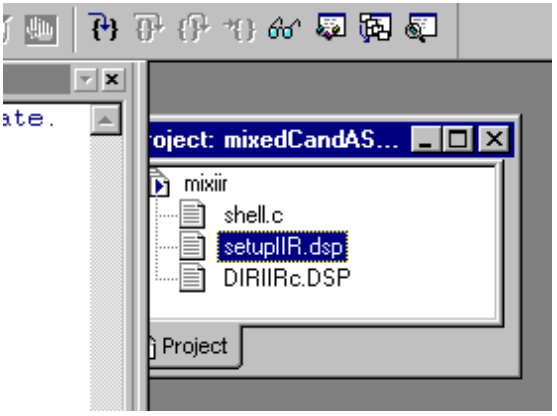

Figure 2-29. Project Window Floating in Main Window (2 of 2)

The Project window in Figure 2-30 is floating, but not in the main window (**Float In Main Window** is not selected).

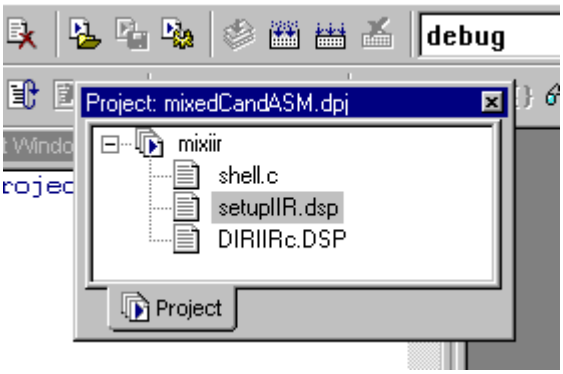

Figure 2-30. Project Window Floating but Not in Main Window

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, a window specified as an MDI child cannot be positioned 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 2-31.

Figure 2-31. 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** page that displays standard I/O text messages such as file load status, and error messages and streams, and a **Build** page that displays build messages. You can interactively enter script commands and view script 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 (ADPSP-21xx processors only)
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

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

Window	Provides
Register	Current values of registers. You can change register contents and change the number format.
Custom Register	Current values of registers. Select the registers that you want to monitor.
Memory	A view of DSP memory. Similar number format and edit features as register windows, plus fill and dump capability.
BTC Memory	A view of background telemetry channel contents in real time. The window displays the contents of the address that you want to see.
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 (ADSP-219x and ADSP-BF561 processors only). This window allows you to define and manage groups of processors for synchronous multiprocessor commands.
Pipeline Viewer	A simulation-only view of instructions in the pipeline and event details (does not apply to ADSP-218x processors).
Cache Viewer	Analysis of a DSP application's use of cache, which is helpful in optimizing DSP application performance
VDK 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
Image Viewer	A view of BMP, JPEG, PPM, or MPEG data from DSP memory or from a file on your PC. You can edit, copy, print, or export image data.

Disassembly Windows

By default, a **Disassembly** window appears when you open a new session. You can also open a **Disassembly** window by choosing **View**, **Debug Windows**, and **Disassembly**.

Figure 2-32 and Figure 2-33 show examples of **Disassembly** windows, one with and one without the address bar enabled.

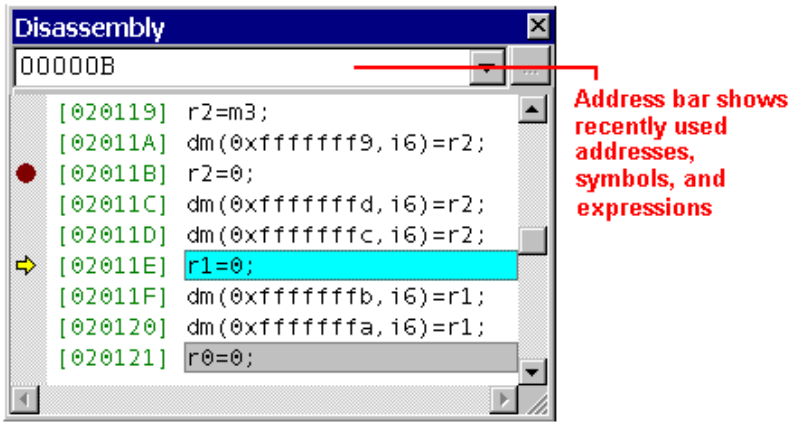

Figure 2-32. Disassembly Window with Address Bar

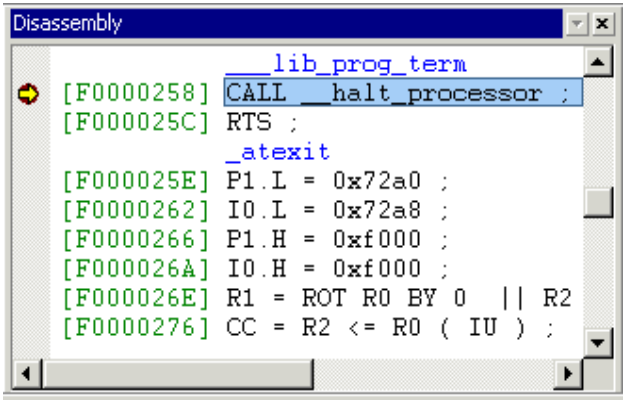

Figure 2-33. Disassembly Window Without Address Bar

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 enables you to examine the assembly code generated by the C/C++ compiler. Choosing **View Source** from the **Disassembly** window's right-click menu enables you to view the C/C++ source code for the loaded file.

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
- Symbols at the far left of the window, denoting program execution stages and pipeline stages

  You can enable and disable the display of pipeline symbols while in mixed mode (C/C++ and assembly).

- An optional address bar that enables you to navigate to an address, symbol, or expression. The address bar maintains a most recently used history of visited locations.

  To display the address bar, right-click in a **Disassembly** window and choose **Address Bar**. A check mark next to this option on the right-click menu indicates that this feature is enabled.

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

Figure 2-34. 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. Note that you can also use the address bar to navigate to an address, symbol, or expression.
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 2-35.

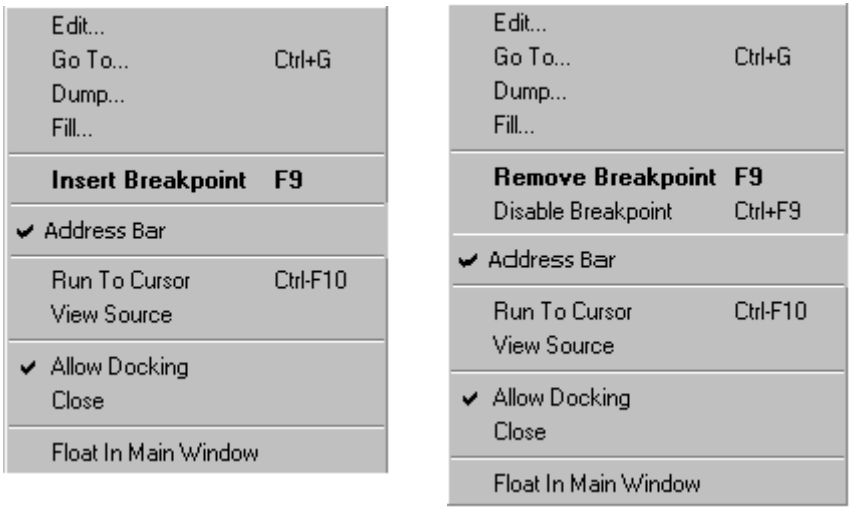

Figure 2-35. Disassembly Window Right-Click Menus

Disassembly Window Symbols

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

The symbols displayed at the left of the **Disassembly** window are shown in Table 2-17.

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

Expressions Window

The **Expressions** window (Figure 2-36) lets you enter an expression to evaluate in your program. Evaluations are based on the current debug context. Open this window by choosing **View**, **Debug Windows**, and **Expressions**.

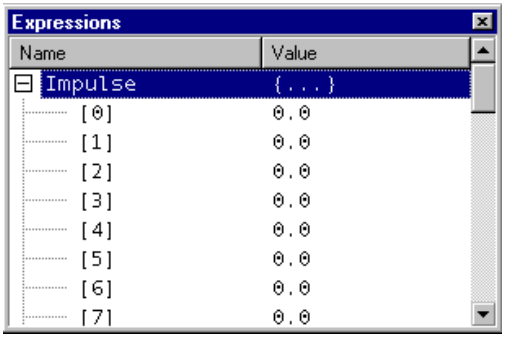

Figure 2-36. Expressions Window

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 from the **Call Stack** window.

The **Expressions** window's right-click menu (Figure 2-37) includes commands that let you change the display's number format.

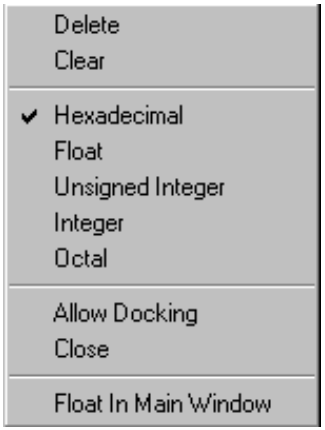

Figure 2-37. Expressions Window Right-Click Menu

Locals Window

The **Locals** window displays the value of local variables within a function, as shown in Figure 2-38. Open this window from the **View** menu by choosing **Debug Windows** and **Locals**.

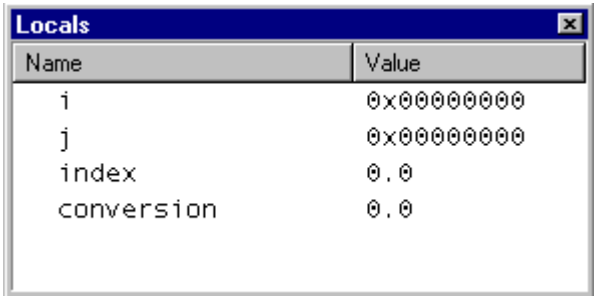

Figure 2-38. Locals Window

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 2-39.

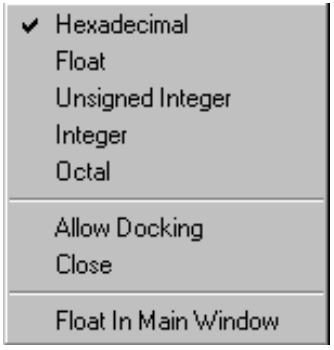

Figure 2-39. Locals Window Right-Click Menu

Trace Window

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. Open a **Trace** window by choosing **View**, **Debug Windows**, and **Trace**. Figure 2-40 shows data displayed in a **Trace** window.

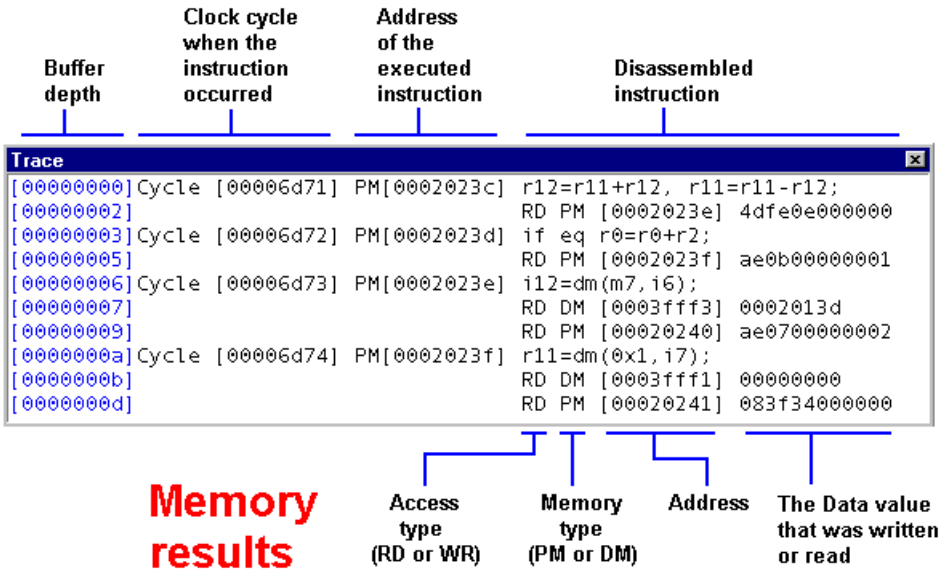

Figure 2-40. Example of Data in a Trace Window

The Trace window displays:

- Buffer depth (**Custom** depth in **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

Statistical/Linear Profiling Results Window

To open a profiling results window, choose **Tools**, **Linear Profiling** or **Statistical Profiling**, and **New**. 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 2-41.

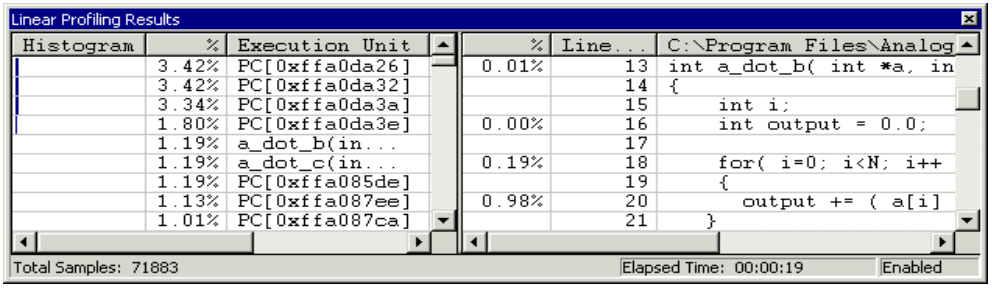

Figure 2-41. Example of a Linear Profiling Results Window

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.

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.

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 2-66 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 with the ADSP-BF532 simulator only, 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(0x2000)`
- Address ranges, for example, `[2000-2050]`

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 2-42 shows an example of code displayed for a function.

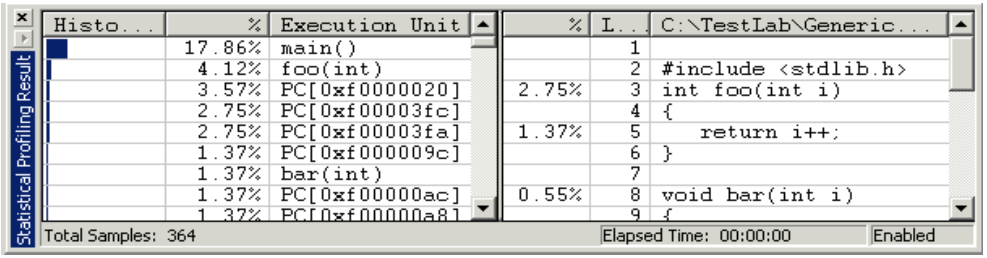

Figure 2-42. Code Displayed for a Function

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.

Switching Display Modes

The right-click menu's **Mixed** and **Source** commands simplify switching between two views. Figure 2-43 shows the source mode view and Figure 2-44 on page 2-69 shows the mixed mode view.

You can display the right pane in source mode

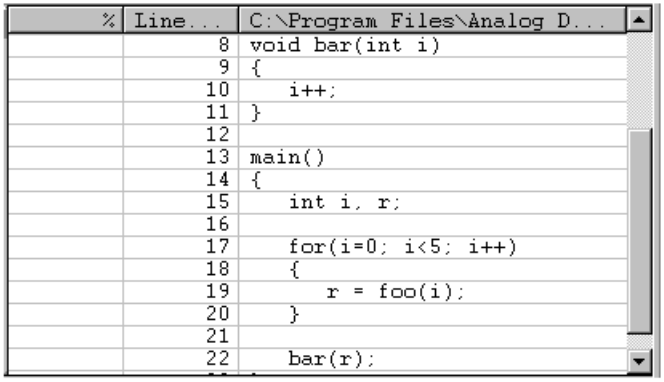

Figure 2-43. Source Mode View

Mixed mode dispalys C/C++ source lines and assembly code.

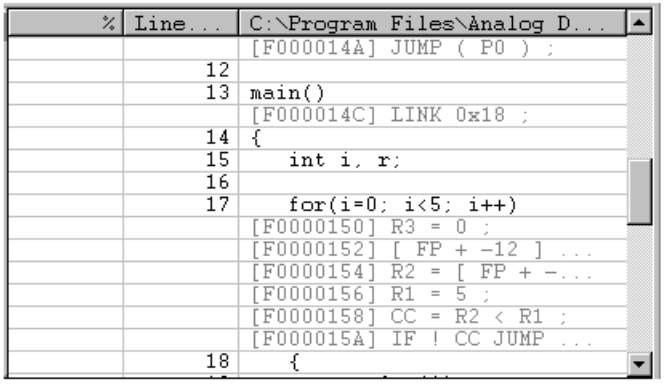

Figure 2-44. Mixed Mode View

When you view the window in mixed mode, profiling data for each assembly line is displayed, as shown in Figure 2-45. Mixed mode displays profiling statistics for individual assembly instructions.

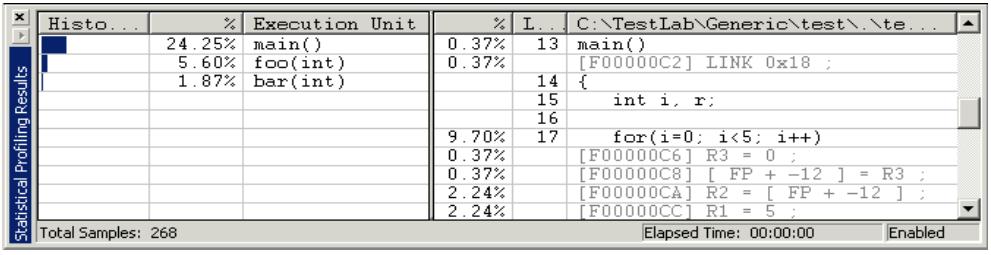

Figure 2-45. Profiling Data for Each Assembly Line (Mixed Mode)

Filtering PC Samples with No Debug Information

Since you spend most of you time building a “debug version” of your code, eliminate non-debug code, such as C run-time library initialization code.

Figure 2-46 shows where a lot of time is spent before filtering.

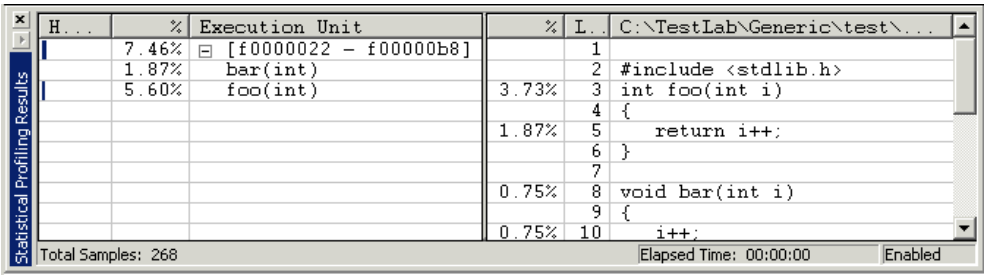

Figure 2-46. Profiling Results Before Filtering

The profiling results after filtering (Figure 2-47) reflect the difference.

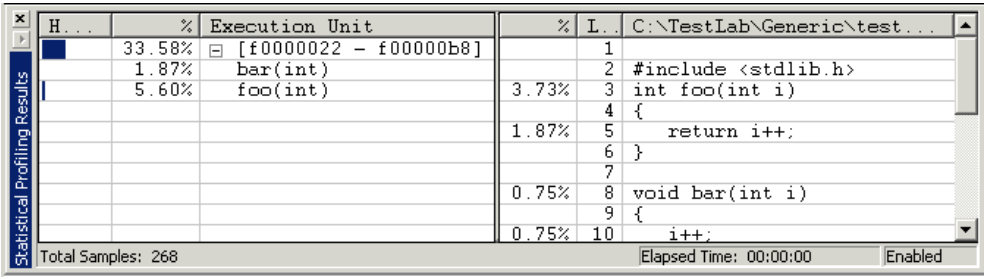

Figure 2-47. Profiling Results After Filtering

Call Stack Window

The **Call Stack** window (Figure 2-48) enables you to double-click on a stack location to move the call stack back to a previous debug context. Open this window by choosing **View**, **Debug Windows**, and **Call Stack**.

Figure 2-48. 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

Memory windows let you:

- 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

You open memory windows from the **Memory** menu. For Blackfin processors, choose **BLACKFIN Memory**. For ADSP-21xx processors, choose the type of memory that you want to display: **Program**, **Data**, **Byte** (ADSP-218x only) or **I/O**.

Memory windows provide:

- Number format and edit features
- Fill and dump capability
- An optional address bar for fast navigation to recently used addresses, symbols, or expressions

  To display the address bar, right-click in a **Memory** window and choose **Address Bar**. A check mark next to this option on the right-click menu indicates that this feature is enabled.

Memory Number Formats

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

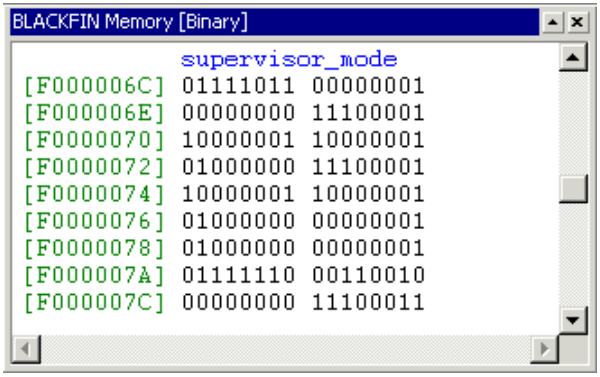

Figure 2-49. Example of Blackfin Memory in Binary Format

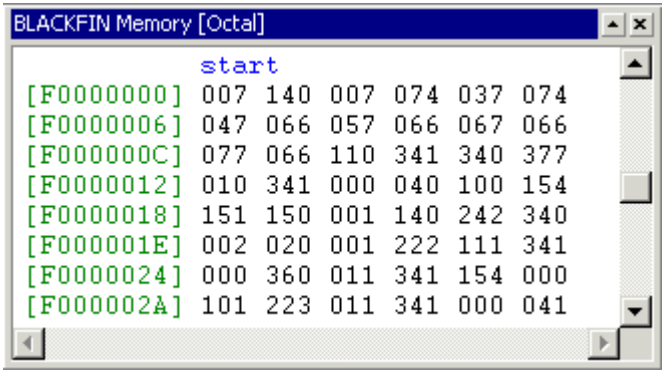

Figure 2-50. Example of Blackfin Memory in Octal Format

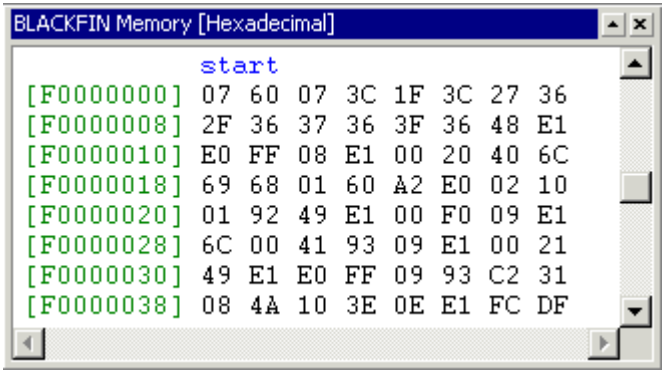

Figure 2-51. Example of Blackfin Memory in Hexadecimal Format

Right-Click Menu

Memory windows provide a right-click menu, shown in Figure 2-52.

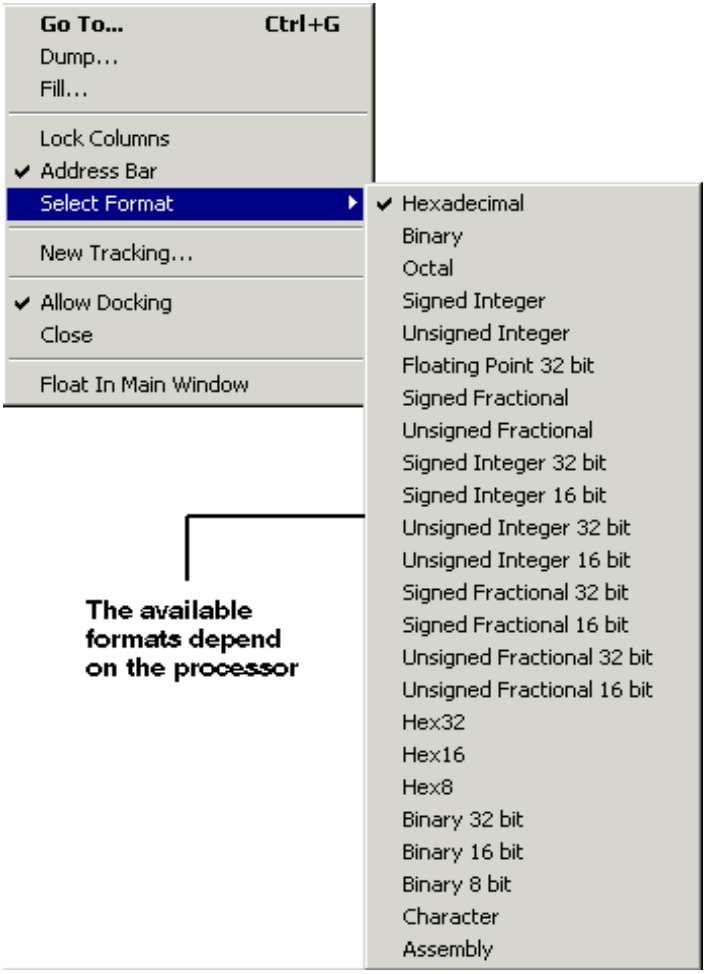

Figure 2-52. Memory Window Right-Click Menu

These commands enable 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.

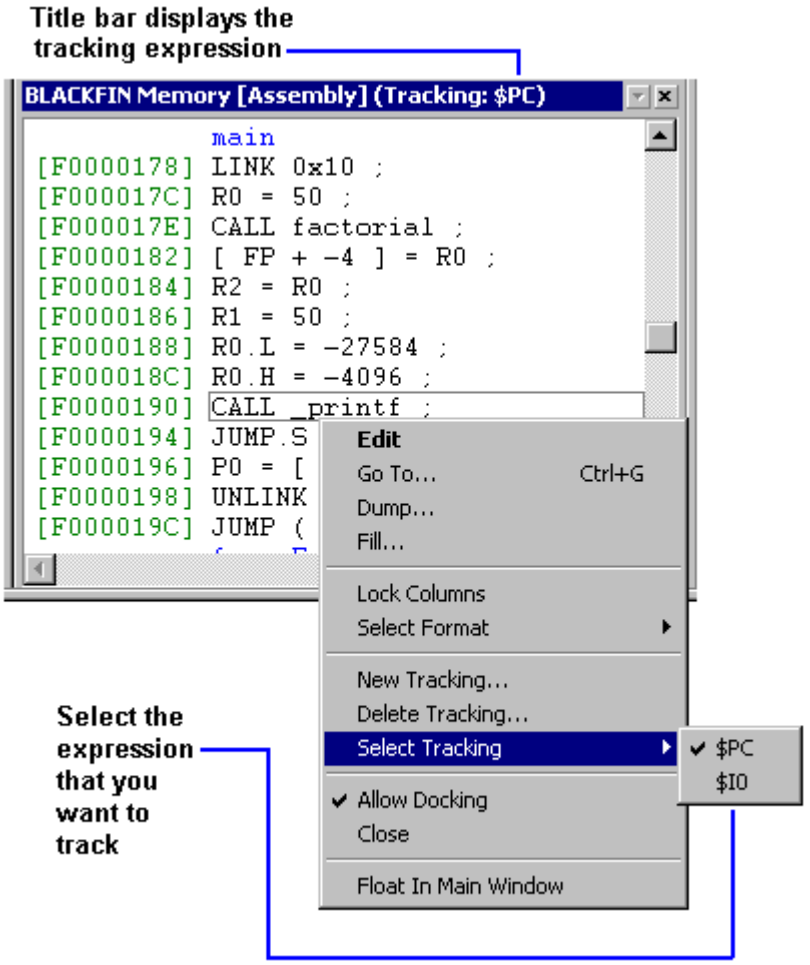

Figure 2-53. Expression Tracking in a Memory Window

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.

Background Telemetry Channel (BTC) Window

Background telemetry channel (BTC) enables VisualDSP++ and a DSP to exchange data via the JTAG interface while the DSP is executing. Before BTC, all communication between VisualDSP++ and a DSP took place when the DSP was in a halted state.

BTC Definitions in Your Program

Background telemetry channels (BTCs) are defined on a per program (`.DXE`) basis. The channels are defined when you load a specific program onto a DSP. You define channels in your program by using simple macros.

The following example code shows channel definitions.

```
#include "btc.h"

.section data_a;

BTC_MAP_BEGIN
   BTC_MAP_ENTRY ('Channel0', 0xf0001000, 0x00100)
   BTC_MAP_ENTRY ('Channel1', 0xf0002000, 0x01000)
   BTC_MAP_ENTRY ('Channel2', 0xf0003000, 0x10000)
BTC_MAP_END
```

The first step in defining channels in a program is to include the BTC macros by using the `#include btc.h` statement. Then each channel is defined with the macros. The definitions begin with `BTC_MAP_BEGIN`, which marks the beginning of the BTC map. Next, each individual channel is defined with the `BTC_MAP_ENTRY_ASM` macro, which takes the parameters described in Table 2-21 on page 2-78.

Table 2-21. Parameters for the BTC_MAP_ENTRY_ASM Macro

Parameter	Description
Name	Name of the channel (32 characters max)
Starting address	Starting address of the channel in memory
Length	Length of the channel in bytes

Once the channels are defined, end the BTC map with the `BTC_MAP_END` macro, which takes a single parameter. This macro indicates the total number of channels being defined. After you add the channel definitions, you must initialize the BTC during the applications startup code by calling the `_btc_init` function. After initialization, BTC commands from the host are processed via the `_btc_poll` function.

BTC Priority

You can call the `_btc_poll` function from a polling loop, the handler of an interrupt, a thread, and so on. Because you decide when to call the `_btc_poll` function, you can effectively change the priority of the BTC, as described in Table 2-22.

Table 2-22. Changing BTC Priority

Placing the BTC Call	Description
In the handler of a high-priority interrupt	The BTC effectively becomes high priority.
In a low-priority interrupt handler	The BTC effectively makes the BTC low priority.
In a polling loop	It is difficult to predict the priority without knowing the impact that interrupts have on the overall system.

The priority of the BTC can impact the response time from when the host requests data and the DSP responds. Once the DSP begins to service the request, interrupts can still be serviced by the DSP. BTC performance is affected by the frequency of system interrupts.

The following example shows a simple polling loop written in assembly language.

```
[--sp] = rets;
call _btc_init;
rets = [sp++];

pollingLoop:

nop;
[--sp] = rets;
call _btc_poll;
rets = [sp++];
nop;
jump pollingLoop;
```

The following example is the same polling loop written in C.

```
btc_init();
while(1)
    btc_poll();
```

After adding the calls to the initialization and polling functions, you must link with the BTC library (`libbtcxxx.dlb`), which contains the initialization and polling functions and other functions that permit data transfer over the BTC.

Examples

The **BTC Memory** window lets you view background telemetry channel contents in real time. The window displays the contents of the address that you want to see. You can change the window's view to meet your needs.

Open this window by choosing **View**, **Debug Windows**, and **BTC Memory**.

The view in Figure 2-54 shows the contents of a specified channel only (for example, Channel1).

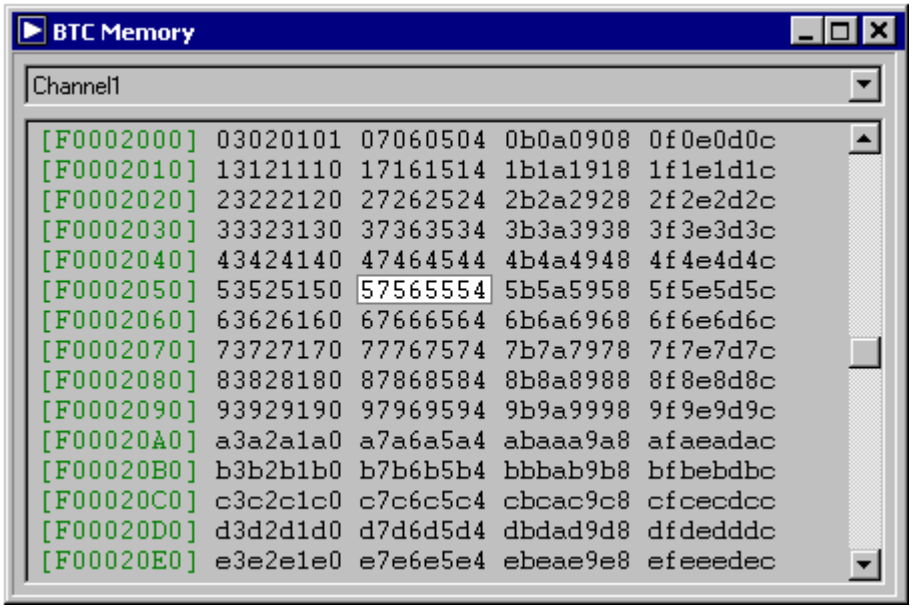

Figure 2-54. Viewing Contents of a Specified Channel Only

The view in Figure 2-55 shows the list of currently defined channels and the contents of the selected channel.

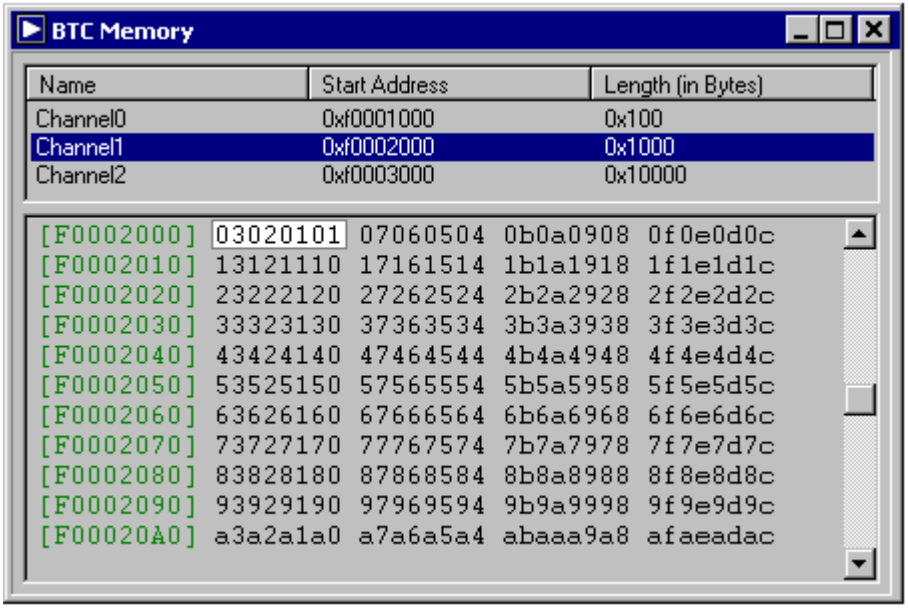

Figure 2-55. Viewing Defined Channels and Contents of a Selected Channel

Right-Click Menu

Table 2-23 describes the **BTC Memory** window's right-click menu.

Table 2-23. BTC Memory Window's Right-Click Menu

Command	Purpose
Go To	Opens the **Go To** dialog box, in which you specify an address. The specified address appears in the top-left corner of the display. The address must be within the range defined for the channel currently being displayed. **Tip:** Double-clicking in the address column also opens the **Go To** dialog box.
Show Map or Hide Map	Shows or hides a more informative map display of all the current channel definitions **Show Map** displays a channel list. Double-click a channel to display its contents in the lower portion of the window. **Hide Map** removes the list of channels. The selected channel remains in the display.
Lock Columns	Locks or unlocks the number of columns currently displayed in the window
Select Format	Specifies how to display data in the window. Choices include double words (32 bits), words (16 bits), and bytes (8 bits).
Refresh Rate	Specifies the refresh rate, which is used when **Auto Refresh** is chosen. The display is updated at the selected interval.
Auto Refresh	Enables the window to refresh itself at given intervals. The rate is specified by **Refresh Rate**. **Auto Refresh** mode is valid only while the processor is running.
Channel Timeout	Specifies the length of time to wait for any single response from the BTC. If the timeout value is exceeded, the current transaction ends.

Memory Map Windows

The **Memory Map** window (Figure 2-56) displays the memory map for the selected processor. Open this window by choosing **Memory** and **Memory Map**.

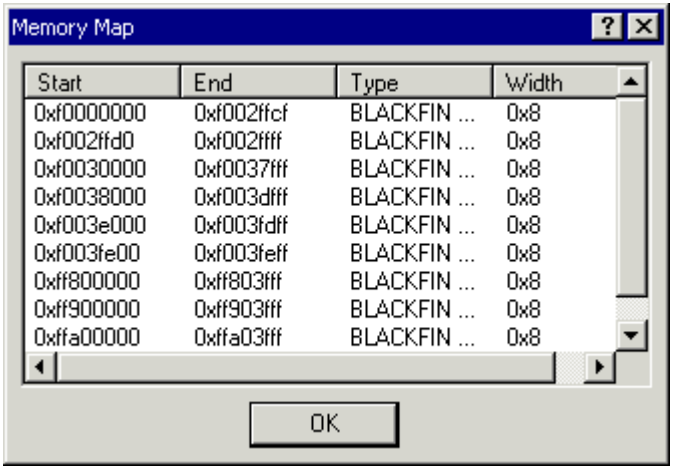

Figure 2-56. Memory Map Window

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

Depending on your processor, you have access to various register windows from the **Register** menu.

The **Core** submenu shown in Figure 2-57 is available for ADSP-BF535 processors.

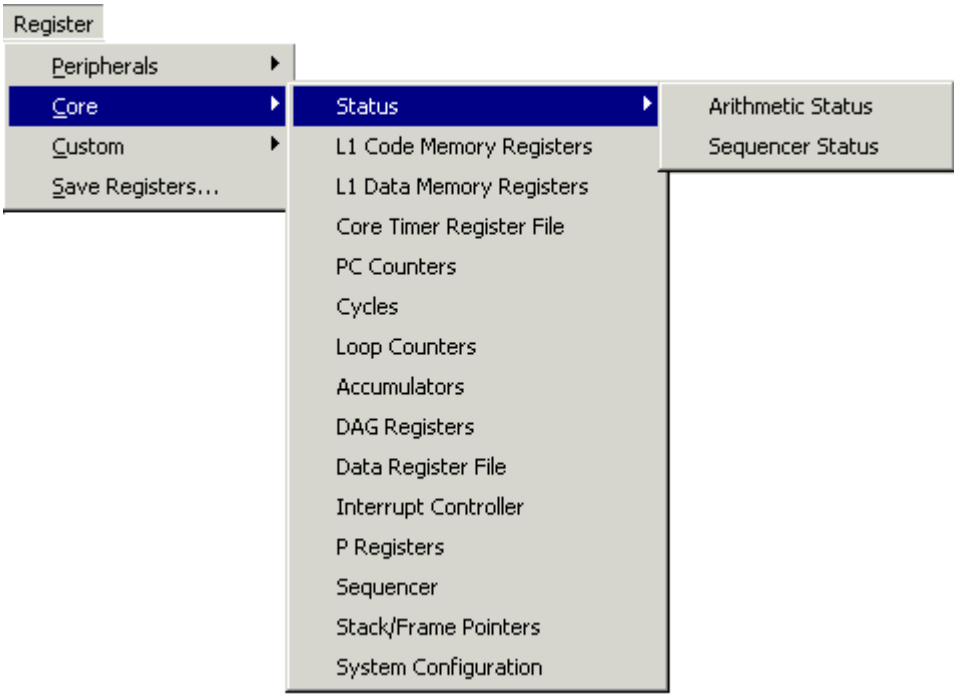

Figure 2-57. Register Windows Available from the Core Submenu

The **Peripherals** submenu shown in Figure 2-58 is available for ADSP-BF535 processors only.

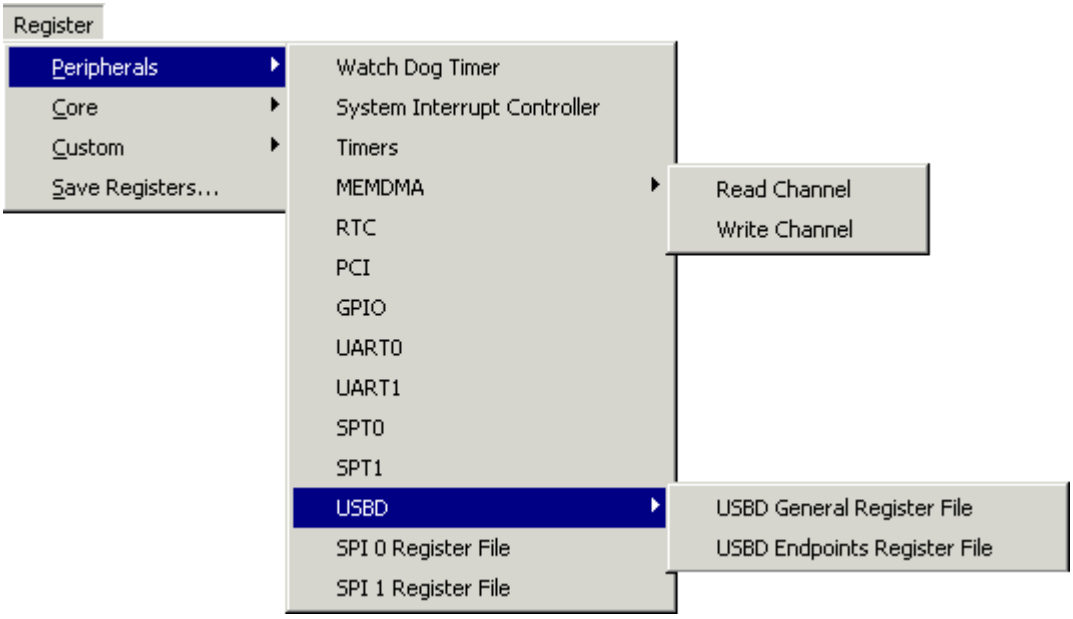

Figure 2-58. Register Windows Available from the Peripherals Submenu

The Register menu shown in Figure 2-59 on page 2-86 is available for ADSP-2191 processors.

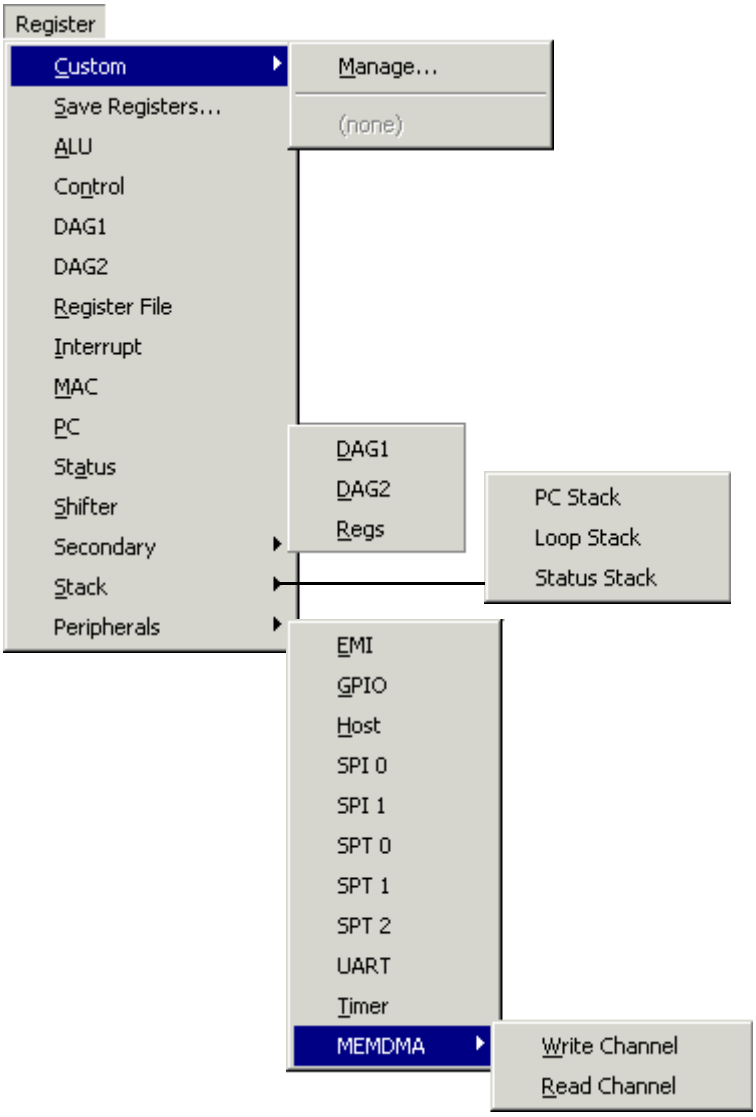

Figure 2-59. Register Windows Available for ADSP-2191 Processors

Figure 2-60 shows an example of a data register file in a register window.

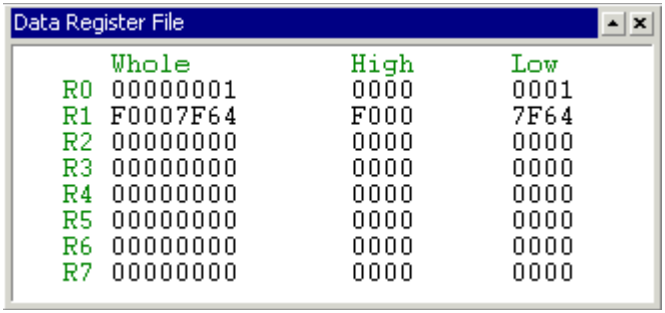

Figure 2-60. Example of a Register Window

A register window enables you to:

- View and change register contents
- Change the presentation (number format)

Register window number formats include standard formats, such as hexadecimal, octal, and binary. Depending on the DSP, other formats are available.

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, including:

- **PC Stack**
- **Counter Stack**
- **Loop Stack**
- **Status Stack**

For more information about your processor's stack windows, consult the online Help.

Custom Register Windows

While debugging, you can configure and display custom register windows. To create a custom register window, choose **Register**, **Custom**, and **Manage**. Then add the registers that you want to display. The custom register window appears immediately after you create it.

Each custom register window displays a title that you specify and only the registers that you choose to monitor. The custom register window shown in Figure 2-61 displays the contents of five registers.

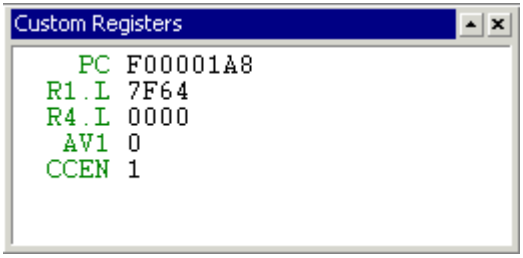

Figure 2-61. Example of a Custom Register Window

Multiprocessor Window

Multiprocessor capability applies only to processors such as the ADSP-2192-12 and ADSP-BF561, which support multiprocessing.

Use the **Multiprocessor** window (Figure 2-62) to select and control the different processors in a multiprocessor debug session.

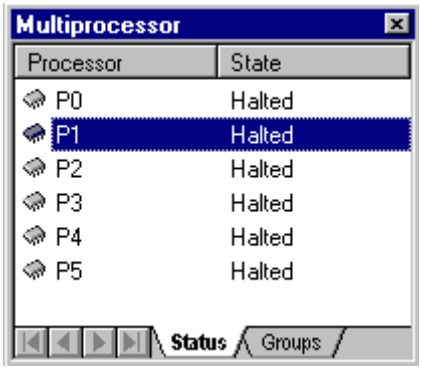

Figure 2-62. Multiprocessor Window

The window consists of two pages, **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 2-62 on page 2-89).

You can pin a register window, a memory window, or **Disassembly** 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** page reflects the change in focus.

Right-Click Menu

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

Figure 2-63. Multiprocessor Window's Right-click Menu

Multiprocessor Window Pages

The **Multiprocessor** window has two tabbed pages, **Status** and **Groups**.

Status Page

The **Status** page (Figure 2-64) shows the status of each processor in a multiprocessor system. The processor with focus is highlighted with a horizontal bar.

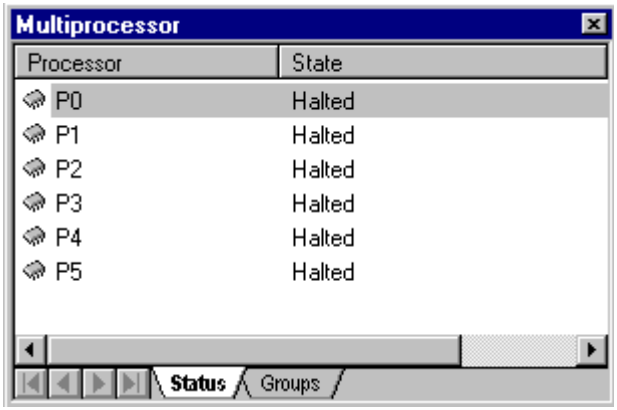

Figure 2-64. Multiprocessor Window – Status Page

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

Groups Page

The **Groups** page (Figure 2-65) shows 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.

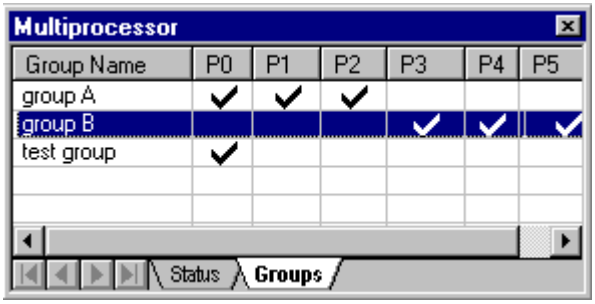

Figure 2-65. Multiprocessor Window – Groups Page

From the **Groups** page, 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**) to only the processors in the currently selected group.

Right-clicking on the **Group** page displays a context menu that lets you add or remove a group.

Pipeline Viewer Window

The **Pipeline Viewer** window (Figure 2-66) enables you to view instructions in the pipeline and event details. Open this window by choosing **View**, **Debug Windows**, and **Pipeline Viewer**.

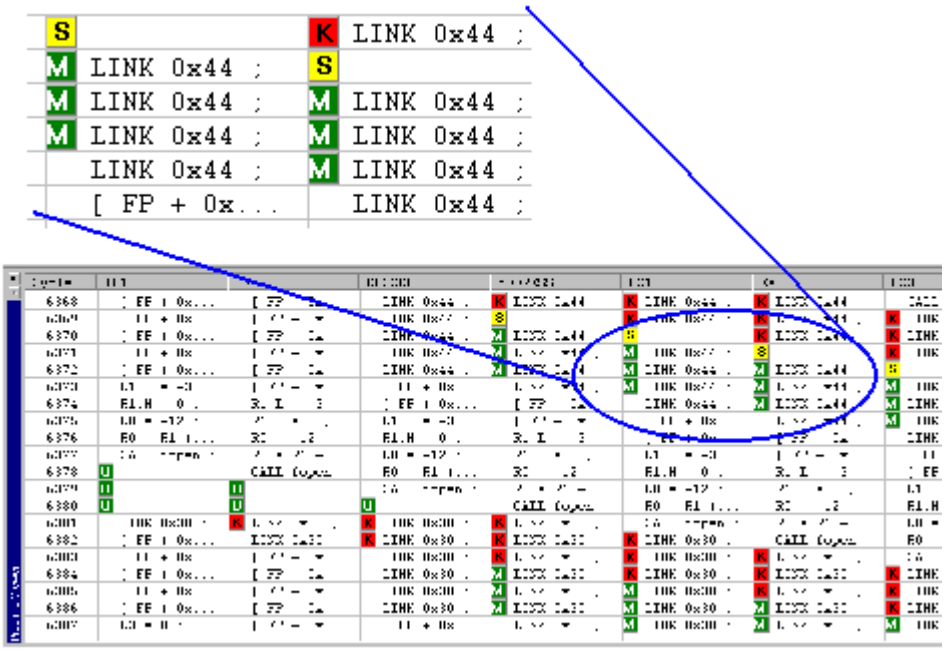

Figure 2-66. Example of a Pipeline Viewer Window

Pipelining is available only for simulation targets.

Column headings refer to pipeline stages for the processor's core registers. Refer to your processor's Hardware Reference for details.

Right-Click Menu

The **Pipeline Viewer** window's right-click menu provides the commands described in Table 2-24.

Table 2-24. Pipeline Viewer Right-Click Menu

Item	Purpose
Enabled	Enables and disables collection of pipeline data while running or stepping
Clear	Clears the current sample buffer
Display Format	Controls the display format of data **Address** shows the hexadecimal-formatted address of the pipeline stage (for example, `0x1234`). Use this format to follow a particular address's route through the pipeline. **Disassembly** disassembles the instruction at that address and shows the opcode mnemonic, similar to a **Disassembly** window. Use this format to determine why a particular event is occurring. **Opcode** format is the hexadecimal representation of the disassembly mnemonic.
Save	Opens the **Save As** dialog box, from which you export the collected data to a text file
Properties	Opens the **Pipeline Viewer Properties** dialog box, from which you view and specify properties (buffer and display depth, display format, column widths, grid lines, and the appearance of stages) for the **Pipeline Viewer window**. You can also modify window colors.

Pipeline Viewer Properties Dialog Box

The **Pipeline Viewer Properties** dialog box enables you to specify how (amount and format) the **Pipeline Viewer** window displays pipeline events. Table 2-25 describes the Pipeline Viewer properties.

Table 2-25. Pipeline Viewer Properties

Property Item	Purpose
Buffer depth	Specifies the total number of pipeline samples to retain at any time. When this buffer overflows, the oldest data shifts out to make room for new samples. The default is `100`.
Display depth	Specifies the number of samples to display. Adjust this number to meet your performance needs. The lower the depth, the faster the target can run. This option cannot be set greater than the **Buffer depth**. The default is `20`.
Display format	Specifies the data's format **Address** includes the hexadecimal-formatted address of the pipeline stage (for example, `0x1234`). **Disassembly** includes the opcode mnemonic, similar to the format displayed in a **Disassembly** window. **Opcode** format is the hexadecimal representation of the disassembly mnemonic.
Show gridlines	Toggles the display of gridlines in the window. The default is **On**.
Auto-size columns	Automatically sizes all columns to have the same width as samples are collected. The default is **On**.
Stages to view	Specifies the stages to appear in the window. Note that all stages are collected, but you view only the stages that you select to appear.

The **Colors** tab lets you specify the colors displayed in the **Pipeline Viewer** window. The current color appears under **Current Color**. You can click a color in the color palette or click **Other** to specify a custom color. Use the **Reset** button to restore the default colors.

Pipeline Viewer Window Event Icons

Table 2-26 shows the **Pipeline Viewer** window icons that indicate Blackfin pipe stage events.

Table 2-26. Icons for Blackfin Pipe Stage Events

Icon	Event	Description
	Abort	The instruction in the pipeline stage has been aborted.
	Bubble	A stall in the previous pipeline stage created an empty slot in this pipeline stage.
	Stall	A stage has stalled. The stall reasons for Blackfin processors are listed in Appendix B.
	Fetch	The pipeline stage was ready to decode but latencies in the memory prevented the fetch pipe from providing an input in time.
	Kill	This status is similar to an abort but is caused by an interrupt, refetch, branch, or mispredicted conditional branch.
	Multi	A stage contains a multicycle opcode such as `pushpopreg`, LINK, or UNLINK.

Table 2-27 shows the **Pipeline Viewer** window icons that indicate ADSP-219x pipe stage events.

Table 2-27. Icons for ADSP-219x Pipe Stage Events

Icon	Event	Description
	Abort	A stage in the pipeline contains an aborted instruction.
	Bubble	The stage following a stall in the pipeline contains an empty slot.
	Stall	A stall has occurred at a stage in the pipeline. The stall reasons for Blackfin processors are listed in Appendix C.

Pipeline Instruction Event Details

In the **Pipeline Viewer** window, moving the cursor over a Pipeline Viewer event icon displays pipeline event details, which appear in a tool tip (message) box, shown in Figure 2-67.

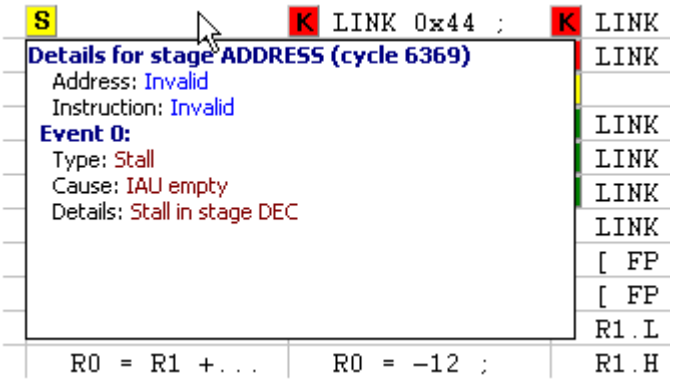

Figure 2-67. Tool Tip Box Showing Pipeline Event Details

A pipeline event can include the details described in Table 2-28.

Table 2-28. Pipeline Event Details

Item	Displays
Address	Address of the pipeline stage at that cycle (if valid)
Instruction	Assembly instruction of that address (if valid)
Type	Type of event
Cause	Cause of the event condition
Details	Further explanation of the cause of the event (if applicable)

Cache Viewer

The VisualDSP++ Cache Viewer provides a means to visualize a processor's cache and find problem areas. The tool shows how instructions are being executed. You can use this valuable information to boost your application's performance.

The **Cache Viewer** window (Figure 2-68) provides a view of each instruction's execution characteristics. Cache Viewer information indicates the type of event and describes the cause of the event. Each instruction that executes from cache is marked with an **H** (hit) or an **M** (miss). Hits represent cache instructions executed without a stall. Misses identify instructions that had to be fetched from slower parts of memory because they were not found in cache.

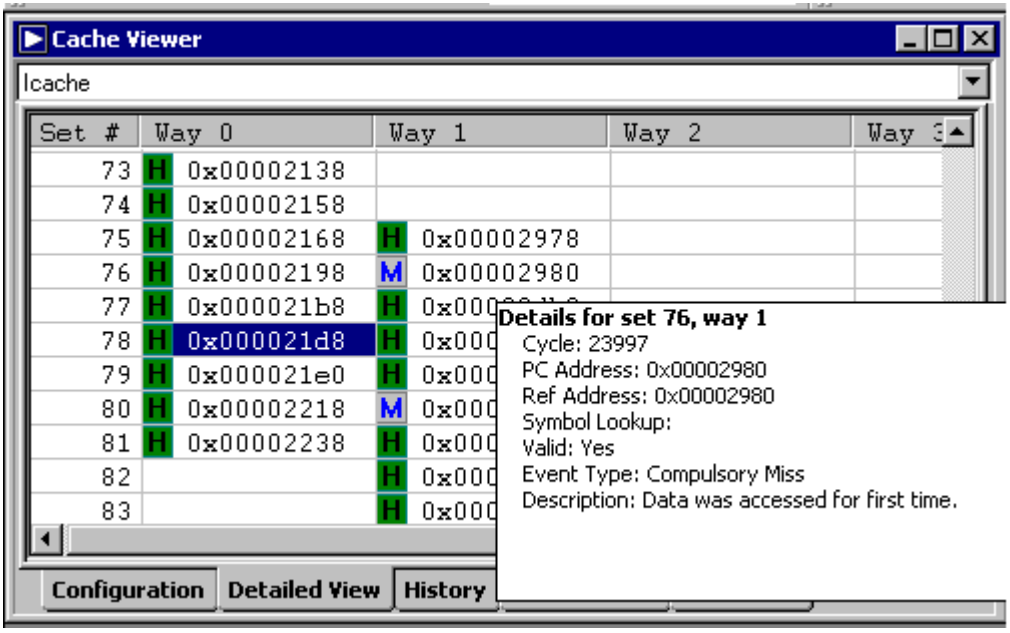

Figure 2-68. Viewing a Cache Event's Details in the Cache Viewer

Use the hit or miss information to increase an application's performance by locating instructions in the cache when they are needed. Ensuring that no cache misses are in frequently executed areas of an application (as highlighted by the profiler utility) is a critical step in optimizing your application's software performance.

As shown in Figure 2-68, the **Cache Viewer** window enables you to view the details of any cache event. These descriptive details help you understand the cause of the cache event. Use this information to isolate areas where performance can be improved.

For example, based on cache event details, you might:

- Modify an application's lay-out in memory to avoid cache thrashing
- Prefetch instructions to avoid compulsory misses
- Lock down ways in the cache to avoid a conflict miss with a frequently accessed instruction

Open the **Cache Viewer** window by choosing **View**, **Debug Windows**, and **Cache Viewer**.

The Cache Viewer consists of several tabbed pages, described in Table 2-29.

Table 2-29. Cache Viewer Pages

Page	Displays
Configuration	Cache configuration information
Detailed View	Location (set and way) of cache events
History	List of cache events
Performance	Cache performance metrics
Histogram	A plot of cache activity
Address View	Cache events on an Address vs. Cycle plot

The **Cache Viewer** window's right-click menu (described in Table 2-30) enables you to read, write, and step an *event log*, a file that records cache events.

Table 2-30. Cache Viewer Window's Right-Click Menu

Menu Option	Description
Enabled	Enables and disables collection of cache data while the target is running or stepping
Clear	Clears all displays and deletes all stored cache data
Map References	Opens the **Map References** dialog box, from which you specify the cache reference map (start address and end address)
Event Log→Read	Opens the **File Open** dialog box, from which you select and open a cache event log file. The log file data is used by the **Cache Viewer** window's **Configuration** view.
Event Log→Write	Opens the **File Save** dialog box, from which you save a cache event log file. Events are written to this log file.
Event Log→Step	Executes one cache event at a time from the cache event log file. The event displays in the **Detailed View**, **History**, and **Histogram** pages of the **Cache Viewer** window. By default, this option is enabled when a cache log file is opened for reading.
Properties	Opens the **Cache Viewer Properties** dialog box, from which you specify the **Cache Viewer** window's appearance

The event log file does not include icons. Thus, the **Cache Viewer** window's **Detailed View** page does not display icons.

Stepping enables you to execute one cache event at a time from the cache events log file. The event is displayed on the **Detailed View**, **History**, and **Histogram** pages. When stepping is configured, a check mark (✓) appears next to **Step** on the right-click menu. By default, this option is enabled when a cache events log file is opened for reading.

Configuration Page

The **Configuration** page (Figure 2-69) displays configuration information for configured cache.

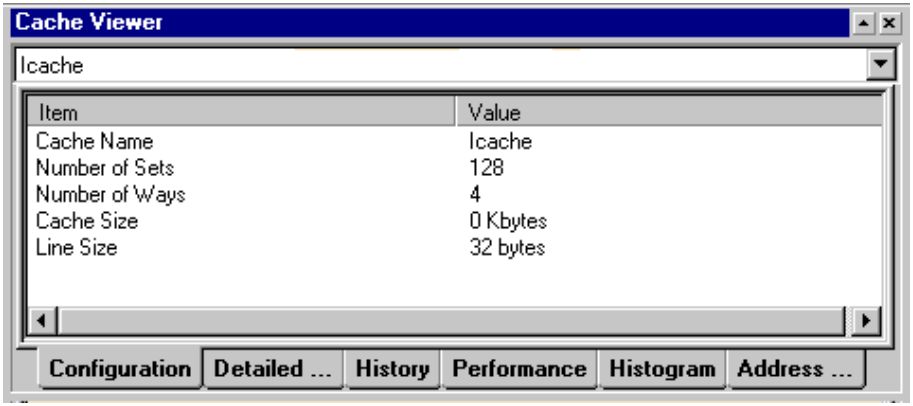

Figure 2-69. Configuration Page

The **Cache Selection** pull-down (top of dialog box) lists cache displays. If more than one cache is configured, you can use this list to change cache displays.

The **Cache Configuration** list box (below the **Cache Selection** pull-down) displays a list of items and their values. The first three items (Cache Name, Number of Sets, and Number of Ways) are required. The target may display additional items, such as Cache Size and Line Size. The list of items depends on the selection in the **Cache Selection** pull-down.

Detailed View Page

The **Detailed View** page (Figure 2-70) displays a grid depicting cache sets (rows) and cache ways (columns).

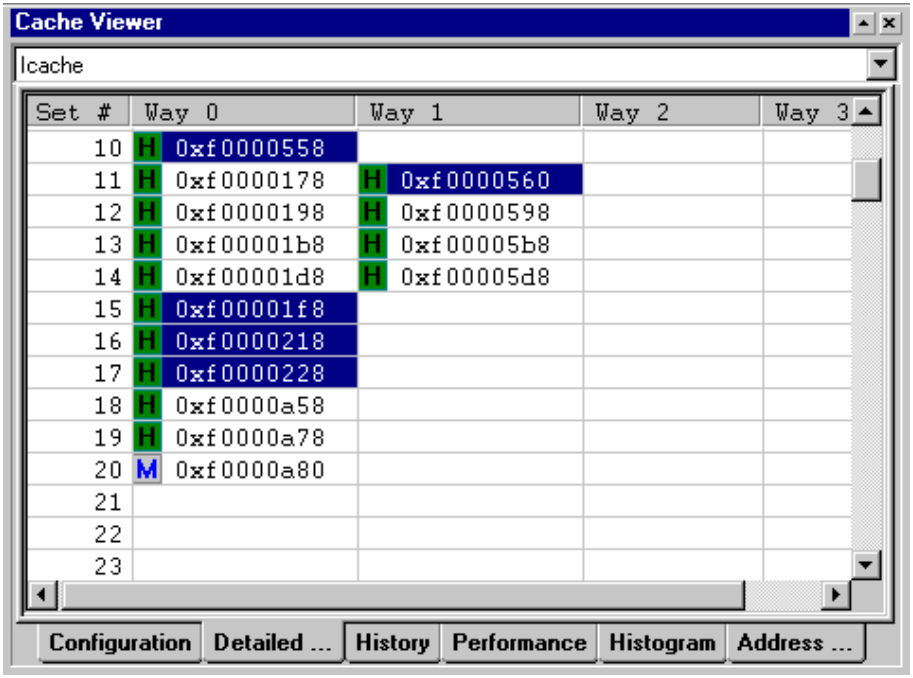

Figure 2-70. Detailed View Page

Data received from a cache event is placed in the cell corresponding to the cache set and way. The most recent events are highlighted.

Each cell has an icon and text entry. The icon indicates the type of cache event that occurred (hit or miss). Depending on the objects you choose to display, you can display details, such as reference address, PC address, cycle count, event type, event description, and so on.

You can display tooltips showing details for the most recent cache event. The appearance of a lock icon in the column header indicates that the cache way is locked.

A reference map icon in the Set # column indicates the results of the reference mapper function. Double-clicking on a cell switches the display to the history view (**History** page) for the selected cell.

History Page

The **History** page (Figure 2-71) displays detailed information for each cache event that occurred in the selected set and way.

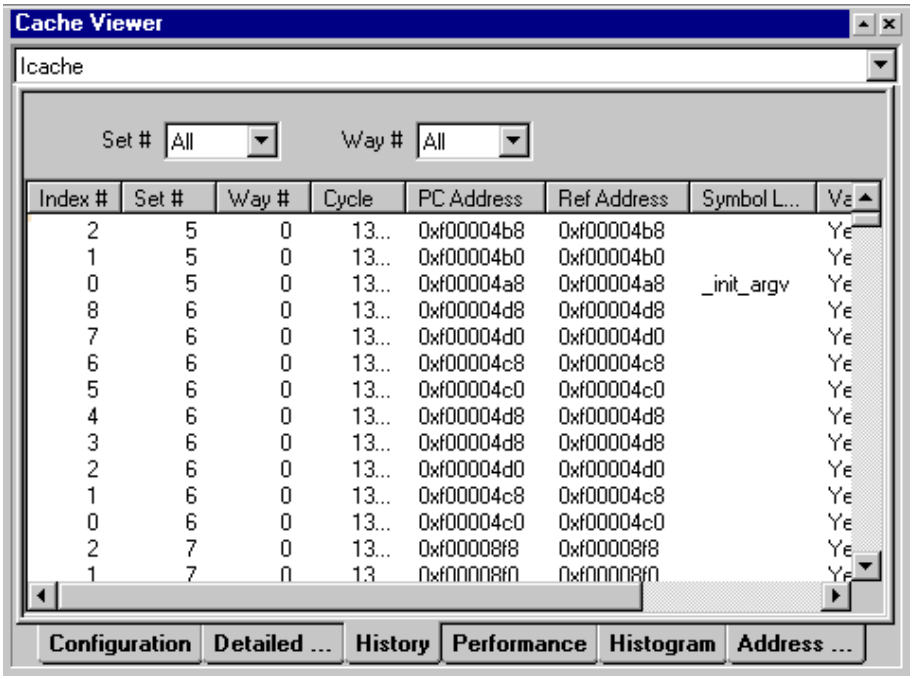

Figure 2-71. History Page

You select the set and way from the pull-down control or by double-clicking a cell on the **Detailed View** page.

You can specify the number of cache events stored. You can sort the rows by clicking on any particular column heading. An up arrow in a column heading indicates an ascending sort order, and a down arrow indicates a decending sort order.

Table 2-31 describes the history information for cache events.

Table 2-31. History Information for Cache Events

Item	Description
Index #	Shows the order in which the cache events were received. The index starts at zero and increments each time an event is received.
Set #	Displays the set number where the cache event occurred
Way #	Displays the way number where the cache event occurred
Cycle	Displays the cycle count when the cache event occurred
PC Address	Displays the PC address of the cache event
Ref Address	Displays the reference address of the cache event
Symbol Lookup	Displays the symbol name when the reference address resolves to a symbol in memory
Valid	Displays the cache line valid flag (**Yes** or **No**)
Event Type	Displays the cache event type, such as **Hit** or **Miss**
Description	Displays the cache event's description

Performance Page

The **Performance** page (Figure 2-72) shows a list of performance metrics (items and values), which are determined by the target.

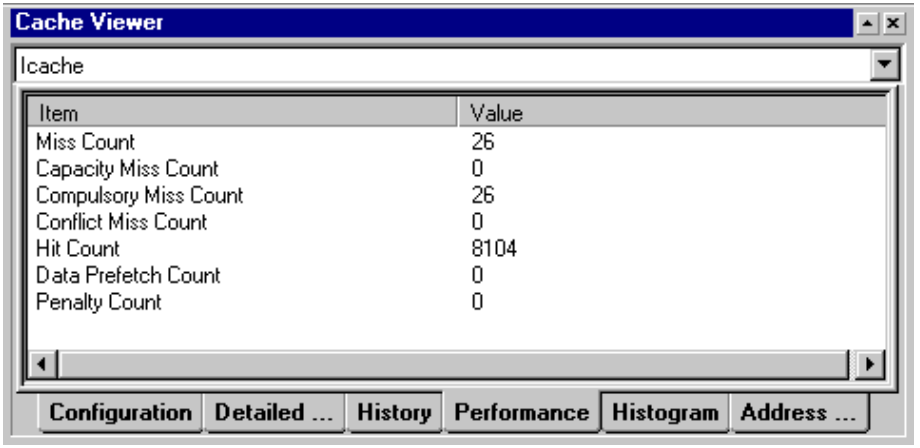

Figure 2-72. Performance Page

The target updates this list. The update rate, however, is not predetermined.

Histogram Page

The **Histogram** page (Figure 2-73) shows a plot of the total number of cache events that occurred in each cache set.

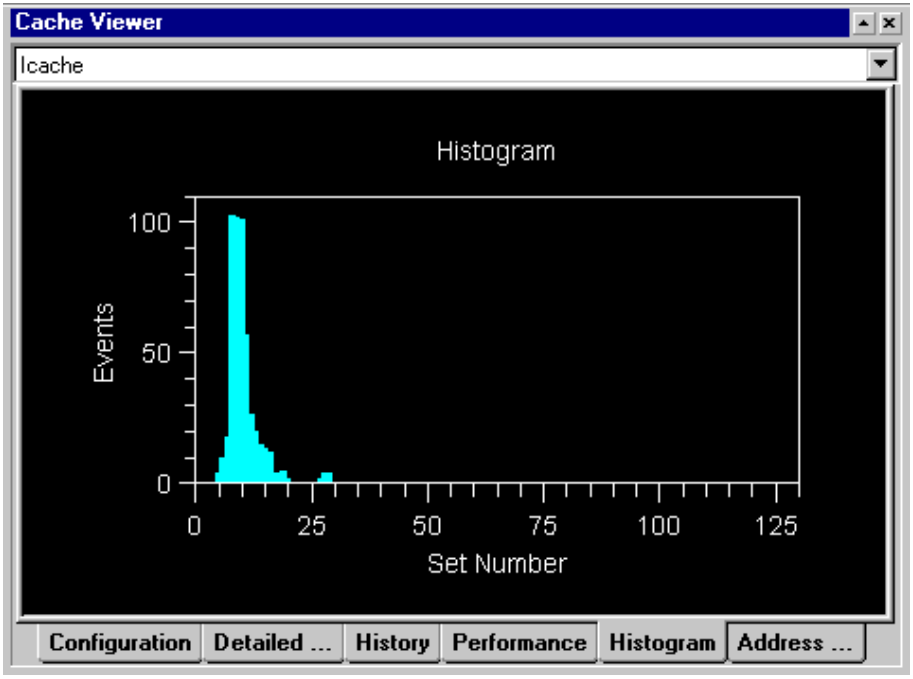

Figure 2-73. Histogram Page

A vertical line is displayed for each cache set. The line starts at zero and ends at the total number of events. Use this plot to identify the most active cache sets.

Address View Page

The **Address View** page (Figure 2-74) displays cache events on an Address vs. Cycle plot. Use this view to display the cache events for the specified addresses over time.

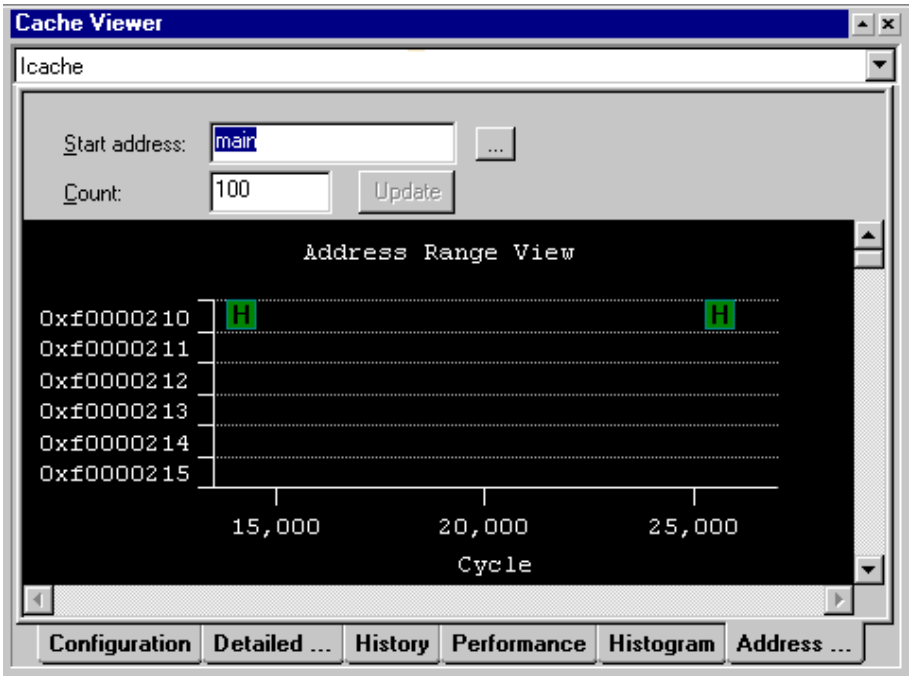

Figure 2-74. Address View Page – Address Range View

Events are displayed as icons, identical to the icons used in the detailed view. A start address and count are required. You can enter the start address as a hexadecimal value or a symbol. Click the browse (…) button to browse for a symbol.

The count determines the number of addresses displayed. After you enter a start address and count, click **Update** to display the event data. Horizontal and vertical scroll bars enable you to scroll the view.

VDK Status Window

The **VDK Status** window (Figure 2-75) is available when an executable is built with VDK support enabled. Open this window by choosing **View**, **VDK Windows**, and **Status**.

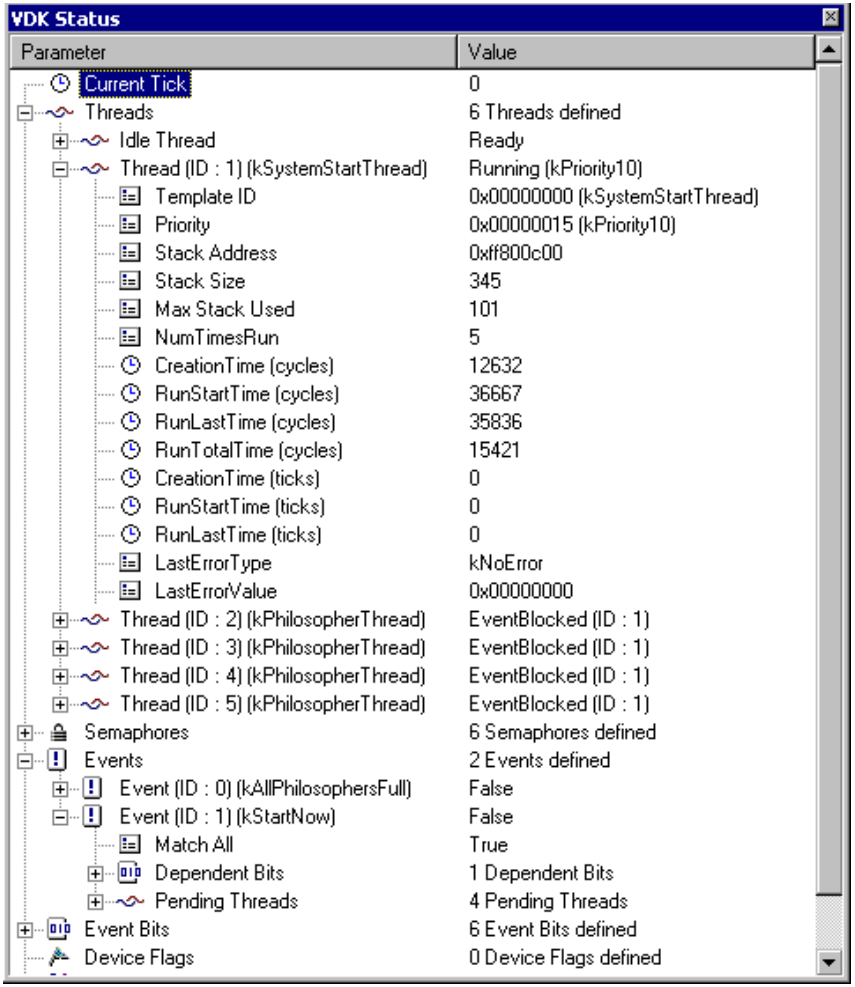

Figure 2-75. VDK Status Window

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, thread and event data are collected in a history buffer if **Full Instrumentation** has been specified for the project. When you halt a running program, the history buffer data is plotted in the **VDK State History** window, described in Figure 2-76. Some features become available only when the data cursor is enabled. Open this window by choosing **View**, **VDK Windows**, and **History**.

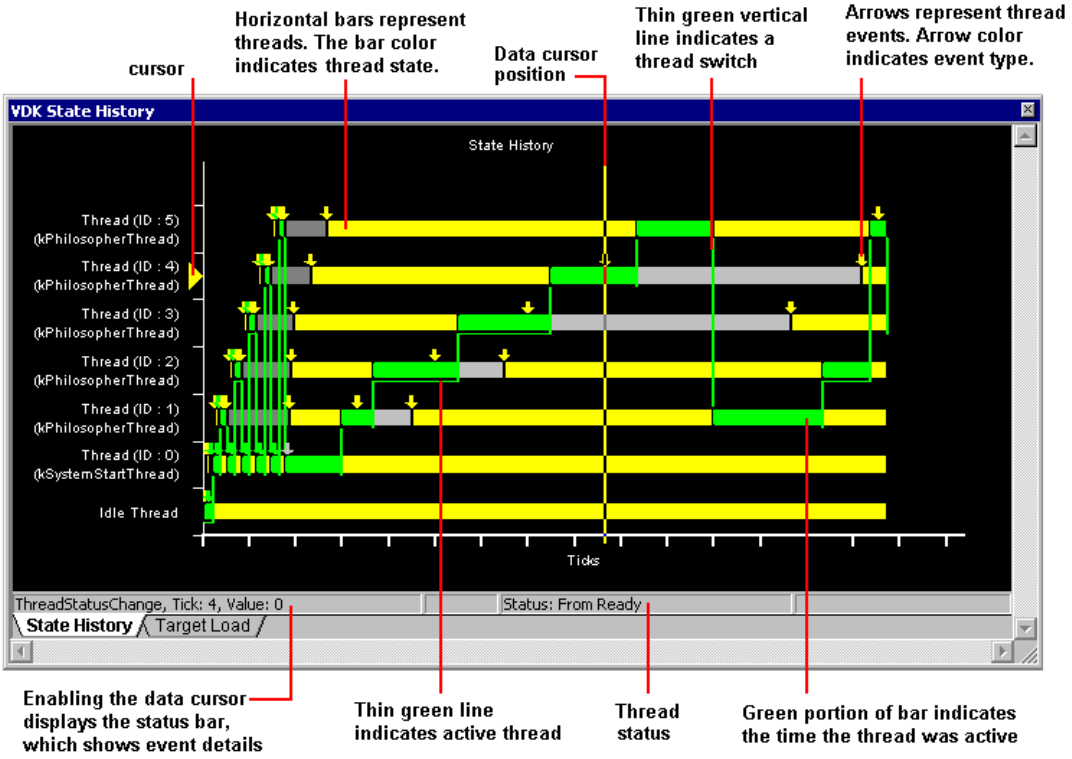

Figure 2-76. Example of a VDK State History Window

Each thread appears as a horizontal bar (thread status bar). The ThreadID and the name of the thread type appear to the left of the bar. When a thread is destroyed, the name of the thread type is no longer displayed. 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 status bar (at the bottom of the plot) on the State History page shows the event's details and thread status when the data cursor is enabled. 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 switched in. The yellow triangle to the left of the thread name indicates the currently active thread.

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 2-77) shows the percentage of time that the target spent in the Idle thread.

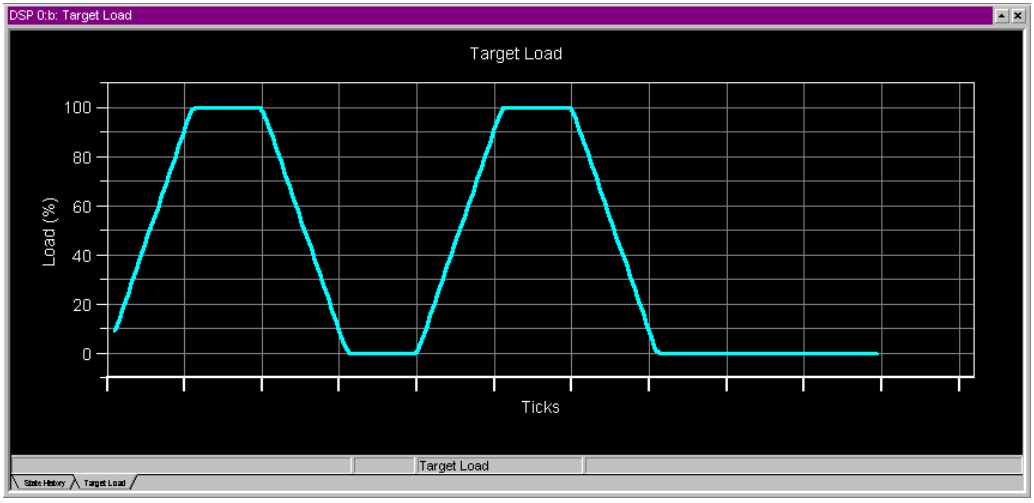

Figure 2-77. Example Target Load Window

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 2-32.

Table 2-32. File Types That Support Syntax Coloring

File Type	File Extension
Assembly	.ASM
C	.C
Linker Description Files	.LDF
C++	.CPP
Header	.H
Script	Various extensions, such as .JS and .VBS

Right-Click Menu

The editor window's right-click menu provides the commands shown in Figure 2-78.

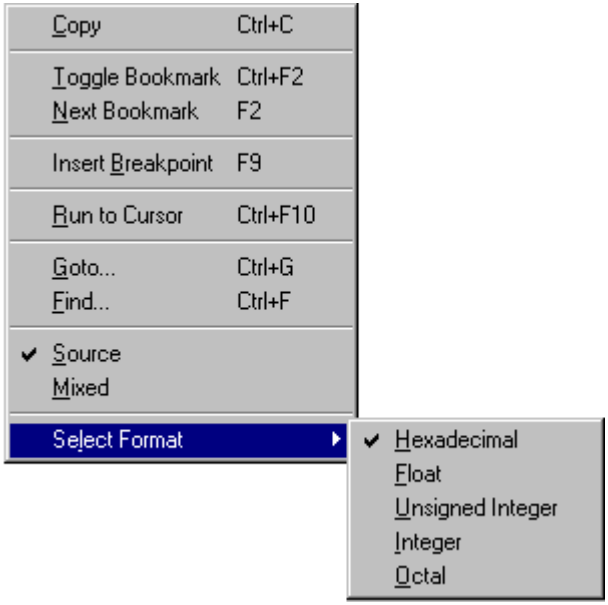

Figure 2-78. Editor Window's Right-Click Menu

Note the following.

- The available number formats under **Select Format** are DSP-dependent.
- An additional command, **Source Script**, is available when you are editing a 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 2-33 describes these icons.

Table 2-33. 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 `.DXE` 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.

Viewing an Expression

You can view an expression in different ways. When the editor window is in mixed mode, you can view an expression 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 2-79, displays C code only.

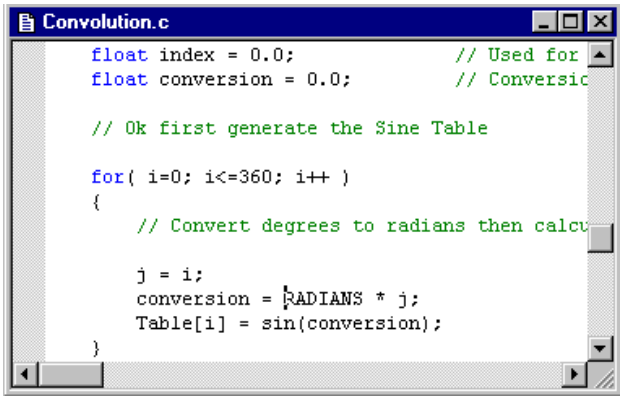

Figure 2-79. Editor Window in Source Mode Format

Mixed Mode

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

Note:

- You must compile the source file with debugging information to view the source file in mixed mode.
- You can enable and disable the display of pipeline symbols while in mixed mode.

Figure 2-80 shows an example of the mixed mode format.

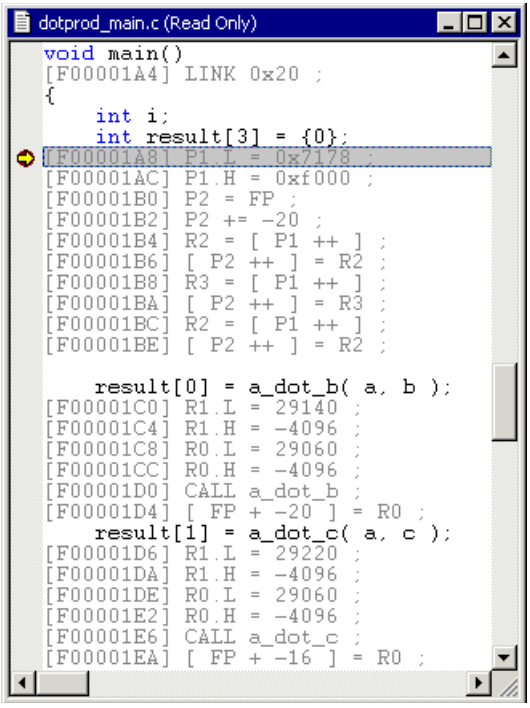

Figure 2-80. Editor Window in Mixed Mode

Expressions in an Expression Window

Table 2-34 describes the types of expressions that you can enter in an **Expressions** window.

Table 2-34. Types of Expressions Allowed in an Expressions Window

Expression	Description
Memory address	Precede memory identifiers with a `$` sign, for example: `$dm(0xF0000000)`
Register expression	Precede register names with a `$` sign, for example: `$r0`, `$r1`, `$ipend`, `$po`, or `$imask`
C/C++ statements	Use standard C/C++ arithmetic and logical operators.

The **Expressions** window displays the current value of each expression as you step through your program.

The evaluation of expressions is 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.

Number Formats

You can select the number format used to display a particular register window or memory window. The available number formats, which depend on your processor family, can include those listed in Figure 2-81.

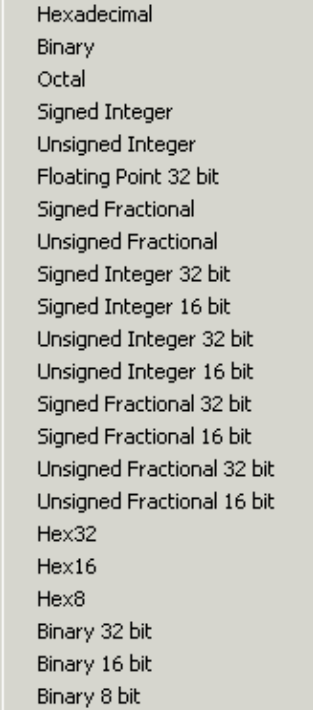

Figure 2-81. Available Number Formats

The following windows are examples of different number formats.

The window in Figure 2-82 appears in hexadecimal format.

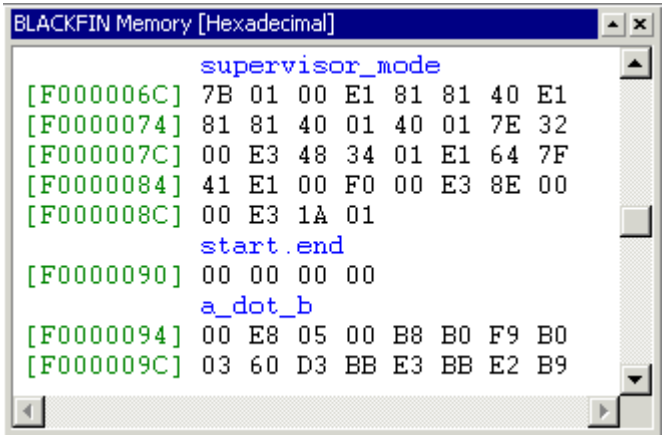

Figure 2-82. Memory Window in Hex Format

The window in Figure 2-83 appears in octal format.

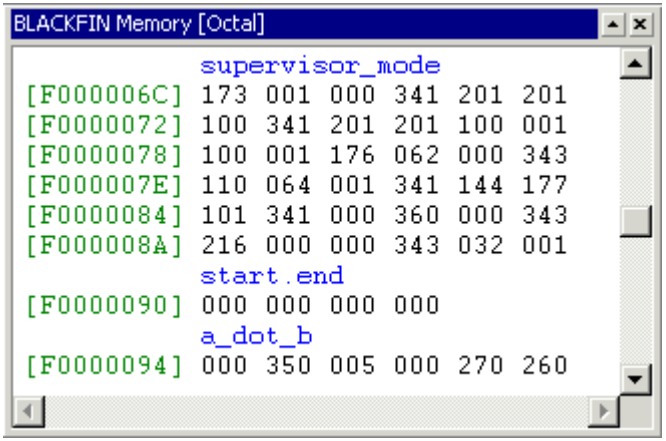

Figure 2-83. Memory Window in Octal Format

The window in Figure 2-84 appears in binary format.

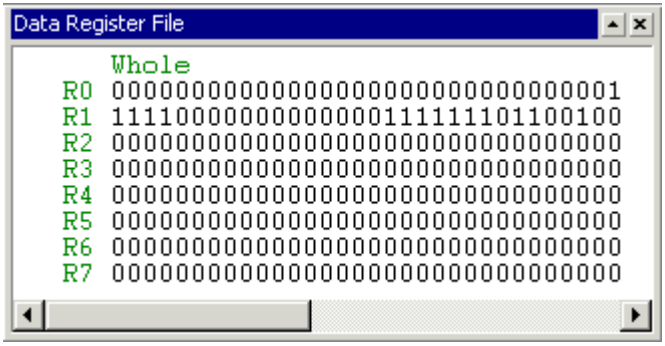

Figure 2-84. Data Register Window in Binary Format

The window in Figure 2-85 appears in signed integer format.

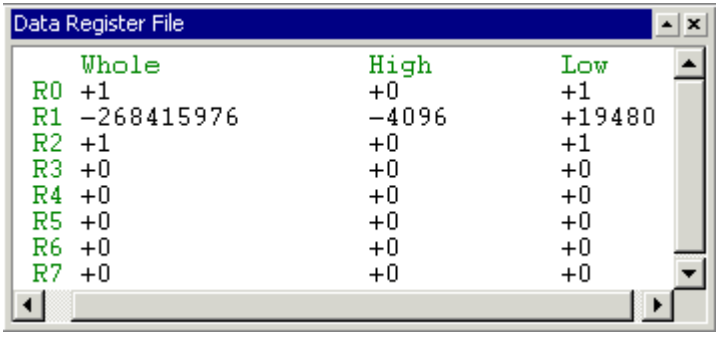

Figure 2-85. Data Register Window in Signed Integer Format

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 by choosing **View**, **Debug Windows**, **Plot**, and **New**.

Figure 2-86 shows an example of a plot in a plot window.

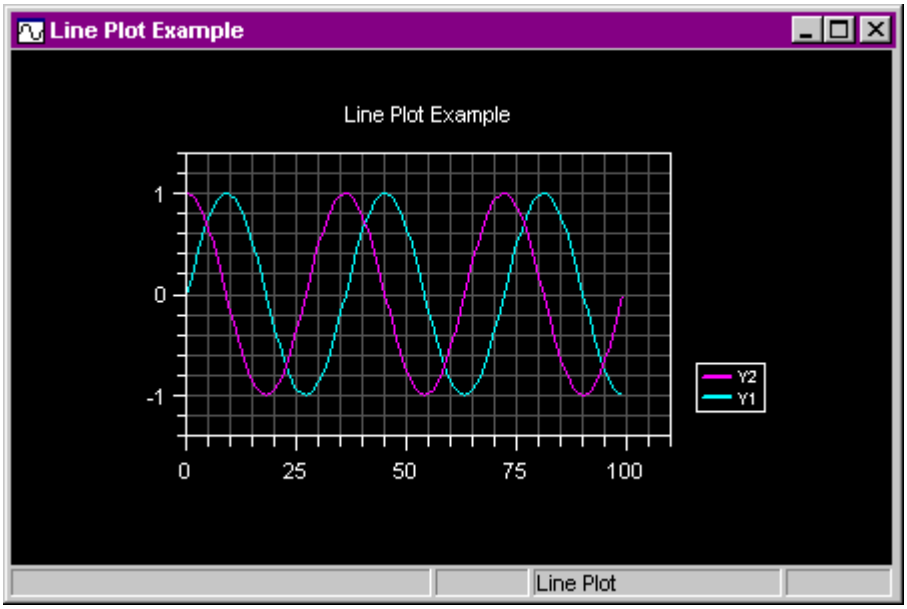

Figure 2-86. Example of a Plot Window

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 potion 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, tool 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.

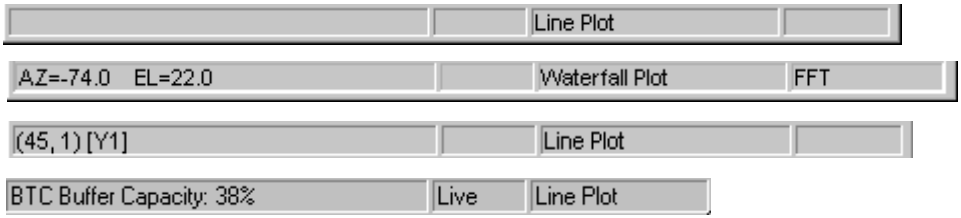

Figure 2-87. Examples of Status Bar Information for Plots

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.

When a plot window's auto-refresh mode is enabled in BTC mode, the status bar indicates current buffer capacity (for example, 89%) and data logging status.

Buffer capacity, which dynamically changes between 0 and 100%, indicates the portion of the buffer currently in use. The ideal size is a little below 100%. Readings of 100% indicate that data loss.

Table 2-35 describes the data logging status indicators in a plot window.

Table 2-35. Data Logging Status Indicators in a Plot Window

Status	Indicates
Record	Real-time data being displayed is also being saved (logged) to a `.BIN` file.
Live	Data is being displayed in real time.
Playback	A previously saved data (log) file is being viewed.

Tool Bar

The plot window's toolbar provides buttons for recording and playing back streaming data and a box for specifying streamed data (`.BIN`) file names.

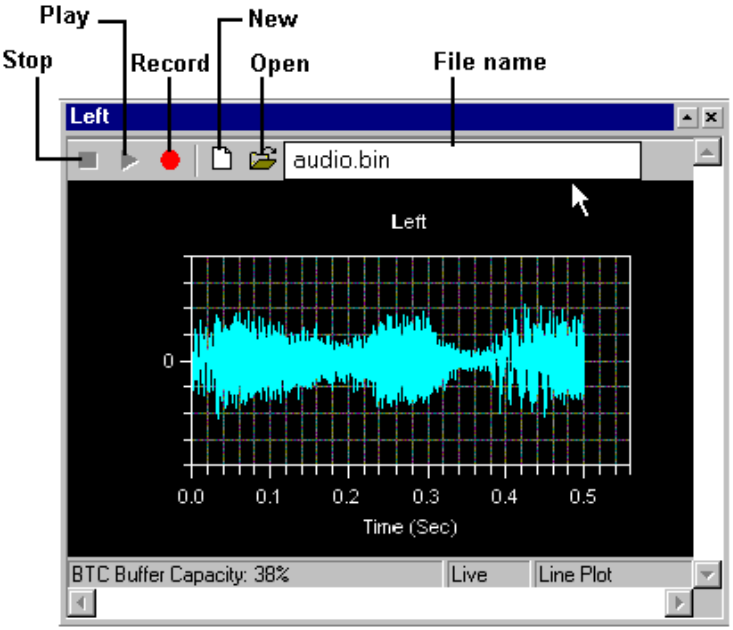

Figure 2-88. Plot Window's Toolbar

Right-Click Menu

The plot window's right-click menu is shown in Figure 2-89.

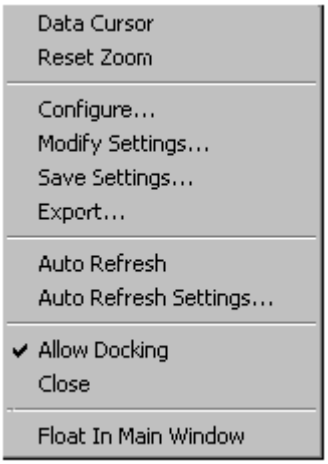

Figure 2-89. Plot Window's Right-Click Menu

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 2-36 on page 2-127.

Table 2-36. Plot Window Operations

Feature	Description
Data Cursor	Displays the data value associated with the position of the plot window's data cursor. View the value on the left side of the plot window's status bar. Press the keyboard's arrow keys to move around on the graph.
Reset Zoom	Resets the plot window to its initial full-scale display
Configure	Opens the **Plot Configuration** dialog box, from which you can add, remove, or modify data sets. You can also change the plot type and rename the plot.
Modify Settings	Opens the **Plot Setting** dialog box from which you can customize the plot's appearance. You can specify plot settings (grids, colors, margins, fonts, axes, and so on) and settings for each data set (data processing).
Save Settings	Saves plot configuration settings for future use. The configuration is stored, but not the data. You can retrieve settings (`.VPS` file) and load new plot data.
Export	Exports 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.
Auto Refresh	Enables a plot window to refresh automatically based on settings that you specify. The auto-refresh timer starts. Streaming data is read and displayed. When you deselect this option, the timer is stopped and streaming data is not processed.
Auto Refresh Settings	Enables you to configure options that control auto-refresh settings for plot windows. These settings determine the method in which memory is transferred.

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 2-90 shows statistics displayed for a portion of audio data.

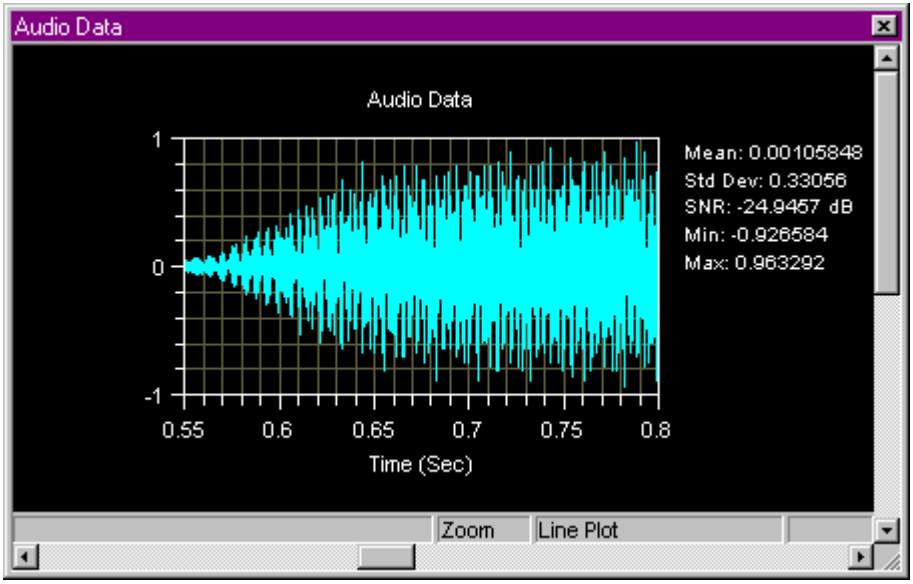

Figure 2-90. Statistics Displayed for a Portion of Audio Data

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

Plot Configuration

A plot configuration comprises two parts: data values and presentation (configuration) settings.

A plot window must contain at least one *data set*, a series of data values in DSP memory. You create data sets in the **Plot Configuration** dialog box, shown in Figure 2-91.

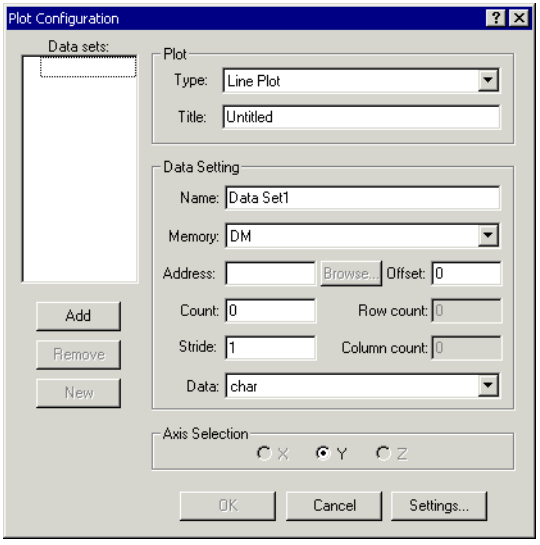

Figure 2-91. Plot Configuration Dialog Box

You specify the type of plot (for example, waterfall), the memory location, the number of values, the axis associated with each data set, and other options that identify the data. Note that 3-D plots require additional specifications for row and column counts.

The **Settings** button enables you to configure presentation options (such as titles, grids, fonts, colors, and axis presentation) for each data set. You can recall a plot from a saved settings file (`.VPS`). VisualDSP++ uses these settings and reads DSP memory to 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 can invoke as follows.

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

The **Plot Settings** dialog box provides the tabs shown in Figure 2-92.

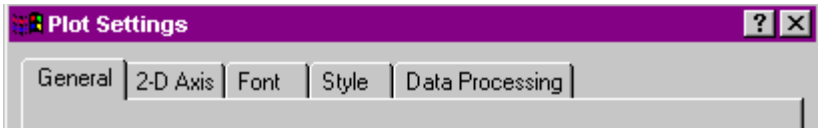

Figure 2-92. Tabs in the Plot Setting Dialog Box

Options on the tab 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 2-93.

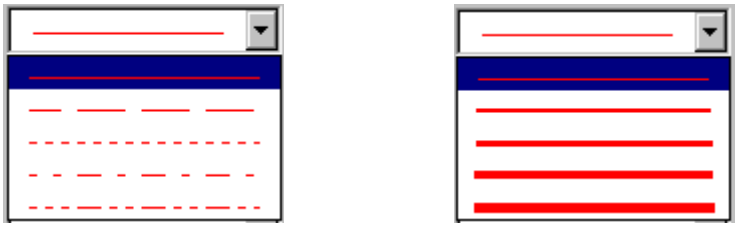

Figure 2-93. Line Styles

In addition to the many presentation options, you can select a rectangular area, as shown in Figure 2-94, and zoom in on it.

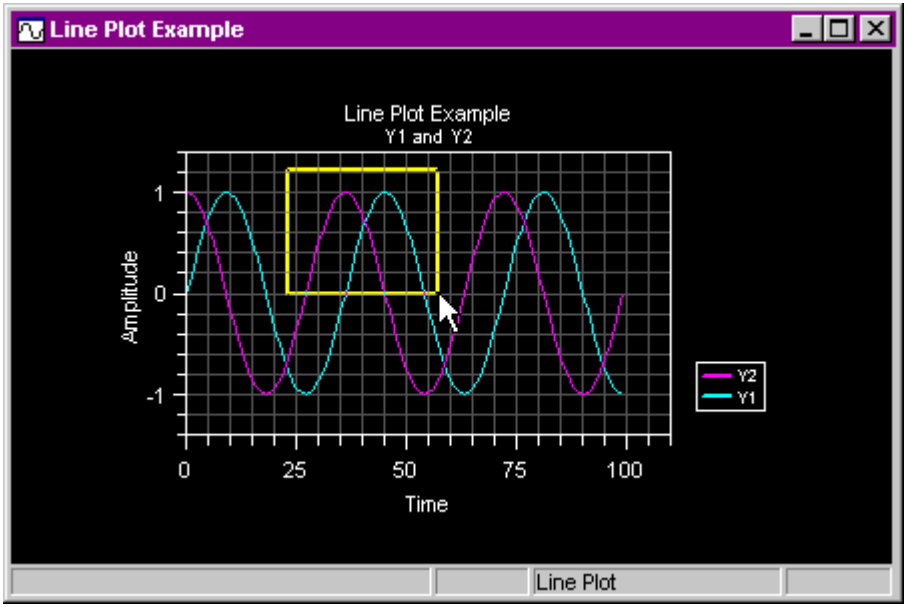

Figure 2-94. Zooming In on a Selected Area

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 2-37.

Table 2-37. 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 traces, 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.
- Configure image attributes such as number of pixels, bits per pixel, and image format
- 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.

To open the **Image Viewer** window, choose **View**, **Debug Windows**, and **Image Viewer**.

The **Image Viewer** window, shown in Figure 2-95, renders the image and provides scroll bars and buttons for zooming in and out.

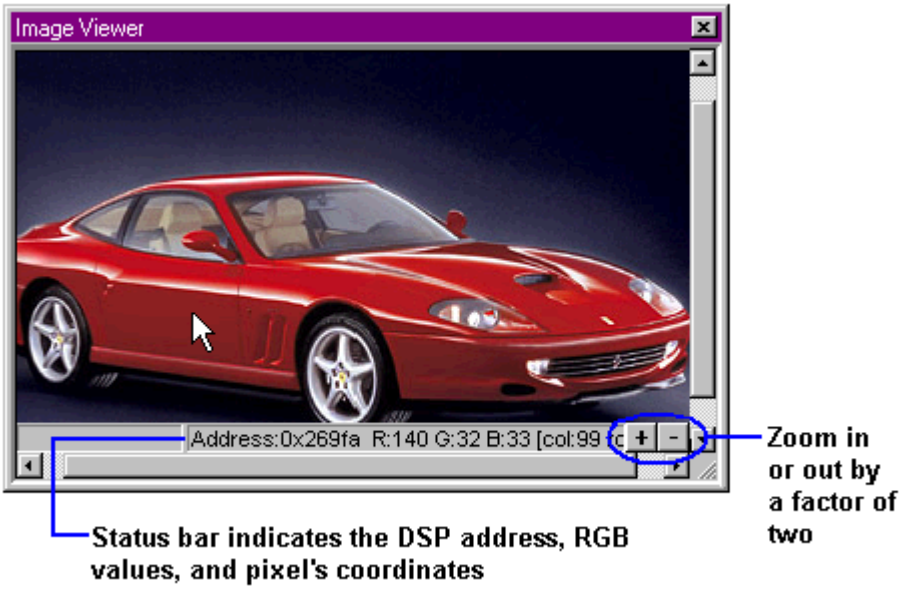

Figure 2-95. Image Viewer Window

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 for color images, intensity values for gray-scale images
- Pixel coordinates (column and row)

Pixel color depth is 24 bits for color images and 8 bits for gray-scale images.

Right-Click Menu

Figure 2-96 shows the **Image Viewer** window's right-click menu.

Figure 2-96. Image Viewer Window's Right-Click Menu

Table 2-38 describes the menu commands.

Table 2-38. 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 2-38. 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 2-39 describes the buttons and fields in the **Image Configuration** dialog box.

Table 2-39. 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)
Pack data	Specifies the storage of continuous bytes of RGB pixel data in target memory. When this option is not selected, each component of the pixel data is stored in a separate memory location.

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 2-40 describes the buttons and fields in the **Gamma Correction** dialog box.

Table 2-40. 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 2-41.

Table 2-41. 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 debug sessions. The topics are organized as follows.

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 *simulator* is software that mimics the behavior of a DSP chip. Simulators are used to test and debug processor code before a DSP chip is manufactured. An *emulator* 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++.

Simulation vs. Emulation

When connected to a simulator session, you may open as many sessions as your system's memory can handle.

When connected to actual hardware through an emulator, you can have only **one** debug session connected to one emulator at any time. If multiple emulators are installed and are connected to multiple target boards, one debug session may be connected to each individual emulator.

When connected to a JTAG emulator, one debug session only may be connected to each physical target/emulator combination. Otherwise, contention issues may arise. Upon switching to a different session, VisualDSP++ detaches from the old session before attaching to the new session.

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.

In addition to software breakpoints, you may also use *hardware breakpoints* in an emulator debug session.

Watchpoints

Watchpoints are like breakpoints. Watchpoints, however, trap on a specified condition.

You can set watchpoints on registers, stacks, and memory ranges. When the condition is reached, program execution halts and all windows update.

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. Multiprocessor debugging is available only for processors such as the ADSP-2192-12 and ADSP-BF561, which support multiprocessing.

Setting Up a Multiprocessor Debug Session

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++ 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 VisualDSP++ Configurator.

If you are running a multiprocessor simulator debug session, select the desired configuration from the **Platforms** list in the **New Session** dialog box. 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** page enables you to view the status of each processor and switch processor focus.

  The **Group** page 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.

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.

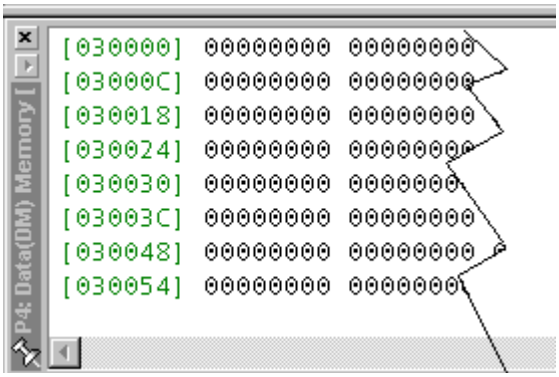

Figure 3-1. Pinned Window in a Multiprocessor Debug Session

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 Profiles and Linear Profiles

VisualDSP++ provides two profiling methods that measure program performance by sampling the target's `Program Counter` (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 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 run-time 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, as shown in Figure 3-2.

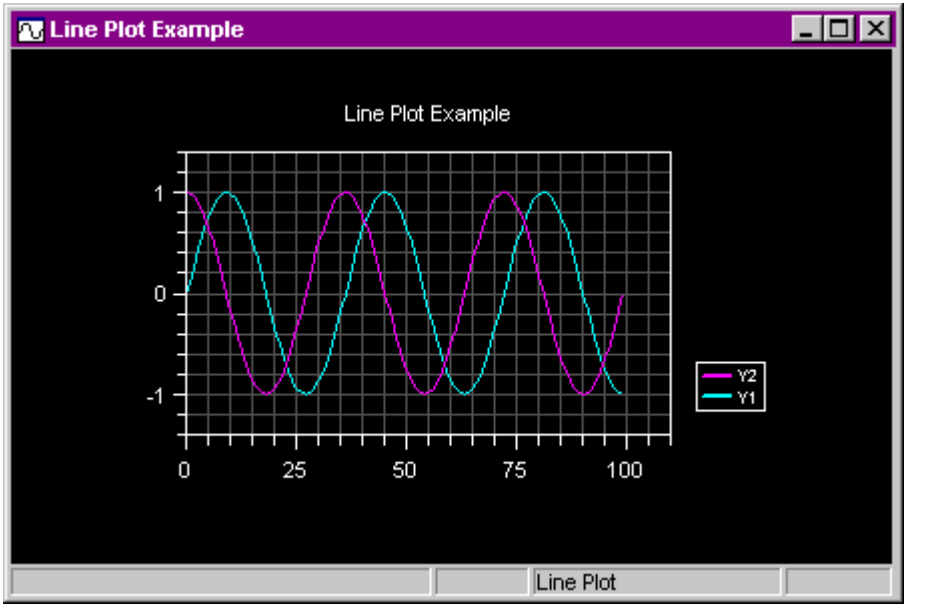

Figure 3-2. Example Plot Window Displaying DSP Memory

You can visualize the DSP memory data and process it by using a data processing algorithm. You can choose from multiple plot types and can specify the plot's data and presentation.

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.

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 (`.DXE`) file.

This debugging feature saves time, as 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.

Use the commands described in Table 3-2 on page 3-13 to control program execution.

Table 3-2. Commands Used to Control Program Execution

Command	Description
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	Stops program execution. All windows are updated after the processor 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	(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` 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 3-3.

Table 3-3. Breakpoint Status Symbols

Symbol	Indicates
	An enabled (set) breakpoint
	A disabled breakpoint (recognized, but cleared)

Using Unconditional and Conditional Breakpoints

You can configure a breakpoint to occur when the `Program Counter` 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

Before you even have the processor, you can use interrupts and data streams within VisualDSP++ to simulate the processor's behavior.

Interrupts

Use interrupts to simulate external interrupts in your program. When you use interrupts with watchpoints and streams, your program simulates real world operation of your DSP system.

Input/Output Simulation (Data Streams)

In many products, processors 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 processors excel in these roles.

You can use data streams to transmit data between:

- A device and a file
- A device and a device
- A device in one processor and a device in another processor in a multi-processor system

Using background telemetry channel (BTC) technology, VisualDSP++ permits the streaming of data from a target DSP without halting the DSP. This capability applies to both simulation and emulation targets.

The plot window receives and displays a stream of data from DSP memory. If the target supports background telemetry, the plot window reads memory and updates the display without halting the target. Otherwise, the plot window halts the processor, reads memory, updates the plot, and resumes the processor.

The plot window allows data to be streamed to (or from) a binary data file. You can convert the data file into ASCII format to input it to other applications such as MATLAB and Excel.

The DSP application may collect and transfer data in four different ways:

- Sampling a test point over time
- Transferring a data array over BTC at a specified point in the DSP application
- Using `GetMem()` directly
- Periodically halting the target to read memory

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.

Plots

VisualDSP++'s data plotting capability helps you to visualize data in the processor's memory.

Plot Types

You specify a plot as one of the plot types described in Table 3-4.

Table 3-4. Available Plot Types

Plot Type	Description	Requires
Line	Displays points connected by a line	Y value for each data point
X-Y	Similar to a line plot, but also uses X-axis data	X value and Y value for each data point
Constellation	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	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 processor memory.

Line Plots

A line plot (shown in Figure 3-3) displays a range of processor memory values connected by a line. The values read from processor memory are assigned to the Y-axis. The corresponding X-axis values are automatically generated.

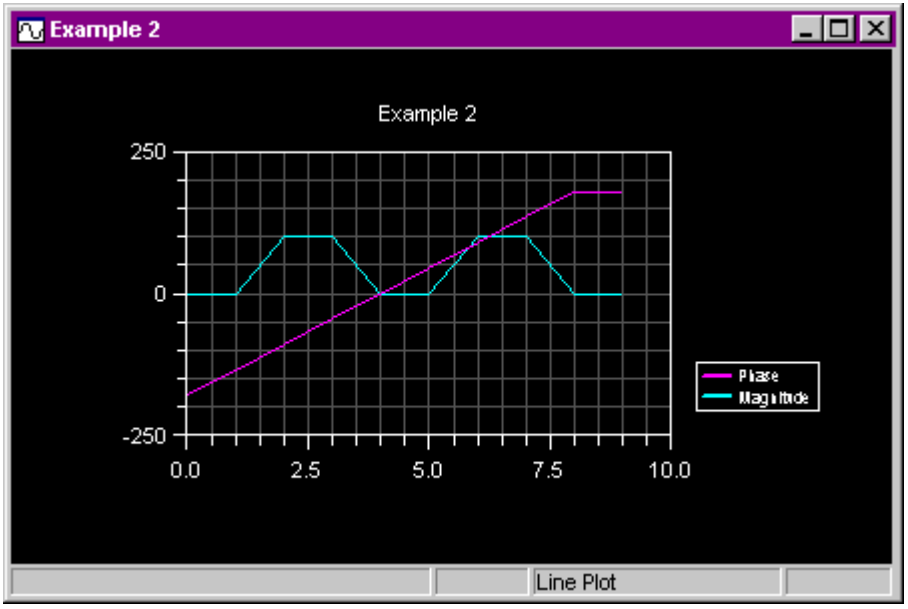

Figure 3-3. Line Plot Example

You can plot multiple data sets on a single graph.

X-Y Plots

An X-Y plot (shown in Figure 3-4) 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.

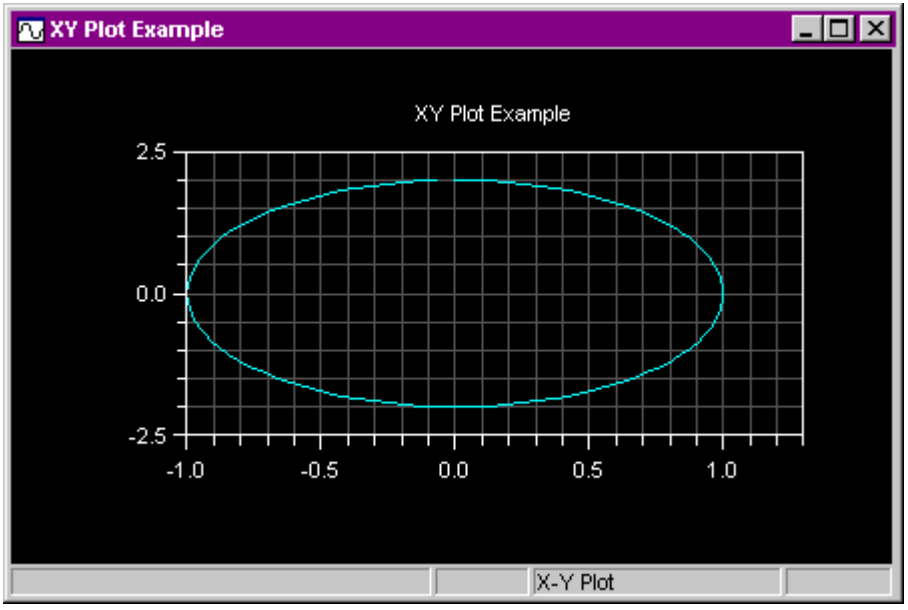

Figure 3-4. X-Y Plot Example

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 3-5) displays a symbol at each (X,Y) data point.

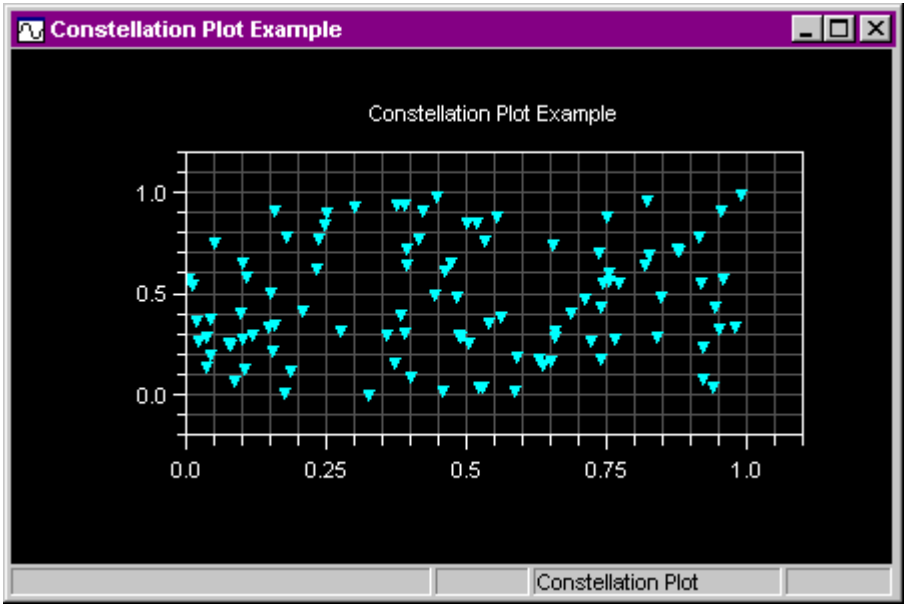

Figure 3-5. Constellation Plot Example

The X and Y data are specified separately in a user-defined processor memory location. The number of X and Y points must be equal.

Eye Diagrams

An eye diagram plot (shown in Figure 3-6) is typically used to show the stability of a time-based signal. The more defined the eye shape, the more stable the signal.

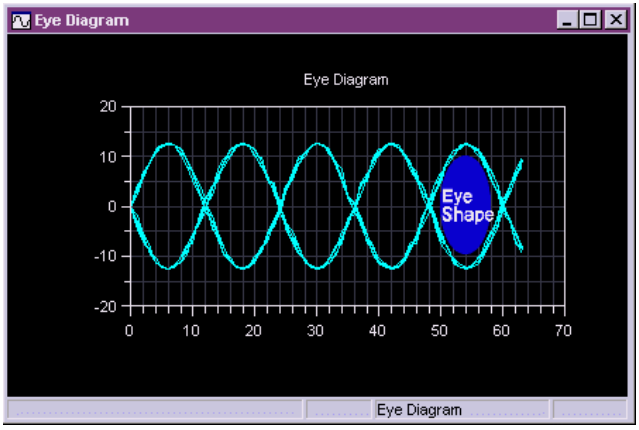

Figure 3-6. Eye Diagram Plot Example

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 is 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 3-7) is typically used to show the change in frequency content of signal over time.

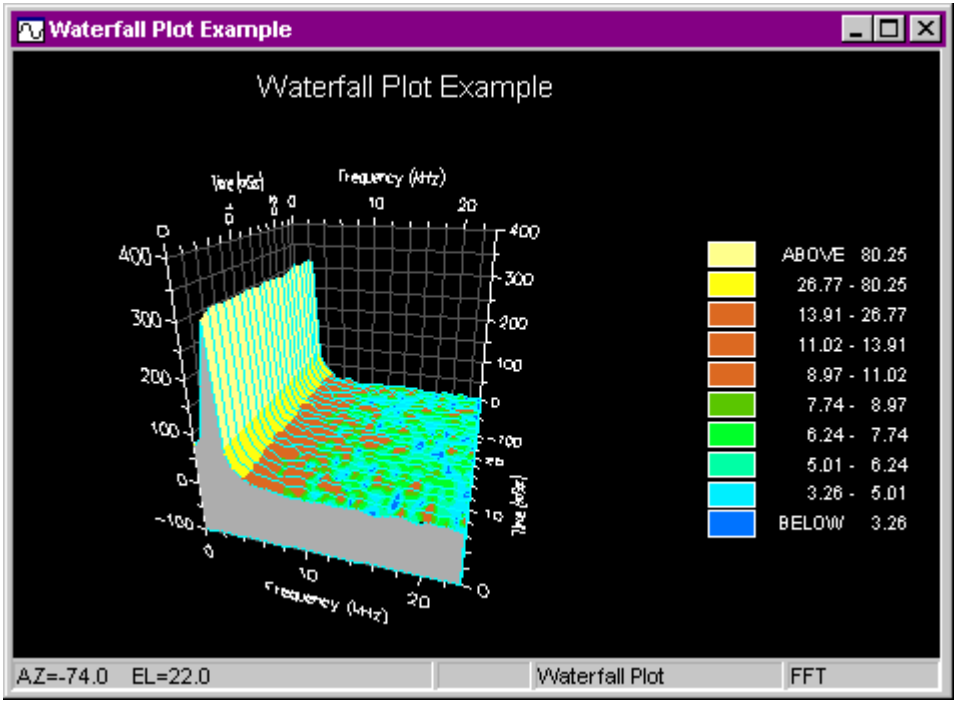

Figure 3-7. Waterfall Plot Example

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–*N*).

Figure 3-8 shows a grid of sampled data.

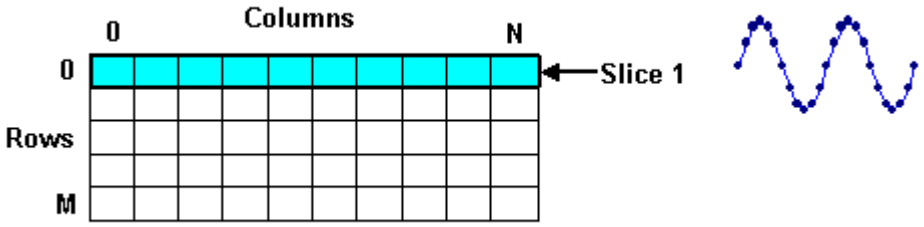

Figure 3-8. Grid of Sampled Data

The next time signal is sampled and stored (in row 1, columns 0–*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 3-9) displays the same data as a 3-D waterfall plot, except in a 2-D format.

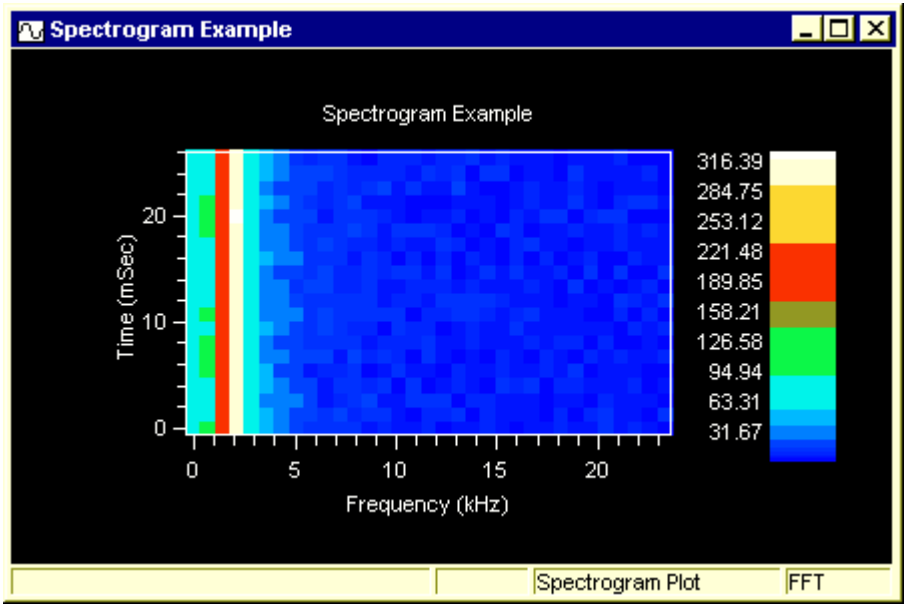

Figure 3-9. Spectrogram Plot Example

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.

Flash Programmer

The VisualDSP++ Flash Programmer provides a convenient, generic interface to numerous processors and flash memory devices. This utility simplifies the process of changing data values on a flash device and modifying its memory. You no longer have to remove the flash memory from the board, use a separate Flash Programmer, and then replace the flash.

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 processor 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 processor, you can use the processor to program the flash device.

Flash Programmer Functions

Use the Flash Programmer to:

- Load a flash algorithm (driver) program onto the processor at any time
- Obtain the flash manufacturer and device codes
- Reset the flash
- Program the flash from an Intel Hex data file
- Fill portions of flash memory with a value and quickly “punch-in” data
- Erase the entire flash

- Erase a single sector
- Send custom commands to the driver for batch processes or user-defined behavior

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 utility, you must first load a flash driver onto the processor. The driver is a DSP application that interfaces with the Flash Programmer and performs all the interaction with the flash device. Analog Devices supplies sample drivers for use on certain EZ-KIT Lite™ evaluation systems.

Flash Programmer Window

Figure 3-10 shows the **Flash Programmer** window.

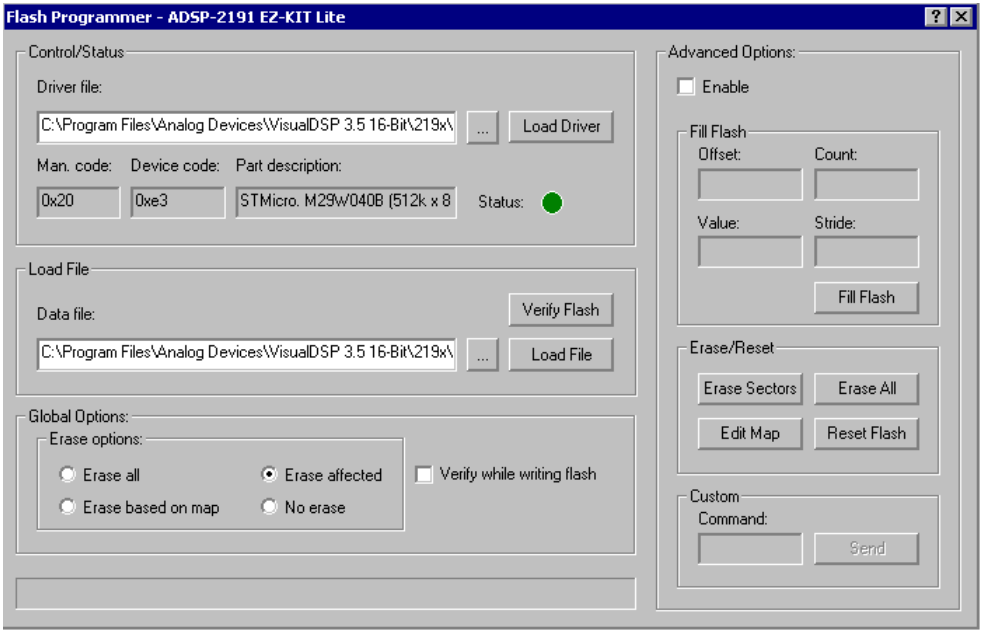

Figure 3-10. Flash Programmer Window

Table 3-5 describes the fields and buttons in the Flash Programmer window.

Table 3-5. Flash Programmer Window Controls

Control	Description
Control/Status	
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 processor
Man. code	Specifies 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.
Load File	
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.
Verify Flash	Compares the flash's current contents against an Intel Hex data file. This procedure assumes that a driver has been loaded
Load File	Loads the specified data file onto the flash memory device
Global Options	
Erase all	Erases all of the device's memory
Erase based on map	Erases based on a sector map

Table 3-5. Flash Programmer Window Controls (Cont'd)

Control	Description
Global Options (Cont'd)	
Erase affected	Erases only the portion of flash affected by the write
No erase	Does not erase flash
Verifying while writing flash	Verifies each performed write by ensuring that what was written matches what is read back from flash
Advanced Options	
Enable	Enables advanced features Selecting this control enables the fields and buttons below it. Clearing this control disables (grays out) the fields and buttons.
Offset	Specifies the first address in which to place data
Value	Specifies the data value to be written
Count	Specifies the number of locations to be written
Stride	Specifies the number of locations to skip between each write. Typically, this is 0x1. Entering 0x2 specifies every other location.
Fill Flash	Loads the specified data value onto the flash memory device
Erase Sector	Erases the specified sector from the flash device
Edit Map	Opens the Sector Map dialog box, which enables you to select the sector (or sectors) you want to erase. This function assumes that a driver has been loaded. **Note**: Erasure is constricted by the selected Global Erase Option.
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

The following terms are important toward understanding VisualDSP++.

Application Programming Interface (API) functions

A set of functions available to an applications programmer. These functions, which are part of an application, can be accessed by other applications. For VDK, 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.

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 toby 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 transmission path between two communicating locations, usually the smallest subdivision of a transmission system. For VDK, 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.

COFF

Common Object File Format. VisualDSP++ does not support COFF formatted files.

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. 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.

CROSSCORE™

Analog Devices DSP development tools, which provide easier and more robust methods for engineers to develop and optimize systems by shortening product development cycles for faster time-to-market. CROSSCORE components include the VisualDSP++ software development environment and EZ-KIT Lite evaluation systems and Emulators for rapid on-chip debugging.

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-BF535s. Another example of a debug session is an ADSP-BF535 EZ-KIT Lite target connected to an ADSP-BF535 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 processor 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 intermediate and output file directories

These 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."

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 processor. 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 API functions.

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 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 processor 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, Event-Bit value, and combination type. Events are statically allocated and enumerated at run-time. An event cannot be destroyed, but its properties can be changed (see Blueelem text).

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-BF53x, and ADSP-TSxxx processors.

executable file

A file or program that has been written and built in VisualDSP++

EZ-KIT Lite

A development board, software, and cable for evaluating a particular processor. The kit includes fundamental debugging software to facilitate architecture evaluations via a PC-hosted tool set. Use the kit to evaluate Analog Devices DSPs, learn about DSP applications, simulate and debug applications, and prototype applications.

focus

Refers to the active processor in a multiprocessor (MP) session that you are debugging

ICE

In-Circuit Emulator. Analog Devices offers emulators that provide non-intrusive target-based debugging of DSP systems. An emulator can single-step or execute a DSP at full speed to enable you to view or alter DSP register and memory contents.

interrupt

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.

JTAG ICE configurator

See "VisualDSP++ configurator."

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.ex`, 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.

Linker Description Files (LDFs)

LDFs describe the target system and maps your program code within the system memory and processors. The `.LDF` file creates an executable file using the target system memory map and defined segments in your source files.

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 processor'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

For VDK, 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.

non-bootable PROM image file

Splitter output, consisting of PROM files that cannot be used to boot-load a system

outdated file

A file that has been edited since the last time it was built

payload

For VDK, 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.

pipelining

A feature that enables you to analyze and tune your code for optimal performance. For ADSP-219x, Blackfin, and TigerSHARC processors, VisualDSP++ provides a simulation-only debugging window (**Pipeline Viewer**) to help you visualize the pipeline by displaying pipeline stalls and aborts. For SHARC processors, the **Disassembly** window displays symbols (F, D, or E) to indicate an instruction's pipeline stage.

platform

The device with which a target communicates. For simulation, a platform is typically one or more processors of the same type. For emulation, you specify the platform with the VisualDSP++ 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-KIT Lite board, the platform is the board in the system on which you wish to focus. Note that JTAG emulation does not apply to ADSP-218x DSPs.

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.

profiling

A technique used during simulation to examine program execution within selected ranges of code. Profiling helps you determine: percentage of time spent executing instructions, number of clock cycles spent executing instructions, number of instructions executed, and the number of times memory is read or written.

The profiler is non-intrusive. It does not report on execution within a called function (daughter function). You use profiling to monitor program memory. By watching one or more profile ranges, you can find areas of code that may be optimized for better performance. A profile session must include one memory range at a minimum. For each range, you must specify a start and end address. You can use symbols or hexadecimal numbers to represent addresses.

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 display

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 these 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 and clears 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

For VDK, a scheduling scheme whereby all threads at a given priority are given processor time automatically in fixed duration intervals. Round-robinpriorities are specified at build time.

scheduler

For VDK, 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.

scripting

You can interact with the IDDE by using a single command or a script file. Scripting languages include VBScript, JavaScript, and Tcl. Output displays in the Console view of the Output window. The output is also logged to the VisualDSP_log.txt file.

semaphore

For VDK, 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 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 by 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

For VDK, 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.

splitter

A PROM splitter utility that transforms an executable file into a non-boot-loadable image. This file is loaded onto external DSP memory.

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 run-time overhead. See also "linear profiling." JTAG emulation does not apply to ADSP-218x DSPs.

stepping

A technique for moving through source or assembly code to observe instruction execution

streams

A debug tool used during simulation to drive other devices or take part in processing a subset of data. Use streams to simulate data input and output.

symbols

Labels for sections, subroutines, variables, data buffers, constants, or port names. For more information, refer to the related build tool documentation.

system configurator

For VDK, 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.”

threads

For VDK, 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 run-time. 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 processors do not support traces.

unscheduled regions

For VDK, 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.

VDK

See "VisualDSP++ Kernel (VDK)."

VisualDSP++

An Integrated Development and Debugging Environment (IDDE) for Analog Devices DSP development tools

VisualDSP++ Configurator

Previously called JTAG ICE Configurator, use this utility to describe the hardware to VisualDSP++ when you connect to a JTAG emulator session. VisualDSP++ requires this description to set up the debug session.

VisualDSP++ Kernel (VDK)

The RTOS kernel from Analog Devices, VDK a software executive between DSP algorithms, peripherals, and control logic. The kernel is integrated with the Integrated Development and Debugging Environment (IDDE), assembler, compiler, and linker programs into the DSP development tool chain.

VDK is supported on the ADSP-219x, SHARC, TigerSHARC, 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	Linker command source file is a text file that contains commands for the linker in the linker's scripting language
.IS .PP .S	Intermediate files	Preprocessed assembly files generated by the preprocessor
.DOJ	Assembler Object file	Binary output of the assembler
.DLB	Archiver file	Archiver's binary output in ELF format
.H	Header file	Dependency file used by the preprocessor, and a source file for the assembler and compiler
.H	Loader output	For ADSP-2192-12 DSPs, the C-language header file output of the boot loader utility (BLU)
.DAT	Data file	Dependency file used by the assembler for data initialization

Table A-1. Files Used with VisualDSP++ (Cont'd)

Extension	Name	Purpose
.DLO .DXE .OVL .SM	Debugging files	Binary output files from the linker in ELF/DWARF format
.MAP	Linker Memory Map file	Optional output for the linker. This text file contains memory and symbol information for executable files.
.TCL .TC8	Tools Command Language files	Tcl scripting language files used to script work
.OBJ	Assembled Object file	(Previous releases only, replaced by .DOJ) Output of the assembler
.LST	Listing file	Optional file output by the assembler
.BNM .H .LDR	Loader format files	The loader's output in ASCII format. Different varieties exist. Used to create boot PROMS.
.IDM	Loader output file	For ADSP-218x DSPs, an IDMA boot-loadable image
.BNM .BNU .BNL	PROM splitter output files	For ADSP-218x DSPs, splitter binary output (middle, upper, and lower)
.H_# .S_# .STK	PROM format files	The loader's output in ASCII format. Different varieties exist. Used to create boot PROMS.
.ACH	Architecture file	(Previous releases only, replaced by .LDF)
.TXT	Linker Command-Line file	(Previous releases only, replaced by .LDF) ASCII text file that contains command-line input for the linker
.EXE	Debugging file	(Used in previous releases, replaced by .DXE)
.EXE	Compiled simulation file	Enables faster execution speed compared to a standard .DXE program

Table A-1. Files Used with VisualDSP++ (Cont'd)

Extension	Name	Purpose
.VDK	VisualDSP++ Kernel Support file	Enables VDK support
.JS .VBS	Script files	Enable you to script work for test applications. Scripting languages let you access the Automation API to interact with the IDDE.
.DSP	Assembly source file	Source file comprising assembly language instructions
.MAK .MK	Makefiles	The output make rule file is used for project builds
.DPG	Project group	An .XML file containing information about projects
.IDL	VCSE input	VCSE Interface Definition Language (VIDL) specification
.XML	Manifest	Used by New Component Package Wizard when packaging a component for distribution

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. Keyboard Shortcuts for 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 (→)

Table A-5. Keyboard Shortcuts for 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 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**

Obtaining Online Help

To obtain online Help, use the keyboard shortcuts listed in Table A-10.

Table A-10. 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-11.

Table A-11. 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]
```

Note: Specify the full path to `idde.exe`.

Table A-12 describes the `idde.exe` command-line parameters.

Table A-12. idde.exe Command-Line Parameters

Item	Description
`-f script_name`	Loads and executes the Tcl script specified by `script_name`. 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.
`-s session_name`	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 `idde.exe` allows you to run a different session. This overrides VisualDSP++'s default behavior of always connecting to the last session.
`-p project_name`	Specifies the project to load at startup. The project must already exist.

Examples:

```
idde.exe -f "c:\\scripts\\myscript.tcl"

idde.exe -s "My BF535 JTAG Emulator Session"

idde.exe -p "c:\\projects\\myproject.dpj"
```

Extensive Scripting

You can issue script commands from a command line, from the **Output** window's **Console** view, from a menu, from an editor window, or from a user tool.

- From a command line

  Load a script from a command window with an `idde` command by typing:

  ```
  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.

- From the **Output** window

  Load a script from the **Output** window's **Console** view by typing one of the following commands.

  For the Microsoft ActiveX script engine, type:

  ```
  Idde.LoadScript (filename)
  ```

  For Tcl, type:

  ```
  source filename
  ```

  As in C/C++, use a backslash (`\`) as an escape character. If you specify paths in the Windows environment, you must escape the escape character, as shown in this example:

  ```
  c:\\my_dir\\my_subdir\\my_file.tcl
  ```

For Tcl only, you can also use forward slashes to delimit directories in a path, as shown in this 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 procedure) for the script interpreter to evaluate at once.

Use the built-in `Idde` object to easily access the properties and methods of the VisualDSP++ Automation API when using a Microsoft ActiveX script engine. For example:

```
Idde.ActiveSession.ActiveProcessor
```

Evaluate expressions by using the "`?`" when a Microsoft ActiveX script engine is selected. For example:

```
? Idde.FullName
```

- From a menu

  You can quickly issue frequently used scripts. From the **File** menu, choose **Recent Scripts** and then select the script.

- From an editor window

  In an open editor window that contains a script, right-click and choose **Load Script**, as shown in Figure A-1.

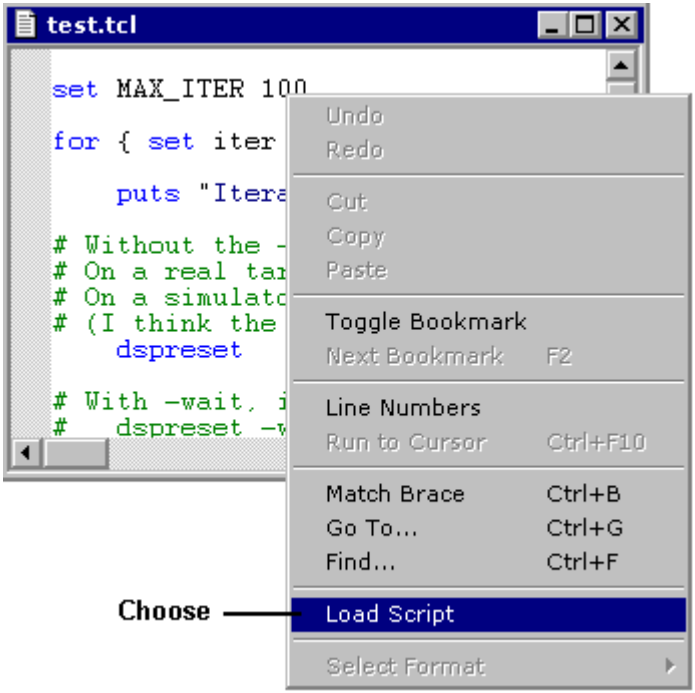

Figure A-1. Running a Script from an Editor Window

- From a user tool

  From a toolbar, click a user tool or choose a user tool from the **Tools** menu.

You can invoke a script (such as `.js` or `.vbs`) automatically when you launch VisualDSP++ from a shortcut set up on your Window's desktop or **Start** button. Right-click on the shortcut and select **Properties** and the **Shortcut** tab. Then append `-f` and the name of the script file to the executable file in the **Target** text box.

The example shown in Figure A-2 runs `myscript.js` automatically when `idde.exe` is launched.

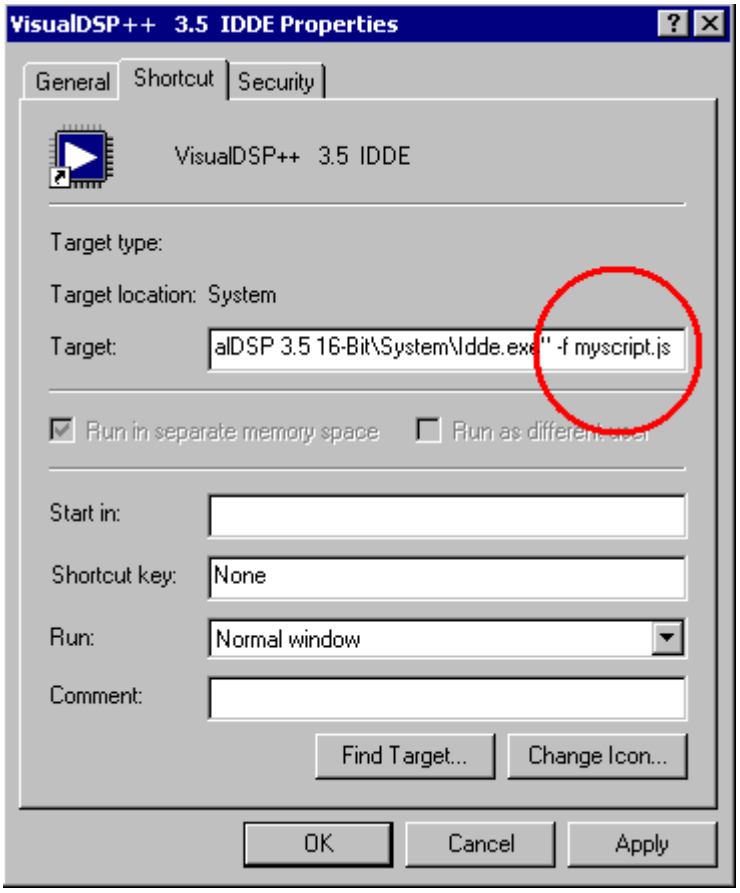

Figure A-2. Example of Loading a Script from a Shortcut

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-13. Toolbar Buttons

Button	Purpose
	Creates a new document
	Opens an existing document
	Saves the active document or template with the same name
	Saves all open files that have been modified, including files not in the current project
	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

Table A-13. Toolbar Buttons (Cont'd)

Button	Purpose
	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
	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

Table A-13. Toolbar Buttons (Cont'd)

Button	Purpose
	Provides context-sensitive Help for a button command or portion of VisualDSP++
	Opens the **About VisualDSP++** dialog box
	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

Table A-13. Toolbar Buttons (Cont'd)

Button	Purpose
	Arranges windows so they overlap
	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-13. 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 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-14. 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.
(*material*)	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-15 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-15. Match Characters

Character	Description
*	An asterisk matches the preceding any number of times, including none at all. Thus, `ap*le` matches `apple`, `aple`, `appppple` and `ale`. For example, `^ *void` matches only when `void` 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, `ap+le` matches `apple` and `aple`, but does not match `ale`.
?	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`. 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-16 represent tagged expressions from the **Find** field.

Table A-16. 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-17.

Table A-17. 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-18 describes the two types of comment strings that you can set for each file type.

Table A-18. Start and Stop 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
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

Online Help Features and Operations

This section describes online Help features and explains how to:

- Use the Help window
- Invoke online Help
- View context-sensitive Help
- Use the Help window's navigation buttons
- Copy example code from Help
- Print Help
- Book-mark frequently used help topics
- Navigate in online Help
- Use the search features
- View and print online manuals
- Use the About VisualDSP++ dialog box

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 A-3 shows the parts of the **VisualDSP++ Help** window.

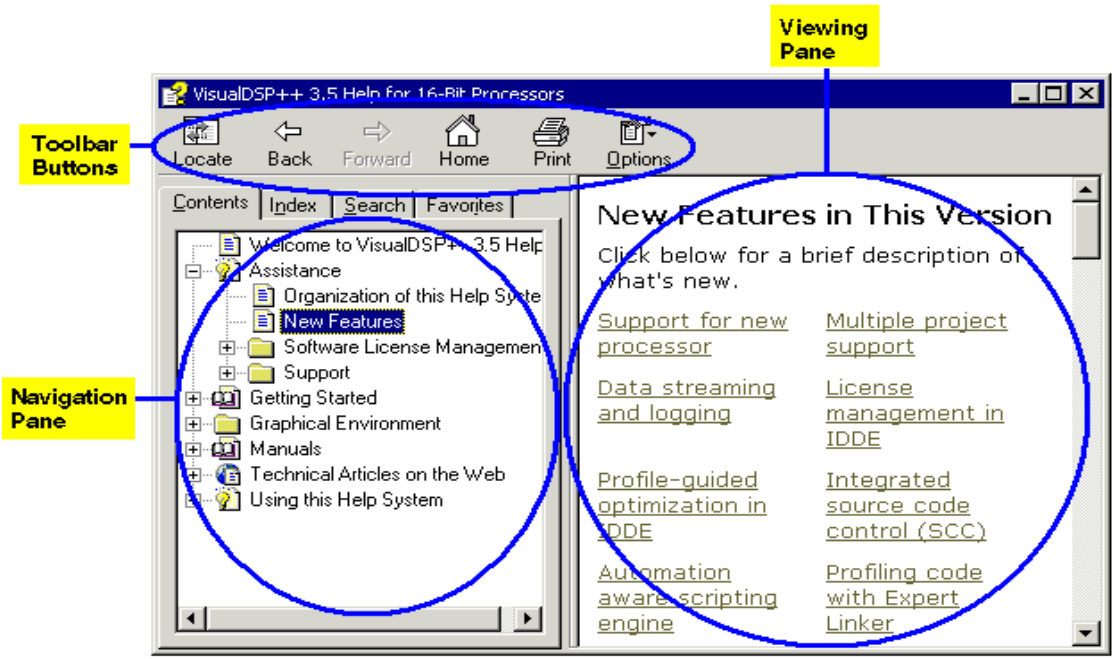

Figure A-3. Parts of the VisualDSP++ Help Window

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 the VisualDSP++ **Help** menu, choose **Contents**, **Search**, or **Index**.

To access online Help from the Windows **Start** button, click the **Start** button and choose **Programs**, **Analog Devices**, **VisualDSP 3.5 for 16-Bit Processors**, 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 installation's `Help` folder.

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. Record 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++.

VisualDSP++'s context-sensitive Help 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 filed) 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 A-4.

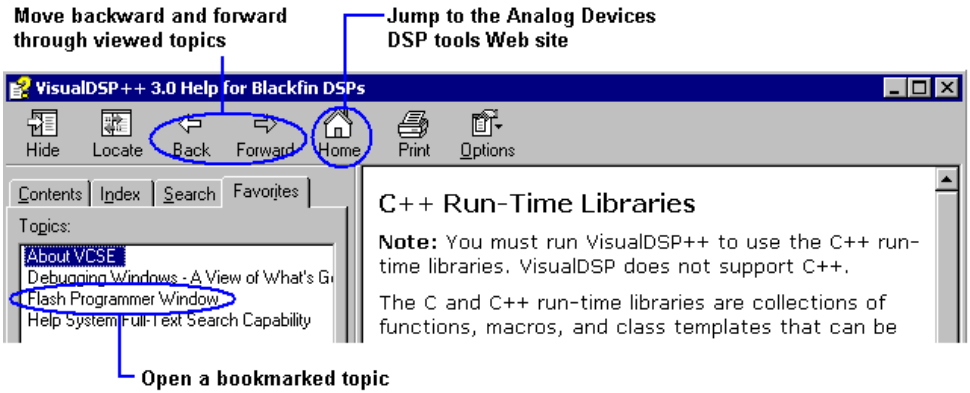

Figure A-4. Help Window Navigational Aids

Other standard Microsoft HTML Help buttons are described in Table A-19.

Table A-19. Standard Microsoft HTML Help Buttons

Button	Purpose
	Hides the Help window's left pane. This button narrows the Help window.
	Displays the Help window's left pane. This button restores a full view after you click **Hide**.
	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 runs.

Printing Help

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

Table A-20. 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**.

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

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

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 A-5)

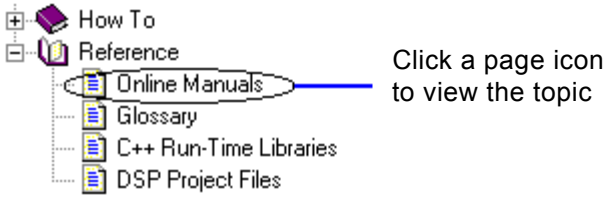

Figure A-5. Contents Page – Online Manuals Topic

- **An index entry on the Index page** (Figure A-6)

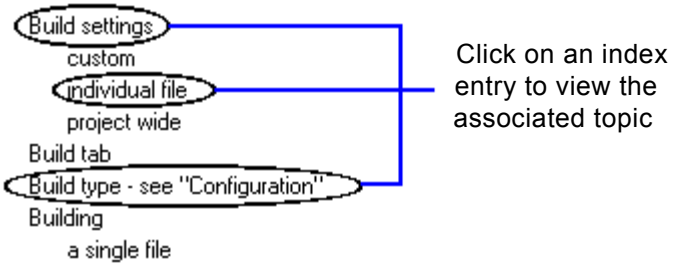

Figure A-6. Index Entries on the Index Page

- **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, semicolon, 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 wildcard expressions to further narrow the search criteria. Figure A-7 on page A-59 shows an example of a Boolean search for "`new AND plot`".

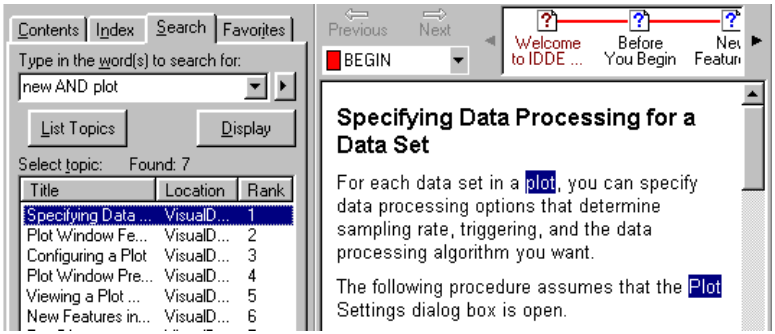

Figure A-7. Boolean Search for “new AND plot”

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 for more precise results.

- 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 A-21 describes the results of these different kinds of searches.

Table A-21. 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 A-22 describes the results of using Boolean operators to define a search.

Table A-22. 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 Manuals

VisualDSP++ includes three types of user documentation.

Table A-23. 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 `VisualDSP\Help` 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 `Docs` folder. Viewing and printing a `.PDF` file requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). Running `setup.exe` 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 `Docs\Reference` 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).

Using the About VisualDSP++ Dialog Box

Selecting the **About VisualDSP++** option from the **Help** menu opens the **About VisualDSP++** dialog box, which provides access to the following types of support information.

- Software versions

  The **General** page (Figure A-8) provides information about your version of VisualDSP++. This information includes the name of the registered user and company, the version of IDDE and its build date, and directory path in which VisualDSP++ is installed.

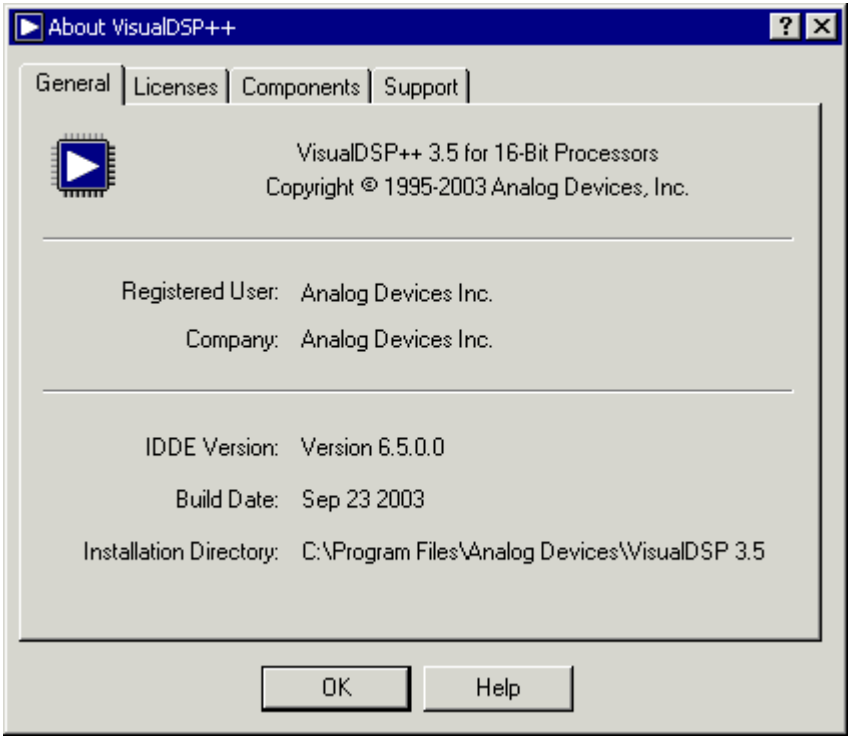

Figure A-8. General Page

- License management

  The **Licenses** page (Figure A-9) provides a centralized view of your current licenses. You can view license status and perform all necessary licensing activities (installing, registering, and validating).

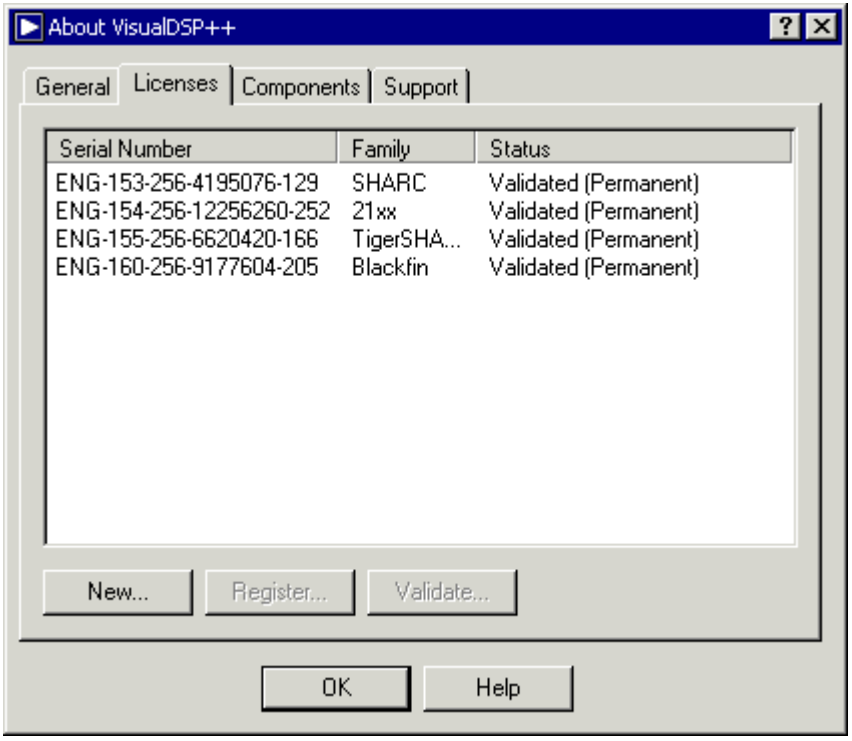

Figure A-9. Licenses Page

- Component versions

  The **Components** page (Figure A-10) displays a list of your system's components and provides information (name, version, provider) about your debug target, symbol manager, and processor library.

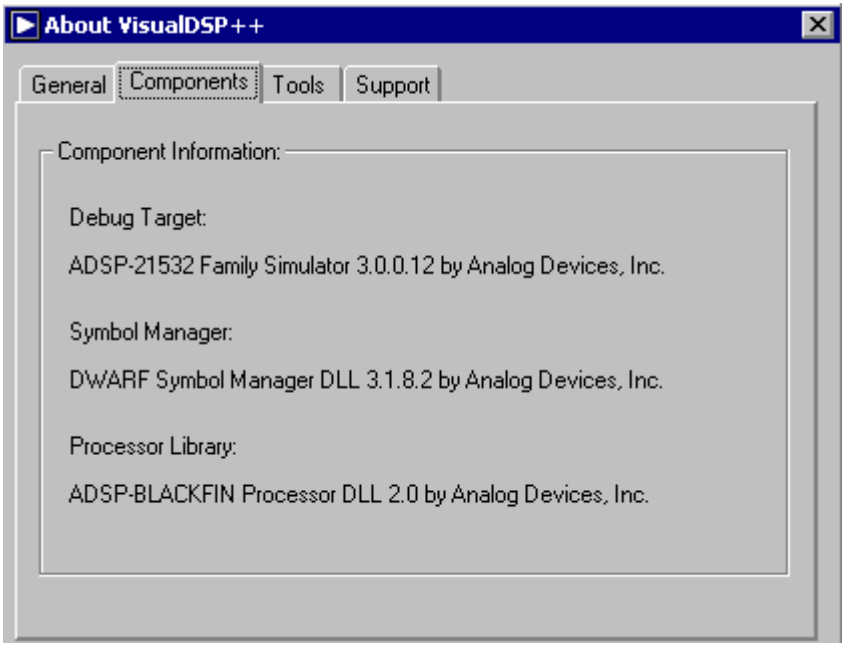

Figure A-10. Components Page

- Tools

  The **Tools** page (Figure A-11) displays a list of your system's tools. Each tool includes a description, version number, and the name of the company that developed it.

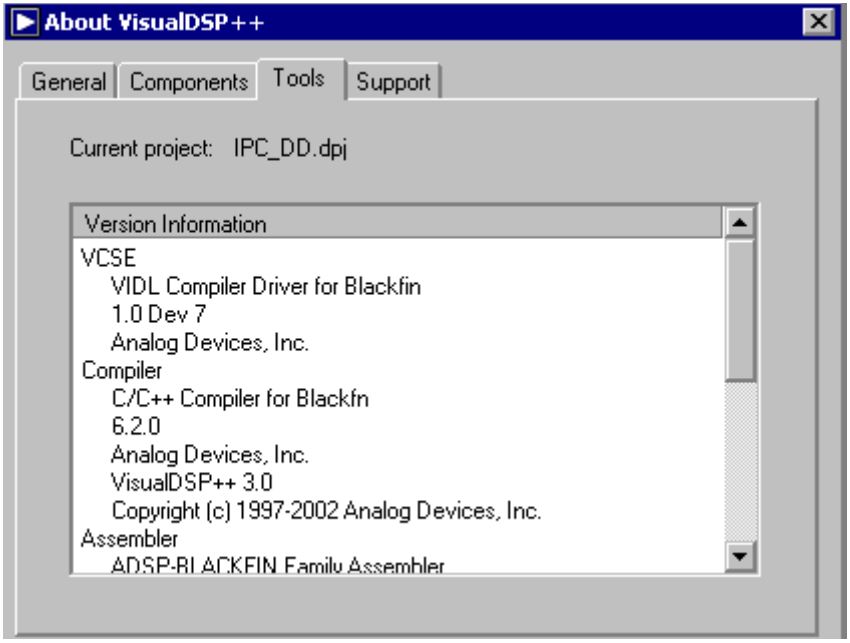

Figure A-11. Tools Page

- Links to support on the Web

  The **Support** page (Figure A-12) provides direct links to various Web pages that contain support information such as application notes, code examples, the DSP Knowledgebase, IC and tools anomalies and workarounds, manuals and datasheets, product comparisons, tools updates, and more. You can also generate the body of an email that automatically contains your system's description.

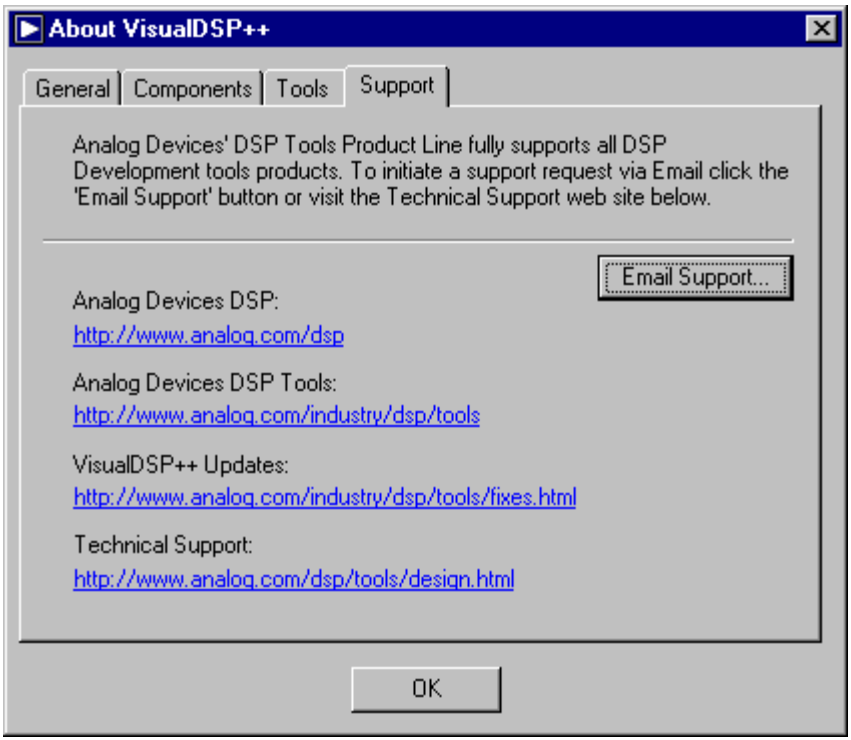

Figure A-12. Support Page

B SIMULATION OF BLACKFIN PROCESSORS

This appendix provides Blackfin simulator-specific information.

The information is organized as follows.

Peripheral Support in Simulators

Use the following key for the tables in this section.

Key	
✓	Implemented
NA	Not applicable
NP	Not planned for implementation
FR	Planned for a future release

Table B-1 summarizes peripheral support in the ADSP-BF535 simulator.

Table B-1. Peripheral Support in the ADSP-BF535 Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	✓	✓	NA
UART	✓	✓	NA
PCI	NP	NP	NP
USB	NP	NP	NP
Flags	✓	✓	NA
System Timers	✓	✓	NA
RTC	✓	NA	NA
EBIU	NP	NA	NA
SPI	✓	✓	NP
Watch Unit	NP	NA	NA
Trace Unit	NP	NA	NA
Core Timer	✓	✓	NA

Table B-1. Peripheral Support in the ADSP-BF535 Simulator (Cont'd)

Peripheral Support	Modeled	Streamable	Bootable
MEMDMA	✓	✓	NA
PROM	FR	NA	FR

Table B-2 summarizes peripheral support in the ADSP-BF535 compiled simulator.

Table B-2. Peripheral Support in the ADSP-BF535 Compiled Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	✓	✓	FR
UART	FR	FR	FR
PCI	NP	NP	NP
USB	NP	NP	NP
Flags	✓	✓	NA
System Timers	FR	FR	NA
RTC	FR	NA	NA
EBIU	NP		NA
SPI	FR	FR	NP
Watch Unit	FR	NA	NA
Trace Unit	FR	NA	NA
Core Timer		FR	NA
MEMDMA	✓	✓	NA
PROM	NP	NA	NP

Table B-3 summarizes peripheral support in the ADSP-BF533 simulator.

Table B-3. Peripheral Support in the ADSP-BF533 Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	✓	✓	FR
UART	FR	FR	FR
Flags	FR	FR	NA
System Timers	✓	✓	NA
RTC	FR	NA	NA
EBIU	FR	NA	NA
PPI	✓	✓	NP
SPI	FR	FR	NP
Watch Unit	✓	NA	NA
Trace Unit	✓	NA	NA
Core Timer	✓	FR	NA
Watch Dog Timer	FR	NA	NA
PROM	NP	NA	NP

Table B-4 summarizes peripheral support in the ADSP-BF533 compiled simulator.

Table B-4. Peripheral Support in the ADSP-BF533 Compiled Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	FR	FR	FR
UART	FR	FR	FR
Flags	NP	NP	NA
System Timers	FR	FR	NA
RTC	NP	NP	NA

Table B-4. Peripheral Support in the ADSP-BF533 Compiled Simulator

Peripheral Support	Modeled	Streamable	Bootable
EBIU	NP	NP	NA
PPI	FR	FR	NP
SPI	FR	FR	NP
Watch Unit	FR	NA	NA
Trace Unit	FR	NA	NA
Core Timer	✓	FR	NA
Watch Dog Timer	FR	NA	NA
PROM	NP	NA	FR

Table B-5 summarizes peripheral support in the ADSP-BF561 simulator.

Table B-5. Peripheral Support in the ADSP-BF561 Simulator

Peripheral Support	Modeled	Streamable	Bootable
SPORT	FR	FR	FR
UART	FR	FR	FR
Flags	FR	FR	NA
System Timers	✓	✓	NA
RTC	FR	NA	NA
EBIU	FR	NA	NA
PPI	✓	✓	NP
SPI	FR	FR	NP
Watch Unit	FR	NA	NA
Trace Unit	FR	NA	NA
Core Timer	✓	FR	NA

Table B-5. Peripheral Support in the ADSP-BF561 Simulator (Cont'd)

Peripheral Support	Modeled	Streamable	Bootable
Watch Dog Timer	FR	NA	NA
PROM	FR	NA	FR

Special Considerations for Peripherals

This section describes the limitations of the simulation software models.

Universal Asynchronous Receiver/Transmitter Peripheral

You can manipulate all the UART configuration bits. Currently, you cannot simulate the data error (Framing Error, Parity Error, Break Interrupt) conditions or the **Modem Status** register status bits (Data Carrier Detect, Ring Indicator, Data Set Ready, Clear To Send). You can specify **Set Break** in the Line Control register, but this setting has no effect. The current simulator does not model the `IRCR` register.

Timer (TMR) Peripheral

In Width Capture (`WDTH_CAP`) mode, the timer counts the number of clocks in both the width and period. The waveform that the timer reads is attached via the **Streams** dialog box in VisualDSP++.

You can attach a file to the following device names.

- `TIMER0_WDTH_CAP`
- `TIMER1_WDTH_CAP`
- `TIMER2_WDTH_CAP`

The format of the input file is as follows.

```
PERIOD_COUNT
WIDTH_COUNT
PERIOD_COUNT
WIDTH_COUNT
```

In WDTH_CAP mode, the timer reads two 32-bit values from the input file. The first value is the number of pulses (clocks) in the period. The second value is the number of pulses in the width.

When PULSE_HI is set, the timer delivers high widths and low periods. When PULSE_HI is not set, the timer delivers low widths and high periods.

Simulator Instruction Timing Analysis for ADSP-BF535 Processors

The ADSP-BF535 Family Simulator is a core cycle-accurate simulator with an eight-stage pipeline. The simulator models all the sequencer and memory events of the ADSP-BF535 processor.

The Pipeline Viewer enables you to understand the execution timing of your program. It shows the flow of instructions through the pipeline and any stalls due to sequencer or memory events. For information about configuring and using the Pipeline Viewer, see "Pipeline Viewer Window" on page 2-93 or the VisualDSP++ on-line Help.

The Pipeline Viewer for the ADSP-BF535 processor displays stages Decode through Writeback. The first two stages of the pipeline, IF1 and IF2, are not displayed because the information, provided by the simulator, in those stages is not significant.

Stall Reasons

The stall reasons are grouped in three categories:

- Multicycle instructions latencies (see "Multicycle Instructions and Latencies" on page B-20)
- Instructions latencies (see "Instruction Latencies" on page B-25)
- L1 data memory latencies (see "L1 Data Memory Stalls" on page B-32)

They are reported in the Pipeline Viewer as:

- Data address generator (DAG) read-after-write (RAW) hazard
- Data register (dreg) hazard: two cycle
- Dreg register (dreg) hazard: one cycle
- Memory stall
- Memory-mapped register (MMR) stall
- CSYNC stall
- SSYNC or IDLE SYNC stall
- Raise stall
- Single-step (SS) mode
- RET read after write
- Unidentified stall

Kill Reasons

The kill reasons are as follows.

- **Branch Kill** – change of flow
- **Mispredict** – mispredicted conditional change of flow
- **Refetch** – refetch, such as following an `IDLE` instruction
- **Interrupt** – interrupt/exception

Pipeline Viewer Window Examples

Figure B-1 shows a RAW hazard stall.

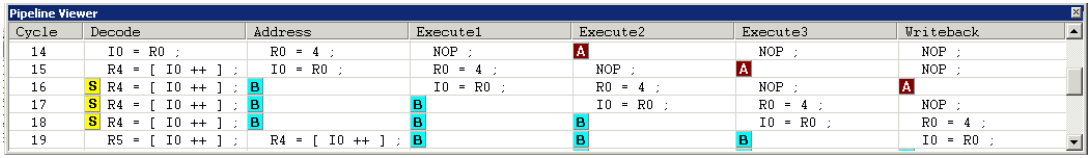

Figure B-1. RAW Hazard Stall

These stalls are detected in the Decode stage. The instruction stalls there until all DAG registers required and updated in later pipe stages are available.

In this example, the instruction "`I0 = R0;`" in the Execute1 (cycle 16), Execute2 (cycle 17) and Execute3 (cycle 18) stage is stalling the instruction "`R4 = [ I0++ ];`" in the Decode stage. This stall is caused by the first instruction because it updates the value of `I0` in stage Writeback, while the second instruction needs the value of `I0` in the Address stage to increment `I0`.

Figure B-2 shows a fetch stall.

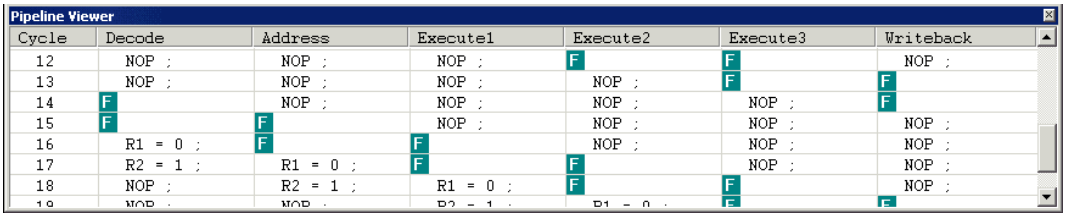

Figure B-2. Fetch Stall

Fetch stalls are detected in the Decode stage and are caused by memory latencies when an instruction is fetched.

In this example, two fetch stalls appear in the Decode stage (cycles 14 and 15) because of a memory latency when instruction "`R1 = 0;`" is fetched. These fetch latencies are then propagated in the pipeline: stage "Address" (cycles 15 and 16), stage "Execute 1" (cycles 16 and 17), and so on.

Pipeline Viewer Window Messages

When you hold the **Ctrl** key down and pause the mouse over a pipeline viewer event icon indicating instructions, the **Pipeline Viewer** window displays informational messages. An example is shown in Figure B-4 on page B-19.

These types of messages may appear:

- Stalls detected
- Kills detected
- Multicycle instruction messages

Pipeline Viewer Detail View Stall Event Messages

Table B-6 shows the messages that occur when a stall is detected.

Table B-6. Stalls Detected Messages (ADSP-BF535)

Message	Explanation	Example
ICache miss	Instruction cache miss	
IAU empty	Instruction alignment unit empty	
DCache miss	Data cache miss	
DCache store buffer full	Data cache buffer overflow. The processor stalls until the FIFO moves forward and a space is free.	
DCache load while store pending	A load access collides with a pending store access in the store buffer. (They are tying to access the same address.)	
DCache load while store pending w/ size mismatch	Load access size is different from that of the store access. The buffer must be flushed before the load can be carried out.	
DCache bank collision	The addresses in a dual- memory access command are accessing the same minibank. It does not matter whether both are loads, or load and store.	
SYNC with store pending	SYNC instructions force all speculative, transient in the core/system to be completed before proceeding.	`SSYNC;`
EU->MUL/MAC RAW hazard	Execution unit, Multiply or Multiply accumulate with a read after write hazard	`R0 = R1 + R0;` `P0 = R0;`
RETx RAW hazard	Writing to one of the RETx (RETS, RETI, RETX, RETN, or RETE) registers immediately followed by the corresponding return instructions.	`RETX = R0;` `RTX;`
Dagreg WAW hazard	Writing to one of the DAG registers, and immediately writing to it again.	`I3 = R3;` `I3 += M0;`
Dagreg RAW hazard	Writing to one of the DAG registers, and immediately reading	`I3 = R3;` `[I3] = R7;`

Table B-6. Stalls Detected Messages (ADSP-BF535) (Cont'd)

Message	Explanation	Example
dsp32alu implied ireg dependency RAW hazard		
ccMV preg->dreg RAW hazard	A conditional move of a preg into a dreg, followed by a read of the dreg	`If CC R0 = P1;` `R0 = R1;`
ccMV dreg->dreg RAW hazard	A conditional move of a dreg into a dreg, followed by a read of the source dreg	`If CC R0 = R1;` `R2 = R0;`
ccMV dpreg->preg RAW hazard	A conditional move of a dreg into a preg, followed by a read of the preg	`If CC P0 = R1;` `P1 = P0 ;`
loopsetup WAW hazard	A LSETUP instruction followed by another LSETUP, both writing to the same LC reg	`LSETUP (LS,LE)LC0=P0;` `LSETUP (LS,LE)LC0=P1;`
loopsetup while lc is nonzero	Using an LSETUP instruction and writing a value other than zero to the Lcreg	`LSETUP (LS,LE)LC0=P0;` `Nop;`
loop top/bot RAW hazard	Writing to a loop top/bottom register, followed by a read of the same register	`LT0 = R0;` `R2 = LT0;`
write to loop cnt stall	A write to a LCreg, followed by any op	`LC0 = R0;` `Nop; (`*`any op`*`)`
multicycle ALU2op instruction	A two-operand ALU instruction requiring more than one cycle to complete	`R0 *= R1;`
multicycle DAG instruction		`[--SP] = (R7:0,P5:0);`
CC2dreg RAW hazard	Reading the CC register into a dreg, and then reading that register	`R0 = CC;` `CC = R0;`
Mac/video after regmv sysreg to dreg raw hazard	Register move of a system register to a dreg, followed by a MAC or video instruction	`R0 = LC0;` `R2.H = R1.L * R0.H;`
Regmv sysreg to dreg followed by ALU op dreg raw hazard	Writing a system register to a dreg, followed by an ALU operation using that dreg as an operand	`R0 = LC0;` `R2 = R1 + R0;`

Table B-6. Stalls Detected Messages (ADSP-BF535) (Cont'd)

Message	Explanation	Example
Video after extracted 3-input add dreg raw hazard		
Extracted 3-input add followed by special dsp32 instruction		
Search followed by exu operation dreg raw hazard	A search instruction followed by any execution instruction with an operand of a dreg used in the search instruction	`(R3,R0) = search R1 (LE);` `R2.H =R1.L * R0.H;`
Regmv hazard: preg to dreg -> dreg to sys/preg RAW	A register move of a preg to a dreg, followed by another register move of that same dreg to a system register or preg	`R0 = P0;` `ASTAT = R0;`
Regmv hazard: sysreg to dreg -> dreg to dreg RAW	A register move of a system register to a dreg, followed by another register move of that same dreg to a dreg	`R0 = ASTAT;` `R1 = R0;`
Regmv hazard: sysreg to dreg -> dreg to sysreg RAW	A register move of a system register to a dreg, followed by another register move of that same dreg to a system register	`R0 = LC0;` `ASTAT = R0;`
Regmv hazard: sysreg to areg -> dreg to areg WAW	A register move of a system register to an accumulator register, followed by another register move of a dreg to the same accumulator register	`A0.w = LC0;` `A0 =R0;`
Regmv hazard: sysreg to areg -> preg to areg WAW	A register move of a system register to an accumulator register, followed by another register move of a preg to that same accumulator register	`A0.w = LC0;` `A0 =P0;`
Regmv hazard: sysreg to areg -> areg to areg WAW	A register move of a system register to an accumulator register, followed by another register move of an accumulator register to that same accumulator register	`A0.w = LC0;` `A0 =A1;`
Regmv hazard: sysreg to areg -> areg to dreg RAW	A register move of a system register to an accumulator register, followed by another register move of that same accumulator register to a dreg	`A0.w = LC0;` `R0 =A0;`

Table B-6. Stalls Detected Messages (ADSP-BF535) (Cont'd)

Message	Explanation	Example
Regmv hazard: sysreg to areg -> areg to sysreg RAW	A register move of a system register to an accumulator register, followed by another register move of that same accumulator register to a system register	`A0.w = LC0;` `ASTAT = A0.w;`
Regmv hazard: sysreg to areg -> load to areg WAW	A register move of a system register to an accumulator register, followed by a load to the same accumulator register	`A0.w = LC0;` `A0.w = [I0];`
Regmv hazard: sysreg to areg -> exu op using areg RAW	A register move of a system register to an accumulator register, followed by any execution unit operation using that accumulator register as an operand	`A0.w = LC0;` `A0 = A0(S);`
AQreg hazard: move to AQ -> exu op using AQ RAW		
CCreg hazard: move to CC -> exu op using CC RAW		

Kills Detected Messages

Table B-7 shows the messages that occur when a kill is detected.

Table B-7. Kills Detected Messages (ADSP-BF535)

Message	Explanation	Example
change-of-flow kill	A branch	`CALL (P0);`
rti change-of-flow kill	Return from interrupt kills	`RTI;`
mispredicted change-of-flow kill	Kills due to mispredicted branches	`R0 = 0; CC = R0;` `If CC JUMP next (bp);`
hardware loop bottom kill		
interrupt kill	Instructions in the pipeline are killed due to an interrupt	`RAISE 1`

Table B-7. Kills Detected Messages (ADSP-BF535) (Cont'd)

Message	Explanation	Example
sync kill	SYNC instructions force all speculative, transient in the core/system to be completed before proceeding, killing instructions in the pipe	`SSYNC;`

Multicycle Instructions

Multicycle instructions are a category of instructions that cannot be completed in less than two cycles. Consequently, the extra cycles generated by such an instruction cannot be removed without removing the multicycle instruction itself.

In Figure B-3, multicycle instruction "`[--SP] = (R7:6, P5:3)`" enters the pipeline Decode stage at cycle 16 and takes five cycles to complete (1 cycle per register to push on the stack SP). The next instruction "`R7 = 0`" takes only one cycle.

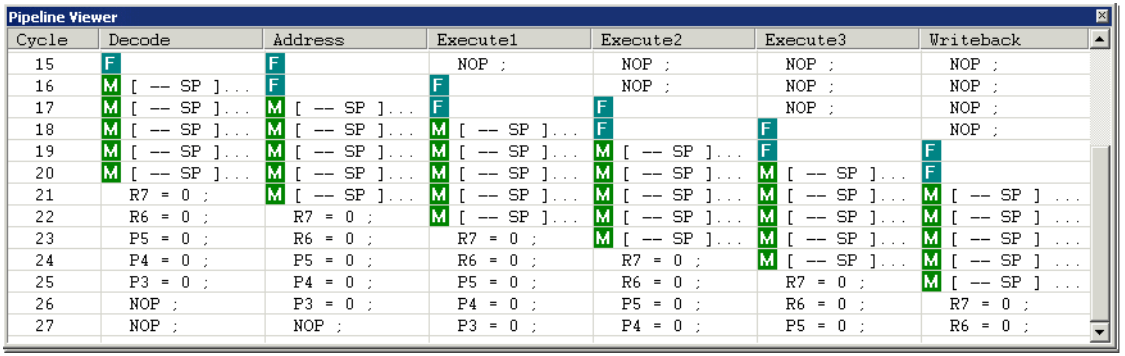

Figure B-3. Example of a Multicycle Instruction in the Pipeline Viewer

For details about multicycle instructions, see "Multicycle Instructions and Latencies" on page B-20.

Abbreviations in Pipeline Viewer Messages

Table B-8 shows abbreviations that may appear in the **Pipeline Viewer** window.

Table B-8. Abbreviations in the Pipeline Viewer Window

Abbreviation	Meaning
ALU	Arithmetic Logic Unit operations (Logical ops, Bit ops, Shift/Rotate ops, Arithmetic ops excluding Mult, Vector ops excluding Mult/MAC)
ALU2op	A two-operand ALU instruction
AQreg	
CC2dreg	CC register move to a dreg
ccMV	Conditional move
CCreg	CC register. This multipurpose flag typically holds the result of an arithmetic comparison.
DAG	Data Address Generator unit
Dagreg	A DAG register (for example, P5-0, I3-0, M3-0, B3-0, and L3-0)
dreg	Data register (for example, R7-0 or A1-0)
Dsp32alu	A 32-bit DSP ALU instruction
EXU	Execution unit
IAU	Instruction Alignment Unit
MAC	Multiplier/Accumulator Unit
MUL	Multiplier Unit operations (for example, Vector Multiply, 32-bit Multiply, Vector MAC)
preg	Pointer register (for example, P5-0, FP, USP, or SSP)
RAW	Read after write
regmv	A register move
sysreg	System Register (for example, LC1/0, LB1/0, LT1/0, SYSCFG, SEQSTAT, ASTAT, RETS, RETI, RETX, RETN, RETE, CYCLES, and CYCLE2)

Table B-8. Abbreviations in the Pipeline Viewer Window (Cont'd)

Abbreviation	Meaning
WAW	Write after write
Video	Video operations (video pixel operations)

Simulator Instruction Timing Analysis for ADSP-BF531, ADSP-BF532, ADSP-BF533, and ADSP-BF561 Processors

The simulator for the ADSP-BF531, ADSP-BF532, ADSP-BF533, and ADSP-BF561 processors is a core cycle-accurate simulator with a ten-stage pipeline. The simulator models all sequencer and memory events.

The Pipeline Viewer enables you to understand the execution timing of your program. It shows the flow of instructions through the pipeline and any stalls due to sequencer or memory events. For information about configuring and using the Pipeline Viewer, see "Pipeline Viewer Window" on page 2-93 or the VisualDSP++ on-line Help.

Stall Reasons

The stall reasons are as follows.

- Data address generator (DAG) read-after-write (RAW) hazard
- Memory stall
- Memory-mapped register (MMR) stall
- Unidentified stall
- Data register (dreg) hazard: two cycle

- Dreg hazard: one cycle
- CSYNC stall
- SSYNC or IDLE SYNC stall
- LSETUPO and not LPO_ALLOWED
- Awkward loop
- Raise stall
- SS mode
- RET read after write

Kill Reasons

The kill reasons are as follows.

- **Branch Kill** – change of flow
- **Mispredict** – mispredict conditional change of flow
- **Interrupt** – interrupt/exception
- **Refetch** – refetch, such as following an `IDLE` instruction

Pipeline Viewer Window Examples

Figure B-4 shows a RAW hazard stall.

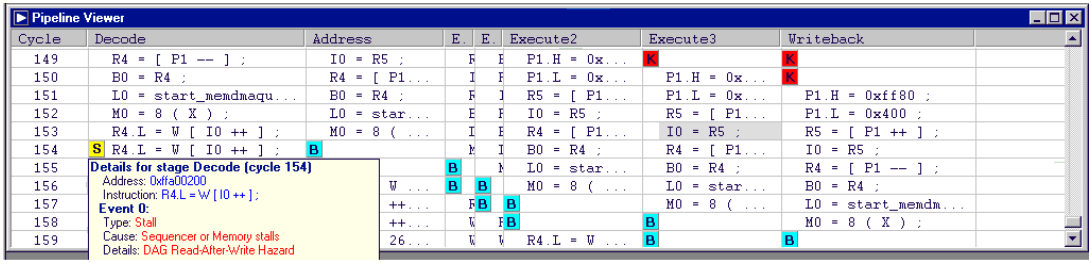

Figure B-4. RAW Hazard Stall

These stalls are detected in the Decode stage. The instruction stalls until all DAG registers required and updated in later pipe stages are available.

In the example, `I0=R5` in the Execute3 stage is stalling the instruction in decode, which wants to increment I0.

Figure B-5 shows an MMR stall.

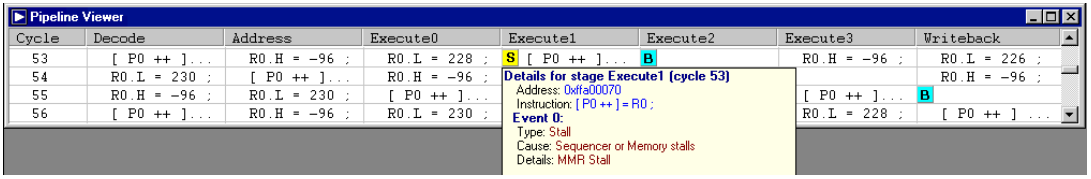

Figure B-5. MMR Stall

MMR stalls occur in E1 while the MMR value is being returned.

Figure B-6 shows a branch kill.

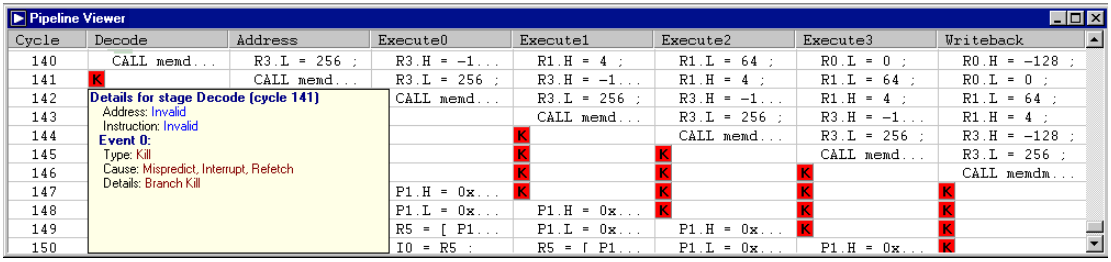

Figure B-6. Branch Kill

In this example, an unconditional control transfer kills several stages that were behind it. Fetching begins at the destination of the control transfer instruction after the killed stages.

Multicycle Instructions and Latencies

This section contains a description of all Blackfin processor multicycle instructions and latencies.

Multicycle behavior exists when an instruction, sometimes only under certain circumstances, is completed in more than one cycle. This cycle loss cannot be avoided without removing the instruction that caused it.

A *latency condition* exists when a pair of instructions incur extra cycles between them because of their proximity to each other in the code. You can avoid a latency condition's cycle loss by separating the two instructions by as many instructions as cycles lost. Each multicycle and latency entry indicates whether it is currently supported in the simulation environment.

All multicycle and latency conditions described here are native to the first implementation of Blackfin processor architecture. Future implementations may be different. The tables in this section show the cycle latencies of the 10x core processors, represented by the ADSP-BF532 and the ADSP-BF535 processor.

Multicycle Instructions

All instructions not mentioned here are completed in one cycle. This section describes instructions that take more than one cycle. Instruction names are consistent with the *Blackfin Processor Instruction Set Reference*. The cycle counts in the following examples represent the entire cycle time of the instruction shown.

Push Multiple or Pop Multiple

PushPopMultiple is completed in *n* cycles, where *n* is the number of registers pushed or popped.

Table B-9. PushPopMultiple Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
[--SP] = (R7:0,P5:0);	14 cycles	14 cycles
(R7:0,P5:3) = [SP++];	11 cycles	11 cycles

32-Bit Multiply (modulo 2^{32})

Table B-10. Bit Multiply Instruction and Cycles

Instruction	ADSP-BF532	ADSP-BF535
R0 *= R1;	3 cycles	5 cycles

Call and Jump

Table B-11. Call and Jump Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
CALL 0x22;	5 cycles	4 cycles
CALL (PC + P0);	5 cycles	4 cycles
CALL (P0);	5 cycles	4 cycles
JUMP 0x22;	5 cycles	4 cycles
JUMP (PC + P0);	5 cycles	4 cycles
JUMP (P0);	5 cycles	4 cycles

Conditional Branch

The number of cycles that a branch takes depends on the prediction as well as the actual outcome.

Table B-12. Conditional Branch Cycles

Prediction	taken				not taken			
Outcome	taken		not taken		taken		not taken	
Cycle Time	BF532	BF535	BF532	BF535	BF532	BF535	BF532	BF535
	4 cycles	4 cycles	8 cycles	7 cycles	8 cycles	7 cycles	1 cycle	1 cycle

Return

Table B-13. Return Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
RTX;	5 cycles	7 cycles[1]
RTE;	5 cycles	7 cycles[1]
RTN;	5 cycles	7 cycles[1]
RTI;	5 cycles	7 cycles
RTS;	5 cycles	4 cycles

1 Best case

Core and System Synchronization

Table B-14. Core and System Synchronization Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
CSYNC;	10 cycles	7 cycles
SSYNC;	10 cycles	7 cycles

Linkage

Table B-15. Linkage Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
LINK 8;	3 cycles	4 cycles
UNLINK;	2 cycles	3 cycles

Interrupts and Emulation

Table B-16. Interrupts and Emulation Instructions and Cycles

Instruction	ADSP-BF532	ADSP-BF535
RAISE 10;	3 cycles	3 cycles
EXCPT 3;	3 cycles	7 cycles
EMUEXCPT;	3 cycles	3 cycles[1]
STI R4;	3 cycles	3 cycles[1]

1 Best case as determined by physical characteristics of external memory

Testset

The TESTSET instruction is a multicycle instruction that is executed in a variable number of cycles. It is dependent on the cycles needed for a read acknowledge from off-core L2 memory and whether the address being tested is both in the cache and dirty. The number of cycles is determined as follows.

cycles = 1 (instruction) + 1 (stall) + *x* (read ack) + *y* (cache penalty)

In an optimum environment, *x* would be 5 and *y* would be zero. If the address resides in a dirty line, *y* is determined by the cycles to fill the dirty line plus any core boundary latencies. The address should not reside dirty in the cache as the address contents are meant to be updated across multiple processors and not be a local variable. This instruction is dependent on off-core conditions, so it is not modeled by the simulation environment.

Table B-17. TESTSET Instruction

Instruction	ADSP-BF535
TESTSET (P0);	7+ cycles[1]

1 Best case as determined by physical characteristics of external memory

Instruction Latencies

In addition to being based on instruction, instruction latencies are contingent on placement of specific instruction pairs relative to one another. You can avoid them by separating them by as many instructions as the number of cycles incurred between them. For example, if a pair of instructions incur a 2-cycle latency, separating them by two instructions will eliminate that latency.

In the tables that follow note that bold type identifies register dependencies within the instruction pairs. No bold type in an entry means that the latency condition will occur regardless of what registers are used. For a list of accumulator to data register (Areg2Dreg), math, video, multiply, and ALU ops, as well as register groupings, see "Instruction Groups" on page B-39 and "Register Groups" on page B-40. Instruction names are consistent with the *Blackfin Processor Instruction Set Reference*.

Calculate the total cycle time of each entry by adding the cycles taken by the instruction to the number of stall cycles for the instruction.

Accumulator to Data Register Latencies

Table B-18. Accumulator to Data Register Latencies

Description	Example <cycles + stalls > instruction	
	BF532	BF535
dreg = Areg2Dreg op video op using **dreg** as src	1 1 + 1	< 1 > **R1** = R6.L * R4.H (IS); < 1 + 2 > R5 = BYTEOP1P (R3:2, **R1**:0);
dreg = Areg2Dreg op rnd12/rnd20 using **dreg** as src	1 1	< 1 > **R4**.L = (A0 = R3.H*R1.H); < 1 + 1 > R0.H = R2 + **R4** (RND12);
dreg = Areg2Dreg op shift/rotate op using **dreg** as src	1 1	< 1 > **R4**.L = (A0 = R3.H*R1.H); < 1 + 1 > R1 = ROT R2 BY **R4**.L;

Table B-18. Accumulator to Data Register Latencies (Cont'd)

Description	Example <cycles + stalls > instruction	
	BF532	BF535
dreg = Areg2Dreg op add on sign using **dreg** as src	1 1	< 1 > **R0**.H=**R0**.L=SIGN(R2.H)*R3.H+SIGN(R2.L)*R3.L; < 1 + 1 > R6.H=R6.L= SIGN(**R0**.H)*R1.H+SIGN(**R0**.L)*R1.L;
dreg = math op Areg2Dreg op using **dreg** as src	1 1	< 1 > **R2** = R3 + R1; < 1 + 1 > R4.H = **R2**.L * R0.H;

Register Move Latencies

In each of the following cases the stall condition occurs when the same register is used in both instructions.

Table B-19. Register Move Latencies

Description	Example <cycles + stalls > instruction	
	ADSP-BF532	ADSP-BF535
dreg = sysreg ALU op using **dreg** as src (or vector ALU op)	1 1 1 1	< 1 > **R0** = LC0; < 1 + 1 > R2 = R1 + **R0**; < 1 > **R2** = LC0; < 1 + 1 > R1.L = **R2** (RND);
dreg = preg sysreg = **dreg**	1 1	< 1 > **R0** = P0; < 1 + 1 > ASTAT = **R0**;
dreg = sysreg dreg = **dreg**	1 1	< 1 > **R0** = ASTAT; < 1 + 1 > R1 = **R0**;
dreg = sysreg multiply/video op with **dreg** as src	1 1 + 1	< 1 > **R0** = LC0; < 1 + 2 > R2.H = R1.L * **R0**.H;
dreg = sysreg accreg = **dreg**	1 1	< 1 > **R0** = LC0; < 1 + 1 > A0 = **R0**;

Table B-19. Register Move Latencies (Cont'd)

Description	Example <cycles + stalls > instruction	
	ADSP-BF532	ADSP-BF535
preg = dreg any processor op using **preg**	1 1 + 4	< 1 > **P0** = R3; < 1 + 3 > R0 = **P0**;
dagreg = dreg any processor op using **dagreg**	1 1 + 4	< 1 > **I3** = R3; < 1 + 3 > R0 = **I3**;
dreg = sysreg sysreg = **dreg**	1 1	< 1 > **R0** = LC0; < 1 + 1 > ASTAT = **R0**;
accreg = sysreg **accreg** = dreg	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > **A0** = R0;
accreg = sysreg **accreg** = preg	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > **A0.w** = P0;
accreg = sysreg accreg = **accreg**	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > A1 = **A0**;
accreg = sysreg dreg = **accreg**	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > R0.L = **A0.x**;
accreg = sysreg sysreg = **accreg**	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > ASTAT = **A0.w**;
accreg = sysreg math op using **accreg** as src	1 1	< 1 > **A1.x** = LC0; < 1 + 1 > R1.H = (A0+=**A1**);
accreg = sysreg POP to **accreg**	1 1	< 1 > **A0.w** = LC0; < 1 + 1 > **A0.w** = [SP ++];
POP to **dagreg** any processor op using **dagreg**	1 1 + 3	< 1 > **I3** = [SP++]; < 1 + 3 > R0 = **I3**;
LOAD/POP to **preg** any processor op using **preg**	1 1 + 3	< 1 > **P3** = [SP++]; < 1 + 3 > R0 = **P3**;
R0.L = R1.L+R2.L R3 = R0.H*R4.L	The 10x core considers register halves to be independent, so this condition is not a register hazard.	

Move Conditional and Move CC Latencies

In each of the following cases the stall condition occurs when the same register is used in both instructions.

Table B-20. Move Conditional and Move CC Latencies

Description	Example(s) <cycles + stalls > instruction	
	ADSP-BF532	ADSP-BF535
dreg = CC if CC dreg = **dreg**	1 1	< 1 > **R0** = CC; < 1 + 1 > if CC R1 = **R0**;
if CC **dreg** = dreg multiply/video op using **dreg** as src	1 1 + 1 1 1 + 1	< 1 > if CC **R0** = R1; < 1 + 1 > R2.H = R1.L * **R0**.H; < 1 > if CC **R1** = R3; < 1 + 1 > SAA (R3:2, **R1**:0);
if CC **dreg** = preg math op using **dreg** as src	1 1 1 1	< 1 > if CC **R0** = P0; < 1 + 1 > R2 = R1 + **R0**; < 1 > if CC **R3** = P1; < 1 + 1 > SAA (**R3**:2, R1:0);
dreg = CC math op using **dreg** as src	1 1 1 1	< 1 > **R0** = CC; < 1 + 2 > R2.H = R1.L * **R0**.H; < 1 > **R1** = CC; < 1 + 2 > SAA (R3:2, **R1**:0);
dreg = CC CC = **dreg**	1 1	< 1 > **R0** = CC; < 1 + 2 > CC = **R0**;
if CC **preg** = dpreg any op using **preg**	1 1 + 4	< 1 > if CC **P0** = R1; < 1 + 3 > R4 = **P0**;
if CC **dreg** = dpreg CC = **dreg**	1 1	< 1 > if CC **R0** = R1; < 1 + 1 > CC = **R0**;

Loop Setup Latencies

Table B-21. Loop Setup Latencies

Description	Example <cycles + stalls > instruction	
	BF532	BF535
loop setup loop setup with same LC	1 1 + 6	< 1 > LSETUP (top1, bottom1) LC0 = P0; < 1 + 1 > LSETUP (top2, bottom2) LC0 = P1;
modification of LT or LB loop setup with same loop registers	1 1 + 9	< 1 > LT0 = [SP++]; < 1 + 3 > LSETUP (top, bottom) LC0 = P0;
loop setup with LC0 and LC0 != 0 any processor op	1 1	< 1 > LSETUP (top, bottom) LC0 = P0; < 1 + 1 > NOP;
loop setup with LC1 and LC1 != 0 any processor op	1 1	< 1 > LSETUP (top, bottom) LC1 = P0; < 1 + 1 > NOP;
LC0/LC1 reg written to any processor op	1 1 + 9	< 1 > LC0 = R0; < 1 + 4 > NOP;
LT0/LB0 written to and LC0 != 0 any processor op	1 1 + 9	< 1 > LT0 = [SP++]; < 1 + 4 > NOP;
LT1/LB1 written to and LC1 != 0 any processor op	1 1 + 9	< 1 > LB1 = P0; < 1 + 4 > NOP;
kill while loop buffer is being written due to: *interrupt*, *exception*, *NMI*, *emulation* events	0	3-cycle stall

Instructions Within Hardware Loop Latencies

The following stall conditions occur when the listed instruction or condition within a hardware loop results in a 3-cycle stall at the next iteration of the loop.

- Move conditional or POP into any of the LC/LB/LT registers
- Loop set up through the use of the same loop count registers in the first 3 instructions of the loop
- Branch in the first 3 instructions of the loop (JUMP, CALL, conditional branch)
- Interrupt or exception in the first 4 instructions of the loop
- CSYNC or SSYNC
- The inner hardware loop's bottom is strictly within the outer hardware loop's first four instructions.
- If the inner hardware loop's bottom is equal to the outer hardware loop's bottom, a 3-cycle stall applies to each iteration of the inner loop in addition to the 3-cycle stall of the outer loop.
- RTS, RTN, RTE, RTX, RTI
- If the loop's top instruction is not executed in the first iteration of the loop, a one-time 3-cycle stall penalty is incurred at the beginning of the second iteration (for example, a jump into the hardware loop to any instruction but the first).
- None of the above applies to the 10x core. The 10x core stalls when the loop start does not directly follow the LSETUP. This condition causes a one-time 3-cycle stall while the loop buffer is filled at the beginning of the second loop iteration.
- LSETUP to the same loop count register in the shadow of a previous LSETUP is held in D code until the first LSETUP commits.

Instruction Alignment Unit Empty Latencies

If the instruction alignment unit (IAU) is empty of the next instruction, that next instruction incurs a 1-cycle stall while the IAU is being filled. The following conditions can result in an IAU empty stall.

- Instruction cache miss or SRAM fetch miss
- Change of flow to an instruction address aligned across a 64-bit boundary
- The second instruction after a hardware loop is aligned across a 64-bit boundary
- The sixth instruction within a hardware loop is aligned across a 64-bit boundary

Table B-22. Instruction Alignment Unit Empty Latencies

Description		Example(s) <cycles + stalls > instruction
	BF532	**BF535**
Move register or POP to I0 or I1 SAA,BYTEOP2P,BYTEOP3P	1 1 + 4	< 1 > I1 = [SP++]; < 1 + 5 > R0 = BYTEOP3P (R1:0, R1:0) (HI);
Move register or POP to I0 or I1 BYTEOP1P/16P/16M, BYTEUNPACK	1 1 + 4	< 1 > I0 = R0; < 1 + 5 > R3 = BYTEOP1P (R3:2, R1:0);
Write to return register (RT[S,N,E,X,I]) return op	1 1 + 4 1 1 + 4	< 1 > RETI = P0; < 1 + 4 > RTI; < 1 > RETS = P3; < 1 + 4 > RTS;
math op video op with RAW data dependency	1 1 + 1	< 1 > **R3** = R2 + R4; < 1 + 1 > SAA (**R3**:2, R1:0);
dreg = search math op using **dreg**	1 1 + 2	< 1 > (R3, **R0**) = SEARCH R1 (LE); < 1 + 2 > R2.H = R1.L * **R0**.H;

Table B-22. Instruction Alignment Unit Empty Latencies (Cont'd)

Description		Example(s) <cycles + stalls > instruction
	BF532	BF535
core and system MMR access		< 1 + 2 > R0 = [P0]; // P0 = MMR address
L0/B0 = dreg I0 modulo update In general, any length and base dagreg assignment to a dreg followed by the corresponding index dagreg modulo update	1 + 4 1 + 4 1 + 4 1 + 4 1 + 4 1 + 4 1 + 4 1 + 4	< 1 > **L0** = R0; < 1 + 3 > R1 = [**I0**++]; < 1 > **B1** = R2; < 1 + 3 > **I1** += 4; < 1 > **L2** = R3; < 1 + 3 > R4 = [**I2**++M2]; < 1 > **B3** = R5; < 1 + 3 > **I3** += M2;

L1 Data Memory Stalls

L1 data memory (DM) stalls are incurred through reading or writing from L1 data memory. Accesses can either be direct (to or from DM SRAM) or indirect (to or from DM cache). Some of these stalls are multicycle instruction conditions (they occur as a result of a specific instruction).

Some stalls are latency conditions (they occur only when the two offending instructions are too close). The specifics are described in each entry. The following memory configurations apply to the ADSP-BF535 processor. Note that the causal factors in offending instructions and the stall consequences appear in **bold** type.

Minibank Access Collision

This section describes the following stalls.

- SRAM access
- Cache access

SRAM Access (1-Cycle Stall)

This stall can occur only when an instruction accesses a bank configured as SRAM. The memory regions associated with SRAM banks are calculated when an offset is added to the value of `SRAM_BASE_ADDRESS MMR`. The start addresses for banks A and B are:

- Bank A: `(SRAM_BASE_ADDRESS << 22) + 0x000000`
- Bank B: `(SRAM_BASE_ADDRESS << 22) + 0x100000`

The minibanks are contiguous 4096 byte (4KB) chunks within the A and B address space. With two simultaneous accesses (via a multi-issue instruction) to the same minibank, a 1-cycle stall is incurred. For example:

```
(I0 is address 0x001348, I1 is address 0x001994)
R1 = R4.L * R5.H (IS), R2 = [I0++], [I1++] = R3;
<1 cycle stall> (due to a collision in the second minibank in
superbank A)
```

A collision occurs regardless of whether the accesses are both loads, or load and store. If the first access is a load (DAG0) and the second is a store (DAG1), the cycles incurred are seen by the store buffer (see 3.4.0). Since the `SRAM_BASE_ADDRESS` value must be 4MB aligned (thus each minibank starts at `0xXXXXX000`), it is easy to determine if two addresses are going to collide in a minibank. If `((addr1>>12)==(addr2>>12))`, a collision occurs.

Cache Access (1-Cycle Stall)

This stall can occur only when one or both banks are configured as cache.

Only One Bank is Configured as Cache

In this case, data memory accesses are always cached to the same superbank, so you have to determine only the cache minibank. You must first find out how much data bank memory is modeled in the implementation of the Blackfin processor that you are using.

The standard Blackfin processor architecture model is 16 KB, thus four 4-KB minibanks. In this case you have to look at only bits 13 and 12 of the address to see what minibank the data memory access is cached to.

Table B-23. Minibanks Selected for 16KB of Data Bank Memory

Addr[13:12]	Minibank Selected
00	minibank 1 (0x0000–0x1000)
01	minibank 2 (0x1000–0x2000)
10	minibank 3 (0x2000–0x3000)
11	minibank 4 (0x3000–0x4000)

Every time the available bank memory is doubled, another bit must be used. For example, if an implementation of Blackfin processor architecture has 32 KB of data bank memory (eight 4-KB memory banks), bits 14, 13, and 12 must be used.

Table B-24. Minibanks Selected for 32KB of Data Bank Memory

Addr[14:12]	Minibank Selected
000	minibank 1 (0x0000–0x1000)
001	minibank 2 (0x1000–0x2000)
010	minibank 3 (0x2000–0x3000)
011	minibank 4 (0x3000–0x4000)
100	minibank 5 (0x4000–0x5000)
101	minibank 6 (0x5000–0x6000)
110	minibank 7 (0x6000–0x7000)
111	minibank 8 (0x7000–0x8000)

For simplicity, this document assumes the standard 16-KB data memory model. If the addresses in a dual memory access (multi-issue) instruction is cached to the same minibank, a 1-cycle stall is incurred.

```
(I0 is address 0x002348, I1 is address 0x002994)
R1 = R4.L * R5.H (IS), R2 = [I0++], [I1++] = R3;
<1 cycle stall> (due to a collision in minibank 3)
```

A collision occurs regardless of whether the accesses are both loads, or load and store. If the first access is a load (DAG0) and the second is a store (DAG1), the cycles incurred are seen by the store buffer (see "Store Buffer Overflow" on page B-37). If `(Addr1[13:12]== Addr2[13:12])`, a collision occurs.

Both Banks Are Configured as Cache

If both banks are cacheable, you must determine which superbank the accesses are cached to (in addition to the minibank) to determine whether a stall exists. This information depends on the value of the DCBS bit of the `DMEM_CONTROL MMR`. If DCBS is `1`, address bit 23 is used as bank select. If DCBS is `0`, address bit 14 is used as bank select.

(Note that these values are used for the 16-KB implementation of Blackfin processor data memory). Refer to "Cache Access" for details about how to determine the minibank. The following table assumes that DCBS is `0`.

Table B-25. Superbank, Minibank Selected When DCBS is 0

Addr[14:12]	Superbank, Minibank Selected
000	superbank A, minibank 1 (0x0000–0x1000)
001	superbank A, minibank 2 (0x1000–0x2000)
010	superbank A, minibank 3 (0x2000–0x3000)
011	superbank A, minibank 4 (0x3000–0x4000)
100	superbank B, minibank 1 (0x0000–0x1000)

Table B-25. Superbank, Minibank Selected When DCBS is 0 (Cont'd)

Addr[14:12]	Superbank, Minibank Selected
101	superbank B, minibank 2 (0x1000–0x2000)
110	superbank B, minibank 3 (0x2000–0x3000)
111	superbank B, minibank 4 (0x3000–0x4000)

If the addresses in a dual memory access (multi-issue) instruction is cached to the same superbank and minibank, a 1-cycle stall is incurred.

```
(I0 is address 0x002348, I1 is address 0x002994)
R1 = R4.L * R5.H (IS), R2 = [I0++], [I1++] = R3;
<1 cycle stall> (due to a collision in minibank 3)
```

A collision occurs regardless of whether the accesses are both loads, or load and store. If the first access is a load (DAG0) and the second is a store (DAG1), the cycles incurred are seen by the store buffer (see "Store Buffer Overflow" on page B-37).

When DCBS is `0` and `(Addr1[14:12]== Addr2[14:12])`, a collision occurs. When DCBS is `1` and `(Addr1[23,13:12]== Addr2[23,13:12])`, a collision occurs.

MMR Access

A read from any MMR space (on-core and off-core) results in a 2-cycle stall because the Blackfin processor architecture must wait for acknowledgement from the peripherals mapped to the MMRs being accessed.

```
(I0 contains an address between 0xFFC00000 and 0xFFE00000)
R2 = [I0++]; (In Supervisor Mode)
<2 cycle stall>
```

System Minibank Access Collision

A system access occurs when some external device, such as another processor in a multiple core system, accesses Blackfin processor architecture L1 memory. Whenever the system accesses a minibank currently being accessed by the core, a **<1-cycle stall>** is incurred because system memory accesses have higher priority than core accesses.

Store Buffer Overflow

The store buffer is a 5-entry FIFO that manages Blackfin processor instruction stores to L1 and L2 memory. All instruction stores must go through the store buffer. Thus, if the buffer is full, the Blackfin processor stalls until the FIFO moves forward and a space is freed.

The earliest time that a store can leave the buffer is 4 instructions (not cycles necessarily) after it was entered. Consequently, under ideal circumstances a continuous series of stores will take up 4 out of the 5 slots in the store buffer. If only one of the stores is delayed by an extra cycle, no penalty is imposed as the store buffer has 5 slots. Many scenarios can cause the store buffer to become full. To account for them, you must keep track of the proximity of stores and how many cycles they each take.

If a multicycle store is required, you must be sure that it is not followed too closely by other stores as they may become backed up. Here is a list of multicycle stores:

- Stores to non-cacheable memory (for example, MMR space)
- Stores to external L2 memory (memory addressed beyond L1 SRAM)
- Minibank conflict where the store is from DAG1 (the second access in a load/store multi-issue instruction—see "Minibank Access Collision" on page B-32)

Store Buffer Load Collision

This section describes cases where a load access collides with a pending store access in the store buffer because they have the same address (refer to section "Store Buffer Overflow" on page B-37 for a description of the store buffer).

Load/Store Size Mismatch

If the load access's size (8 bit, 16 bit, 32 bit) is different from that of the store access, the store buffer must be flushed before the load can be carried out. The stall time depends on how many stores are currently in the buffer and how long they each take to complete.

```
W [P0] = R0;
<N cycle stall as the buffer is flushed>
R1 = B [P0];
```

Store Data Not Ready

The data portion of a store does not necessarily have to be ready when it is entered into the store buffer. Store data coming from the dagregs and pregs has no delay, but all other store data is delayed by 3 instructions. If a load access collides with a store whose data is not ready, the Blackfin processor stalls for 3 cycles.

```
W [P0] = R0;
<3 cycles>
R1 = W [P0];
```

```
[P0] = P3;
<0 cycles>
R1 = [P0];
```

Instruction Groups

All instruction group members conform to naming conventions used in the *Blackfin Processor Instruction Set Reference*. Instruction groups described are not necessarily mutually exclusive in that the same instruction can belong to multiple groups.

Table B-26. Math Ops Instruction Groups

Math Ops		
Video Ops	**Mult Ops**	**ALU Ops**
Video Pixel Ops	Vector Multiply	Logical Ops
	32-bit Multiply	Bit Ops
	Vector MAC	Shift/Rotate Ops
		Arithmetic Ops (except Mult)
		Vector Ops (except Mult/MAC)

Table B-27. Areg2Dreg Ops Instruction Groups

Areg2Dreg Ops		
MAC to half reg	MAC to data reg	Vector Multiply
RND12	RND20	Add on Sign
Modify – Increment, only this case: [dreg|dreg_hi|dreg_lo] = (A0 += A1);		

Register Groups

Table B-28. Allreg Register Groups

allreg			
dreg	**preg**	**sysreg**	**dagreg**
R0	P0	ASTAT	I0
R1	P1	RETS	I1
R2	P2	RETX	I2
R3	P3	RETI	I3
R4	P4	RETN	M0
R5	P5	RETE	M1
R6	FP	LC0	M2
R7	SP	LT0	M3
statbits	**accreg**	LB0	L0
ASTAT [0]: AZ	A0	LC1	L1
ASTAT [1]: AN	A0.x	LT1	L2
ASTAT [2]: AC	A0.w	LB1	L3
ASTAT [3]: AV0	A1	CYCLES	B0
ASTAT [4]: AV1	A1.x	CYCLES2	B1
ASTAT [5]: CC	A1.w	SEQSTAT	B2
ASTAT [6]: AQ		SYSCFG	B3

Compiled Simulation

A traditional simulator decodes and interprets one instruction at a time. Each executed instruction often requires repeated decoding. Compiled simulation removes the overhead of having to repeatedly decode each instruction.

Compiled simulation is a process whereby the .DXE file loaded into a traditional simulator is converted into an .EXE file that is executed directly on the system hosting VisualDSP++. The execution speed of a compiled simulation program is greater than that of a standard .DXE program.

Compiled simulation employs a simulation compiler that preprocesses instructions in the .DXE file and generates an intermediate C++ source program. This program is compiled and linked with a standard set of libraries to produce an .EXE file that effects the simulation of the original .DXE file. Within VisualDSP++, you interact with the .DXE file as in traditional simulation. You do not directly interact with the .EXE file.

In a compiled simulation session, loading and executing the .DXE file loads and executes the corresponding .EXE file. You can view the .DXE file in disassembled form, set breakpoints, run, step, display registers and memory, and so on. The compiled simulation debug target maps user requests to the appropriate operations in the .EXE file and returns the results to the IDDE. You can also invoke an .EXE file in stand-alone mode from the command line.

To prepare a program for compiled simulation, you can begin with either of the following.

- The source files from which the .DXE was built
- An existing .DXE file

Program Preparation Starting from Source Files

To prepare a program for compiled simulation by using source files, perform the following tasks from within the VisualDSP++ environment.

1. Specify a debug session for compiled simulation.
2. Create a compiled simulation project containing the source files.
3. Build the compiled simulation project to create the `.DXE` and `.EXE` files.
4. Perform one of these actions:
    - Execute the program within VisualDSP++ by loading the `.DXE` program. This action causes the `.EXE` program to be loaded. Run or step the program in the normal way.
    - Execute the program outside of VisualDSP++ by opening a command window and entering the name of the `.EXE` file. If the program uses streams, append `-streamfile=<filename>` to the command.

Specifying a Session for Compiled Simulation

To run the `.EXE` file under the control of VisualDSP++, you must configure the debug session for compiled simulation as follows.

1. From the VisualDSP++ **Session** menu, choose **New Session** to open the **New Session** dialog box.
2. In the **Debug target** box, select **Blackfin Family Compiled Simulator** from the drop-down list.
3. In the **Platform** box, select **Blackfin Family Compiled Simulator** from the drop-down list.

4. In the **Processor** box, select one of these processors: **ADSP-BF531**, **ADSP-BF532**, **ADSP-BF533**, or **ADSP-BF535**.

At present, only the Blackfin family processors are supported.

5. In the **Session name** box, enter a name for this session.
6. Click **OK**.

Specifying Project Options for Compiled Simulation

When built, a compiled simulation project compiles the sources to the `.DXE` file and then invokes the compiled simulation driver to construct the corresponding `.EXE` file.

To create a project for compiled simulation:

1. From the **Project** menu, choose **New** to open the **Save New Project As** dialog box.
2. Enter a **file name** and click **Save**.

   The **Project Options** dialog box appears.
3. Click the **Project** tab.
4. From the **Processor** drop-down list, select one of these processors: **ADSP-BF531**, **ADSP-BF532**, **ADSP-BF533**, or **ADSP-BF535**.

At present, only the Blackfin family processors are supported.

5. In the **Type** box, select **Compiled simulation file** from the drop-down list.

6. (*Optional*) Click the **Compiled Simulation** tab and from the **Compiler Optimization** drop-down list select the optimization level that the driver will request of the compiler that builds the `.EXE` file. Your options are:
    - **None** (default) – no optimizations requested. This selection typically results in the shortest build time.
    - **Medium –** requests optimization configured to produce a significant improvement in simulation execution speed. This selection results in longer build time.
    - **Maximum –** requests all optimizations available to achieve the fastest possible execution speed. This selection can produce build times that are significantly longer than those produced by the **Medium** setting.

Medium and **Maximum** differ in effect only if `-cmvs` is specified in the **Additional options** box.

7. (*Optional*) On the **Compiled Simulation** page, enter **–cmvs** in the **Additional options** box to select the Microsoft Visual C++ compiler (6.0 or 6.1) for compilation.

The `-cmvs` option does not apply unless the Microsoft Visual C++ compiler (6.0 or 6.1) is installed on your system.

8. Click **OK**.
9. Build the project to create the `.DXE` and `.EXE` files.

Program Preparation Starting from an Existing .DXE File

Invoke the compiled simulator driver at the command line to generate the `.EXE` file from the `.DXE` file. You can invoke the `.EXE` file in stand-alone mode from the command line or from within a VisualDSP++ compiled simulation session.

You can invoke the compiled simulation driver (`simcc.exe`) from the command line to produce an executable `.EXE` file from a `.DXE` file. You can then run the executable to simulate the `.DXE`.

The `simcc` command syntax is:

```
simcc -chip [switches and parameters] dxe-filename
```

Table B-29 describes the `simcc` command and command-line items (switches and parameters).

Table B-29. simcc Command and Command-Line Items

Item	Description
`simcc`	Runs the compiled simulation driver
`-chip`	Specifies the processor. Valid selections are `-BF531`, `-BF532`, `-BF533`, or `-BF535`.
`-o exe-filename`	Specifies the name of the output `.EXE` file. By default, the output file has the same name as `dxe-filename` with an `.EXE` extension.
`-O`	Optimizes code (at the medium level) in the generated `.EXE` file
`-Omax`	Optimizes code (at the maximum level) in the generated `.EXE` file
`dxe-filename`	Specifies the input `.DXE` file
`-help`	Displays available `simcc` options
`-cmvs`	Selects the Microsoft Visual C++ 6.0 or 6.1 compiler for compilation. This choice may result in improved compilation speed. Note that `-cmvs` applies only if Microsoft Visual C++ 6.0 or 6.1 is installed on your system.

Execution of an .EXE File from the Command Line

When the generated `.EXE` file is executed from the command line, you configure stream support by supplying a stream configuration file as an argument to the `.EXE` file. The stream configuration file is read at program startup, and the specified streams are created and opened.

The command syntax is:

```
<exe-filename> [-streamfile=<streamfile>]
```

Note that `<streamfile>` is the stream configuration file. The file format is described as follows.

- Each line in the file represents one stream.
- The input is case insensitive.
- The file is a text file with a `.TXT` extension.

The syntax of each line is:

```
filename device [address] direction [flags] [format]
```

The filename must be the first entry on each line, but the other entries may appear in any order. Double brackets ([]) denote optional entries.

Table B-30 describes the line parameters in the stream configuration file.

Table B-30. Line Parameters in the Stream Configuration File

Item	Description
`filename`	The name of the data file to be read or written. If the name contains embedded spaces, it must appear within quotes.
`device`	One of the following peripherals. • **memory** – denotes memory as the input or output device • **spt0** – denotes serial port 0 as the input or output device • **spt1** – denotes serial port 1 as the input or output device
`address`	The device's memory address, which is required for a memory stream. The address can be hexadecimal (prefix 0x or 0X), octal (prefix 0), or decimal.
`direction`	The stream direction, which is either of the following. • **input** • **output**
`Flags`	Stream characteristics, as defined by either of the following. • **circular** – causes the program to continue reading data from the start of the file when the end of file is reached • **nocircular** (default)
`Format`	The file's data format, which is one of the following. • **hexadecimal** (default) • **octal** • **binary** • **signed integer** • **unsigned integer** • **integer** • **signed fractional** • **unsigned fractional** • **fractional** **Note:** Unless preceded by **unsigned**, **integer** and **fractional** denote signed values.

C SIMULATION OF ADSP-21XX PROCESSORS

This appendix provides simulator-specific information for ADSP-219x processors. Boot simulation also applies to ADSP-218x processors.

Example programs for the ADSP-219x peripherals in this appendix are in the `Simulator Peripheral Examples` folder. To view the examples, open the `Analog Devices` folder and then open:

`VisualDSP 3.5 16-Bit\219x\Examples\Simulator Peripheral Examples`

The information is organized as follows.

Peripheral Support in Simulators

Use the following key for the tables in this section.

Key	
✓	Implemented
NA	Not applicable
NP	Not planned for implementation

Table C-1 summarizes peripheral support in the ADSP-2191 simulator.

Table C-1. Peripheral Support in the ADSP-2191 Simulator

Peripheral Support	Model	Streamable	Bootable
EMI	✓	NA	NP
GPIO	✓	✓	NP
Host Port	✓	✓	NP
SPI0	✓	✓	NP
SPI1	✓	✓	NP
SPT0	✓	✓	NP
SPT1	✓	✓	NP
SPT2	✓	✓	NP
UART	✓	✓	NP
Timer	✓	NA	NA
MEMDMA	✓	NA	NA
INTC	✓	NA	NA
Boot PROM	✓	NA	✓
Programmable Flags	NP	NP	NA

Table C-2 summarizes peripheral support in the ADSP-2192-12 simulator.

Table C-2. Peripheral Support in the ADSP-2192-12 Simulator

Peripheral Support	Model	Streamable	Bootable
Timer	✓	NP	NA
USB	✓	NP	NP
PCI	✓	NP	NP

Table C-3 summarizes peripheral support in the ADSP-218x simulator.

Table C-3. Peripheral Support in the ADSP-218x Simulator

Peripheral Support	Model	Streamable	Bootable
Timer	✓	NP	NP
SPORT	✓	✓	NP
IDMA	✓	✓	NP
BDMA	✓	✓	✓

General-Purpose I/O (GPIO) or Flag I/O (FIO) Peripheral

The GPIO peripheral provides flag I/O functionality. The 16 flag I/O bits are grouped as a 16-bit register. The behavior of the bits is controlled with ten, 16-bit registers as shown in Table C-4. Each pin/bit is independent of the others. The reset value of all GPIO registers is 0 (zero).

Table C-4. GPIO Registers

Register	Address	Description
Direction	0x1800	Controls whether bits are input or output (output = **1**) You can alter this value while the program is running.
Control and Status (two registers)	0x1802	Write **1** to clear. Clears the value of output bits.
	0x1803	Write **1** to set. Sets the value of output bits.
	Note: Input bits are read-only. Both registers can be read for current status.	
Interrupt Mask A (two registers)	0x1804	Write **1** to clear. Clears bits in Interrupt Mask A.
	0x1805	Write **1** to set. Sets bits in Interrupt Mask A.
	Note: Both registers can be read for the current Mask A value.	
Interrupt Mask B (two registers)	0x1806	Write **1** to clear. Clears bits in Interrupt Mask B.
	0x1807	Write **1** to set. Sets bits in Interrupt Mask B.
	Note: Both registers can be read for current Mask B value.	
Polarity	0x1808	Relation between GPIO bits and chip pins (reverse = **1**)
Interrupt Sensitivity	0x180A	Level or edge sensitive interrupts (edge = 1)
Interrupt Edge Sensitivity	0x180C	Double- or single-edge sensitive (double = 1)

Input and Output Handling

Use streams to simulate GPIO flags. The input values are read from stream `GPIO.RX`, and the output values are written to stream `GPIO.TX`. Only bits currently set as input bits are changed from the input stream, and only bits currently set as output bits are written to the output stream.

You can individually set the GPIO flags by using the signal interface, which overrides the streams for the bits in question. Any input bit that has no signal or stream input value is either set to `0` (zero) initially, or is returned to its previous value.

GPIO Window in VisualDSP++

You can view the GPIO register from the GPIO window.

To open the GPIO window:

1. From the **Registers** menu, choose **Peripherals**.
2. Choose **GPIO**.

When you set registers from the window, their values are updated immediately.

Host Port Interface (HPI) Peripheral

The ADSP-2191 Host Port Interface peripheral enables you to simulate host-port functionality. The peripheral supports DMA-controlled accesses and external host-initiated accesses.

The Host Port has ten registers to control its operation.

- Host Port Configuration register (0x1C01)
- Host Port Page register (0x1C02)
- Host Port Status register (0x1C03)

The DMA Control register conforms to the standard ADSP-2191 DMA descriptors.

- DMA Descriptor Pointer register (0x1D00)
- DMA Configuration register (0x1D01)
- DMA Page register (0x1D02)
- DMA Address register (0x1D03)
- DMA Count register (0x1D04)
- DMA Next Descriptor Pointer register (0x1D05)
- DMA Descriptor Ready register (0x1D06)

Input and Output

The Host Port has three associated streams to handle input, output, and external control.

- HOST.RX – Data input is from this stream. Data for DMA and external host-initiated writes to memory is read from the input stream.
- HOST.TX – Data output goes to this stream. Data from DMA and external host-initiated reads from memory is written to this output stream.
- HOST.CONTROL – External host-initiated operations are controlled from this control stream. Commands are read from this file and carried out until the end of the file is reached or the STOP command is found.

DMA operations have priority over external initiated operations, and, consequently, force the suspension of any external-initiated operations until the DMA operation is finished. Processing of the external initiated operations then resumes from the point at which they were suspended.

External-Initiated Control File Commands

The control file is a stream file, which is a list of numbers. You must construct this file if you have to simulate external host-initiated operations.

The C header file `hostcommands.h`, used in the VisualDSP++ build, is included to ensure that the exact definitions are available. They are documented in "Command Bit Definitions" on page C-8.

In the control file, any line that is blank or zero is skipped. On the same peripheral cycle, the control file is read until a command is reached. Arguments, however, are expected on the line immediately following their command. Therefore, they can be zero.

Processing of the control file continues during DMA operations until any of the following occur.

- External initiated direct operation – This operation is delayed until the DMA operation is finished.
- **Stop** command – This command stops processing.
- **Wait for DMA** command – This command suspends the control file until a DMA interrupt occurs.

Command Bit Definitions

The following bit definitions are used to construct the HPI commands. The word is read from the control file and decoded. If the word requires arguments, subsequent words are read to provide the arguments (such as counts for repeated operations).

- `#define HOST_DIRECT` (0x100)

  An external host-initiated direct operation

  The particular direct operation is specified in the lower 8 bits.

  In the case of a burst operation, the number of operations is specified as an argument.

  For extra bit definitions, see "External-Initiated Direct Operation Bit Definitions" on page C-9.

- `#define HOST_DELAY` (0x200)

  Provides a delay

  The number of peripheral cycles is specified as an argument.

- `#define HOST_REPEAT` (0x300)

  The Next command is repeated a specified number of times.

The number of repetitions is specified as an argument.

- `#define HOST_NULL` (0x400)

  No operation

  Provides a one peripheral cycle delay

  No argument

- `#define HOST_STOP` (0x500)

  Terminates processing of the control file until a reset operation occurs

  No argument

- `#define HOST_WAIT_FOR_DMA` (0x600)

  Suspends processing of the control file until a host DMA interrupt is generated

  No argument

External-Initiated Direct Operation Bit Definitions

For external-initiated direct operations, the following bits further define the operation.

- `#define HOST_READ` (0x00)

  A host-port read, memory-write operation

- `#define HOST_WRITE` (0x01)

  A host port write, memory read operation

  Data memory (HSC0) or IO memory (HSC1)

- `#define HOST_HSC0` (0x02)

  Simulates the external pin HSC0 being high

- `#define HOST_HSC1` (0x04)

  Simulates the external pin HSC1 being high

- `#define HOST_MEMHOST_HSC0`

- `#define HOST_ONCHIP_IOHOST_HSC1`

  Provided for convenience

- `#define HOST_SLAVE` (0x00)

  A single direct operation

- `#define HOST_BURST` (0x08)

  A burst operation

  The number of accesses is specified by an argument.

- `#define HOST_LATCH` (0x00)

  The operation is conducted with **Address Latch Enable**.

- `#define HOST_CYCLE` (0x10)

  The operation is conducted with **Address Cycle Control**.

Host Port Window in VisualDSP++

You can view the host port registers from the Host Port window.

To open the Host Port window:

1. From the **Registers** menu, choose **Peripherals**.
2. Choose **Host**.

Unsupported Features

Packing is not simulated. Words are read from and written to stream files as complete words.

Example – DMA Transfer to the Host Port

The HPI example sets up a DMA transfer to the host port and generates an interrupt upon completion.

Set up a stream file to accept the output from the stream `HOST.TX`.

This example is located in the following folder.

```
Simulator Peripheral Examples\HPI
```

Serial Peripheral Interface (SPI)

The SPI peripheral module provides industry-standard synchronous serial link functionality. Two SPI peripherals are on the ADSP-2191 DSP.

In the external four-wire interface, only the `MOSI` and `MISO` pins are simulated while `SPICLK` and `PIO_nSPISSIN` are not. You can use the stream interface with the `MOSI` and `MISO` pins only.

The simulator supports DMA and non-DMA modes as both master and slave, which takes into account baud rates but not clock phase and polarity.

The simulator supports the following SPI registers.

- Control[15:0] – Control
- Flag[15:0] – Flag (content is ignored)
- Status[6:0] – Status
- TDBR[15:0] – Transmit Data Buffer
- RDBR[15:0] – Receive Data Buffer
- Baud [15:0] – Baud Rate
- RDBR_SHADOW[15:0] – Receive Data Buffer (Shadow)
- DMA Current Pointer [15:0]
- DMA Configuration [15:0]
- DMA Start Page [8:0]
- DMA Start Address [15:0]
- DMA Word Count [15:0]
- DMA Next Descriptor [15:0]
- DMA Descriptor Ready [0]
- DMA Interrupt [0]

SPI Global Status and Control

Table C-5 describes the bits in the SPI Global Status and Control register.

Table C-5. Register Bits

Bit	Status[6:0]	Global Status and Control
0	SPIF	Single-word transfer complete
1	MODF	Mode fault error for master device (**not supported**)
2	TXE	Transmission error: transmission occurred with no new data in TDBR
3	TXS	TDBR status: **0** = empty, **1** = full
4	RBSY	Receive error: data is received with RDBR full
5	RXS	RBDR status: **0** = empty, **1** = full
6	TXCOL	Transmit collision error: possible corrupted data transmitted

The mode fault error bit in the SPIFLG register is not supported.

The configuration bits in Table C-6 do not affect the simulator.

Table C-6. Register Bits that Do Not Affect Simulation

Bit	Config[15:0]	SPI Configuration Register
4	PSSE	PIO_nSPISSIN input for master
5	EMISO	MISO pin as output
10	CPHA	Clock phase
11	CPOL	Clock polarity
13	WOM	Open drain data output

SPI Signal Usage

The following signals are not recognized.

- PIO_nSPISSIN
- SPICLK

Modes of Operation

Master mode and slave mode can each be operated with or without DMA.

Master Mode Operation (No DMA)

The core writes to SPICTL and SPIBAUD to configure the device as master and to set the word length and baud rate. Transfer starts upon a write to TDBR or a read from RDBR, depending on the configuration of the TIMOD bits in SPICTL. Data is read from the MISO pin into RDBR and sent out to the MOSI pin from TDBR.

If TDBR remains empty or RDBR remains full, the SPI operates according to the SZ and GM bits in the SPICTL register. The simulator does **not** generate the programmed clock pulses on SPICLK during transmission.

Slave Mode Operation (No DMA)

Slave mode operation is summarized as follows.

1. The core writes to SPICTL to configure the device as slave and to set the word length.
2. In hardware, the transfer starts once a falling edge of the PIO_nSPISSIN is detected.

   The simulator does **not** simulate transitions of the PIO_nSPISSIN pin. Therefore, transmission starts as soon as slave mode is properly set and the SPI device is enabled.

3. Transmission ends after the proper number of clock cycles but not when PIO_nSPISSIN is released.

4. Data is read from the MOSI pin into RDBR, and is sent out to the MISO pin from TDBR.

If TDBR remains empty or RDBR remains full, the SPI operates according to the SZ and GM bits in the SPICTL register. The simulator does **not** synchronize transmission on SPICLK active edges.

Master Mode DMA Operation

Master mode DMA operation is summarized as follows.

1. The core writes to SPICTL and SPIBAUD to configure the device as master and to set the word length and baud rate. TIMOD bits are set to the "**Transmit or Receive with DMA**" mode.

2. The core generates one or more descriptor blocks in page **0** of the memory space.

3. The core writes to the DMA Configuration register to enable the DMA engine.

4. If the device is configured for transmit operation, the transfer starts when the DMA engine reads data from memory into the DMA buffer. If the device is configured for receive operation, the transfer starts when the DMA engine reads data from DMA buffer into memory.

5. Data is read from the MISO pin into RDBR and is sent out to the MOSI pin from TDBR until the DMA Word Count register reaches **0**.

If TDBR remains empty or RDBR remains full, the SPI operates according to the SZ and GM bits in the SPICTL register.

The simulator does **not** generate the programmed clock pulses on SPICLK during transmission.

Slave Mode DMA Operation

Slave mode DMA operation is summarized as follows.

1. The core writes to SPICTL to configure the device as slave and to set the word length. TIMOD bits are set to the "**Transmit or Receive with DMA**" mode.
2. The core generates a DMA receive descriptor block in page **0** of the memory space.
3. The core writes to the DMA Configuration register to enable the DMA engine. The head of the descriptor block is written to the DMA Next Descriptor register.
4. In hardware the transfer starts once a falling edge of the PIO_nSPISSIN is detected. Note that the simulator does not simulate transitions of the PIO_nSPISSIN pin. Therefore, transmission starts as soon as the slave mode DMA is properly set and the SPI device is enabled.
5. Transmission ends when the DMA Word Count register reaches **0**.
6. The core can continue by queuing the next command buffer descriptor block.
7. Data is read from the MOSI pin into RDBR and is sent to the MISO pin from TDBR.

SPI with Streams

The SPI can read and write from the MISO and MOSI pins by using the streams functionality in VisualDSP++.

You can attach a file to the following device names.

```
SPI0.MISO

SPI0.MOSI

SPI1.MISO

SPI1.MOSI
```

The format of the input file is as follows.

Data0

Data1

…

DataN

Data can be 8 or 16 bits long, depending on the word length set in the SPICTL register.

The MISO and MOSI pins are used as input or output pins, depending on whether the SPI device is configured as master or slave.

Slave Mode DMA Example

This example is located in the following folder:

```
Simulator Peripheral Examples\SPI
```

The example reads data from the SPI0.MOSI pin to a memory buffer by using DMA, and compares the input with internal values.

After you load the project and build the executable file, choose **Streams** from the **Tools** menu to attach the `SPI0.MOSI` pin to the following input file (provided with the example).

```
0x0001

0x0002

0x0004

0x0008

0x0010

0x0020

0x0040

0x0080

0x0100

0x0200

0x0400

0x0800

0x1000

0x2000

0x4000

0x8000
```

For information about using streams, refer to the VisualDSP++ on-line Help.

Serial Port (SPORT) Peripheral

The Serial Port peripheral provides a simulation of serial port functionality. The module supports all modes of operation, including Direct Memory Access (DMA), Multichannel Mode (MCM), and interrupt-driven modes for both transmit (TX) and receive (RX). Additionally, the zero-fill, sign-extend, and A-Law/μ-Law companding data types are supported on transmit and receive. The ADSP-2191 has three serial ports, which are supported in the simulator.

The Serial Port has the following registers in memory mapped register space plus the offsets shown in Table C-7, Table C-8, and Table C-9.

Table C-7. MCM Select and Configuration Registers

MCM Select and Configuration Registers	SPT0	SPT1	SPT2
TX Channel Select MTCS[0:7]	0x0A09-10	0x0C09-10	0x0E09-10
RX Channel Select MRCS[0:7]	0x0A11-18	0x0C11-18	0x0E11-18
Multichannel Configuration (register 1)	0x0A19	0x0C19	0x0E19
Multichannel Configuration (register 2)	0x0A1A	0x0C1A	0x0E1A

Table C-8. Configuration and Status Registers

Configuration and Status Registers	SPT0	SPT1	SPT2
Serial Port TX Configuration	0x0A00	0x0C00	0x0E00
Serial Port RX Configuration	0x0A01	0x0C01	0x2E01
Serial Port TX Buffer	0x0A02	0x0C02	0x0E02
Serial Port RX Buffer	0x0A03	0x0C03	0x0E03
Serial Port TX Serial Clock Divisor	0x0A04	0x0C04	0x0E04
Serial Port RX Serial Clock Divisor	0x0A05	0x0C05	0x0E05
Serial Port TX Frame Synch Divisor	0x0A06	0x0C06	0x0E06

Table C-8. Configuration and Status Registers (Cont'd)

Configuration and Status Registers	SPT0	SPT1	SPT2
Serial Port RX Frame Synch Divisor	0x0A07	0x0C07	0x0E07
Serial Port Status	0x0A08	0x0C08	0x0E08

Table C-9. DMA Control Registers

DMA Control Registers (these conform to the standard ADSP-2191 DMA descriptors)	SPT0	SPT1	SPT2
RX DMA Descriptor Pointer	0x0B00	0x0D00	0x0F00
RX DMA Configuration	0x0B01	0x0D01	0x0F01
RX DMA Page	0x0B02	0x0D02	0x0F02
RX DMA Address	0x0B03	0x0D03	0x0F03
RX DMA Count	0x0B04	0x0D04	0x0F04
RX DMA Next Descriptor Pointer	0x0B05	0x0D05	0x0F05
RX DMA Descriptor Ready	0x0B06	0x0D06	0x0F06
RX DMA IRQ Status	0x0B07	0x0D07	0x0F07
TX DMA Descriptor Pointer	0x0B80	0x0D80	0x0F80
TX DMA Configuration	0x0B81	0x0D81	0x0F81
TX DMA Page	0x0B82	0x0D82	0x0F82
TX DMA Address	0x0B83	0x0D83	0x0F83
TX DMA Count	0x0B84	0x0D84	0x0F84
TX DMA Next Descriptor Pointer	0x0B85	0x0D85	0x0F85
TX DMA Descriptor Ready	0x0B86	0x0D86	0x0F86
TX DMA IRQ Status	0x0B87	0x0D87	0x0F87

Input and Output

Each of the three serial ports has two standard VisualDSP++ data streams.

- SPT0: SPT0.TX, SPT0.RX
- SPT1: SPT1.TX, SPT1.RX
- SPT2 SPT2.TX, SPT2.RX

You can use these streams to associate an input data file with the RX of SPT0, SPT1, or SPT2 or to capture the values of TX into a file. Values in a stream are associated with the RX/TX parallel registers in the serial ports, not the actual serial pins. Therefore, for each "word" received by the SPORT, a value is consumed or generated in the stream file. Simulation of the transfer duration includes consideration of the frame synch and clock divisors.

To associate files with SPT RX/TX, choose **Streams** from the **Settings** menu.

Serial Port Windows in VisualDSP++

The serial port registers appear in the Serial Port window.

To view these registers:

1. From the **Registers** menu, choose **Peripherals**.
2. Choose **SPT0**, **SPT1**, or **SPT2**.

These windows provide a compact view of all the registers for each SPORT. You can access individual status and configuration register fields by creating custom register windows.

Unsupported Features

You can manipulate all the serial port configuration bits. Configuration related to frame synchs does not always have a significant impact on the simulation. For example, although the `FSR` bits are used, the following DSP capabilities are not simulated.

- Internal or external synch
- Synch polarity
- Early or late synch

Serial clock and frame synch divisor registers are simulated. Therefore, they impact the duration required to transfer words.

Example – SPORT DMA

The example program (`spt`) is located in the following folder.

```
Simulator Peripheral Examples\Sport
```

The example uses an input stream and performs a DMA to memory.

To run the example, open the `README.txt` file and follow the instructions.

Universal Asynchronous Receiver/Transmitter (UART) Peripheral

The ADSP-2191 UART peripheral provides a simulation of UART port functionality. It supports all modes of operation, including Direct Memory Access (DMA) and interrupt driven modes for both transmit (TX) and receive (RX).

Table C-10 lists the UART registers used to control its operation.

Table C-10. UART Registers

Configuration and Status	Real	Phantom (if different)
UART Transmit Hold Register (THR)	0x1400	
UART Divisor Latch (Low Byte) (DLL)	0x1400	0x1408
UART Receive Buffer Register (RBR)	0x1400	0x1409
UART Interrupt Enable Register (IER)	0x1401	
UART Divisor Latch (High Byte) (DLH)	0x1401	0x140A
UART Interrupt Identification Register (IIR)	0x1402	
UART Line Control Register (LCR)	0x1403	
UART Modem Control Register (MCR)	0x1404	
UART Line Status Register (LSR)	0x1405	
UART Modem Status Register (MSR)	0x1406	
UART Scratch Register (SCR)	0x1407	

The *Phantom addresses* appear in the VisualDSP++ IOM memory display. The registers are **not** addressable by user code at those locations.

Table C-11 lists the DMA Control registers.

Table C-11. DMA Control Registers

DMA Control Registers (these conform to the standard ADSP-2191 DMA descriptors)	SPT0
RX DMA Descriptor Pointer	0x1500
RX DMA Configuration	0x1501
RX DMA Page	0x1502
RX DMA Address	0x1503
RX DMA Count	0x1504
RX DMA Next Descriptor Pointer	0x1505
RX DMA Descriptor Ready	0x1506
RX DMA IRQ Status	0x1507
TX DMA Descriptor Pointer	0x1580
TX DMA Configuration	0x1581
TX DMA Page	0x1582
TX DMA Address	0x1583
TX DMA Count	0x1584
TX DMA Next Descriptor Pointer	0x1585
TX DMA Descriptor Ready	0x1586
TX DMA IRQ Status	0x1587

Input and Output

Each UART has two standard VisualDSP++ data streams:

- UART.TX
- UART.RX

UART Window in VisualDSP++

The serial port registers appear in the UART window.

To view these registers:

1. From the **Registers** menu, choose **Peripherals**.
2. Choose **UART** to open the UART window.

This window provides a compact view of all the registers for the UART. You can access individual status and configuration register fields by creating a custom register window.

Unsupported Features

You can manipulate all the UART configuration bits. Currently, you cannot simulate the data error (Framing Error, Parity Error, Break Interrupt) conditions or the **Modem Status** status bits (Data Carrier Detect, Ring Indicator, Data Set Ready, Clear To Send). You can specify **Set Break** in the Line Control (LCR) register, but this setting has no effect. The current simulator does not model the `IRCR` register.

Example

The UART example performs a 32-word DMA receive after configuring the interrupt controller, the SPORT, and the DMA engine.

This example is located in the following folder.

```
Simulator Peripheral Examples\UART
```

Timer (TMR) Peripheral

The ADSP-2191 DSP Timer peripheral provides general-purpose timer functionality. The ADSP-2191 DSP has three timer peripherals that are external to the core.

The simulator supports each of the three functional modes:

- PWM_OUT Mode – Pulse Width Modulation
- WDTH_CAP Mode – Pulse Width and Period Capture
- EXT_CLK Mode – External Event Watchdog

Each timer has one bidirectional chip pin, `TMR_PIN`, and one auxiliary module input pin, `AUX_IN`. The simulator does **not**, however, distinguish between these two pins (as described below).

The simulator supports the architected registers for each timer. These registers are:

- Config [15:0] – Configuration
- Width [31:0] – Pulse Width
- Period [31:0] – Pulse Period
- Counter [31:0] – Timer Counter

Timer Global Status and Control

Table C-12 describes the bits in the Timer Global Status and Control register.

Table C-12. Timer Global Status and Control Register Bits

Bit	Status[15:0]	Global Status and Control
0	IRQ0	Timer 0 IRQ “Write one to clear” (also an output)
1	IRQ1	Timer 1 IRQ “Write one to clear” (also an output)
2	IRQ2	Timer 2 IRQ “Write one to clear” (also an output)
3	Reserved	Timer 3 (reserved)
4	OVF_ERR0	Timer 0 Counter overflow
5	OVF_ERR1	Timer 1 Counter overflow
6	OVF_ERR2	Timer 2 Counter overflow
7	Reserved	Timer 3 (reserved)
8	TIMEN0	Timer 0 – “Write one to set”
9	TIMEN0	Timer 0 – “Write one to clear”
10	TIMEN1	Timer 1 – “Write one to set”
11	TIMEN1	Timer 1 – “Write one to clear”
12	TIMEN2	Timer 2 – “Write one to set”
13	TIMEN2	Timer 2 – “Write one to clear”
14	Reserved	Timer 3 (reserved)
15	Reserved	Timer 3 (reserved)

Table C-13 describes the bits in the Timer Configuration register. Note that bit 5 does not affect the simulator.

Table C-13. Timer Configuration Register Bits

Bit	Config[15:0]	Timer Configuration Register	Simulator Semantics
1:0	MODE_FIELD	00 – Reset State - unused 01 – PWM_OUT Mode 10 – WDTH_CAP Mode 11 – EXT_CLK Mode	
2	PULSE_HI	1 – Positive Active Pulse 0 – Negative Active Pulse	
3	PERIOD_CNT	1 – Count to end of Period 0 – Count to end of Width	
4	IRQ_ENA	1 – Interrupt Request Enable 0 – Interrupt Request Disable	
5	AUX_IN_SEL	1 – AUX_IN Select 0 – TMR_PIN Select	For input the simulator uses *streams* as described below. When you use streams as input to the timers, no distinction is made between AUX_IN and TMR_PIN.
15:6	Unused		

Timer Signal Usage

The signals listed in Table C-14 are not recognized. Note that "NA" designates *not applicable*.

Table C-14. Unrecognized Timer Signals

Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
AUX_IN_SEL	Unused	**1** – Select AUX_IN Input **0** – Select TMR_PIN Input	Unused
TMR_AUX_IN	Unused	Auxiliary Input waveform	Unused
WAKE_UP	NA	NA	NA

Modes of Operation

The modes of operation are:

- PWM (Pulse Width Modulation)
- Width capture
- External clock

Timer with Streams Usage

This section describes the timer with streams usage.

WDTH_CAP Mode

In width capture (WDTH_CAP) mode, the timer counts the number of clocks in both the width and period. The waveform that the timer reads is attached via the **Streams** dialog box in VisualDSP++.

You can attach a file to the following device names.

- TIMER0_WDTH_CAP
- TIMER1_WDTH_CAP
- TIMER2_WDTH_CAP

The format of the input file is as follows.

PERIOD_COUNT

WIDTH_COUNT

PERIOD_COUNT

WIDTH_COUNT

In WDTH_CAP mode, the timer reads two 32-bit values from the input file. The first value is the number of pulses (clocks) in the period. The second value is the number of pulses in the width.

When `PULSE_HI` is set, the timer delivers high widths and low periods. When `PULSE_HI` is not set, the timer delivers low widths and high periods.

Example Streams Data File

The timer example is located in the following folder.

```
Simulator Peripheral Examples\TIMER
```

If you are using streams with WDTH_CAP mode, your data file should look like Table C-15.

Table C-15. Streams Data File Attached to `TIMER0_WDTH_CAP`

Data in File	Semantics
0x2	First period value
0x1	First width value

Table C-15. Streams Data File Attached to TIMER0_WDTH_CAP (Cont'd)

Data in File	Semantics
0x4	Second period value
0x2	Second width value
0x8	Third period value
0x4	Third width value

When MODE = WDTH_CAP and PULSE_HI = 0, the resulting waveforms are described in Table C-16.

Table C-16. Resulting Waveforms

0 1	First pulse chain
0 0 1 1	Second pulse chain
0 0 0 0 1 1 1 1	Third period value

To program the timer in WDTH_CAP mode:

1. Write to the Timer Configuration register:

   IRQ_ENA = 1

   PERIOD_CNT = 0

   PULSE_HI = 0

   MODE_FIELD = 0x10 = WIDTH_CAP mode

2. Write to the Timer Global Status and Control register to enable the timer.

   The timer, if connected to a data file via the streams interface, now delivers pulses as outlined above.

External Clock Mode

The external clock is limited to 66.5 MHz. Therefore, the simulator automatically divides the peripheral clock when a timer is enabled in external clock mode.

Memory DMA (MEMDMA) Peripheral

The Memory DMA (MDMA) peripheral enables you to move data and instructions between internal memory and off-chip memory. The MDMA uses the ADSP-219x style of distributed DMA, whereby each DMA channel is located in the peripheral itself. The MDMA has dedicated Write and Read DMA channels. This DMA scheme is further based on *descriptors*, which describe the type of transfer to be carried out. These descriptors are managed in internal memory and are downloaded to the MDMA.

For details about DMA in the ADSP-2191, refer to the *ADSP-219x/2191 Hardware Reference*.

Modes

Of the two DMA modes (Auotbuffer and Descriptor block), only Descriptor block is supported in the MEMDMA peripheral. This mode is set up as follows.

HEAD refers to the value of the current descriptor.

1. The software memory writes **HEAD+1,+2,+3,+4** to internal memory.
2. The software memory writes **HEAD+0** to internal memory. Bit 15 (ownership) must be set to **1**.
3. The software I/O writes the **DescReady** register of the respective DMA channel (if necessary).

4. The software I/O writes the **NXTPTR** register of the DMA channel (only to start the first descriptor).
5. The software I/O writes the **Configuration** register setting **DMAEn** high (only to start the first descriptor).

Note:

- Burst mode is not simulated in the simulator. Therefore, values in memory show up sooner than they would appear in the hardware.
- Simulation of the MEMDMA is not cycle accurate.

Registers

Table C-17 describes the Write Channel registers.

Table C-17. Write Channel Registers

Address	Name	Description
0x900	Current PTR	Pointer of current descriptor being worked on
0x901	DMA Config DMA Enable (bit 0/1 bit) Direction (bit 1/1 bit) Interrupt on Completion (bit 2/1 bit) Data Type (bit 3/1 bit) DMA Buffer Clear (bit 7/1 bit) DMA Buffer Status (bit 12/2 bit) Ownership (bit 15/1 bit)	 Enable Bit - Start transfer Locked as 1 for Write Channel Generate interrupt on completion of transfer 16- or 24-bit transfer Clear Write channel FIFO DMA Buffer Status – 00 empty Descriptor Ownership **0** = DSP, **1** = DMA
0x902	Start Page Start Page (bit 0/8 bits) Space (bit 8/1 bit)	 Page of transfer **0** = mem, **1** = boot
0x903	Start Address	Current address of memory writes
0x904	DMA Count	Current count of transfer
0x905	Next Descriptor	Location of next descriptor
0x906	Descriptor Ready	Indicates descriptor is ready for loading
0x907	IRQSTAT	Status of interrupt

Table C-18 describes the Read Channel registers.

Table C-18. Read Channel Registers

Address	Name	Description
0x980	Current PTR	Pointer of current descriptor being worked on
0x981	DMA Config DMA Enable (bit 0/1 bit) Direction (bit 1/1 bit) Interrupt on Completion (bit 2/1 bit) Data Type (bit 3/1 bit) DMA Buffer Clear (bit 7/1 bit) DMA Buffer Status (bit 12/2 bit) Ownership (bit 15/1 bit)	 Enable Bit - Start transfer Locked as 0 for Read Channel Generate interrupt on completion of transfer 16- or 24-bit transfer Clear Write channel FIFO DMA Buffer Status – 00 empty Descriptor Ownership **0** = DSP, **1** = DMA
0x982	Start Page Start Page (bit 0/8 bits) Space (bit 8/1 bit)	 Page of transfer 0 = mem, 1 = boot
0x983	Start Address	Current address of memory writes
0x984	DMA Count	Current count of transfer
0x985	Next Descriptor	Location of next descriptor
0x986	Descriptor Ready	Indicates descriptor is ready for loading
0x987	IRQSTAT	Status of interrupt

MEMDMA does not generate errors in the transfer. No errors are generated as long as the registers are programmed correctly. Refer to the *ADSP-219x/2191 Hardware Reference* for more information.

Example – MEMDMA Transfer

This example demonstrates multiple internal to internal DMA transfers within the memory of the ADSP-2191 DSP.

The example uses these two methods to check for DMA completion:

- Interrupts: At the end of the first transfer, a DMA completion interrupt is generated.
- Ownership bit of the Configuration word: The second DMA polls this bit in memory to see if it is cleared. At the end of the transfer, the DMA engine writes `0` (zero) to this bit in memory to transfer the control of DMA descriptor block to the DSP.

The MEMDMA example is located in the following folder.

```
Simulator Peripheral Examples\MEMDMA
```

Simulator Instruction Timing Analysis Overview

The ADSP-219x Family Simulator is a cycle-accurate simulator with a six-stage pipeline. The simulator functionality models the behavior of the ADSP-2191 DSP by updating registers, memory, and peripherals. The processor state is updated after each instruction execution.

The correct execution count trails the execution of the instruction by at least the length of the sequencer pipeline and maybe more. When you break execution of the simulator, the cycle count may not be accurate. At the completion of the program, the cycle count is correct.

The Pipeline Viewer enables you to understand the execution timing of your program. It shows the flow of instructions through the pipeline and any stalls due to sequencer or memory events.

For information about configuring and using the Pipeline Viewer, see "Pipeline Viewer Window" on page 2-93 or the VisualDSP++ on-line Help.

Cycle-Accurate Simulator

The ADSP-219x Family Simulator is classified as a cycle-accurate simulator because it models the behavior of the following.

- Instruction set
- Processor sequencer
- Memory hierarchy, including the external SRAM
- Registers, including the MMRs
- All the core peripherals
- All memory transactions

Instruction Pipeline

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions are required, the DSP executes instructions from program memory in sequential order by incrementing the look-ahead address. Using its instruction pipeline, the DSP processes instructions in six clock cycles:

- **Look-Ahead Instruction** (**LA**). The DSP determines the source for the instruction from inputs to the look-ahead address multiplexer.
- **Pre-fetch Instruction** (**PA**) and **Fetch Instruction** (**FA**). The DSP reads the instruction from either the on-chip instruction cache or from program memory.

- **Address Decode (AD)** and **Instruction Decode (ID)**. The DSP decodes the instruction and generates conditions that control instruction execution.
- **Execute (PC)**. The DSP executes the instruction, and completes the operations specified by the instruction in a single cycle.

These cycles overlap in the pipeline. In sequential program flow, when one instruction is being fetched, the instruction fetched three cycles previously is being executed. With few exceptions, sequential program flow has a throughput of one instruction per cycle. The exceptions are the two-cycle instructions: 16- or 24-bit immediate data writes to memory with indirect addressing, long jump (`Ljump`) and long call (`Lcall`).

Any non-sequential program flow can potentially decrease the DSP's instruction throughput. Non-sequential program operations include:

- Program memory data accesses that conflict with instruction fetches
- Jumps
- Subroutine calls and returns
- Interrupts and returns
- Loops (of less than five instructions)

Delay in the Pipeline Viewer Window

The code sequence in Figure C-1 illustrates the lag.

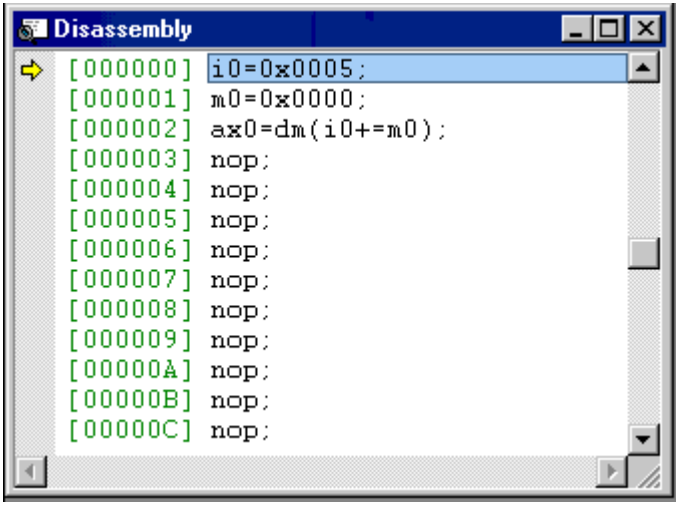

Figure C-1. Lag in the Pipeline

In the above code sequence, data register `AX0` has just been assigned from system register `CNTR` at instruction address `0x00`. The instruction at address `0x02` is the next instruction to be executed.

The **Pipeline Viewer** window shows the execution of the instruction at address 0.

The **Pipeline Viewer** window displays:

- Pipeline stages
- Cycle-by-cycle analysis of the instructions passing through the pipeline stages

Figure C-2 shows the functional simulation of the instruction at address 0 through 4. The pipeline analysis of the instructions is displayed in each stage.

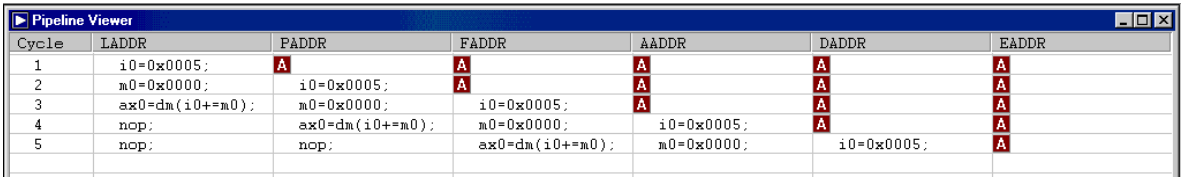

Figure C-2. Instruction at Address 0

Figure C-3 shows that the instruction at address 0x5 is about to be executed. The instructions at addresses 0x00, 0x01, 0x02, 0x03 and 0x04 have already been executed in the cycle-accurate simulator.

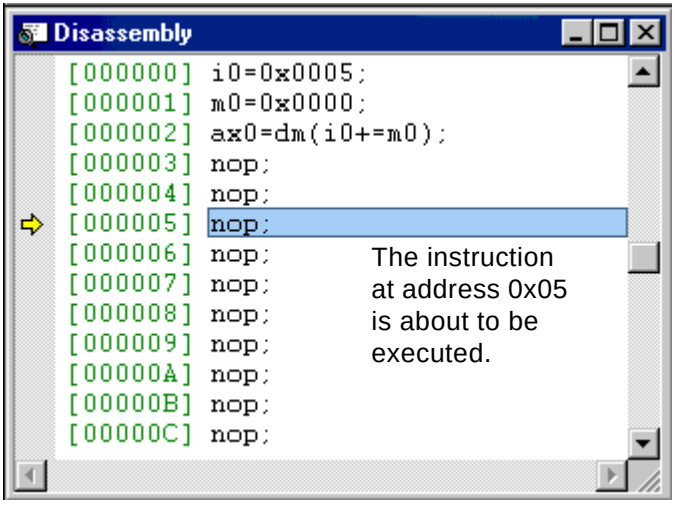

Figure C-3. Pipeline Commit Stage

The **Pipeline Viewer** window (Figure C-4) shows that the instruction at address `0x02` has just reached the commit stage of the timing analysis.

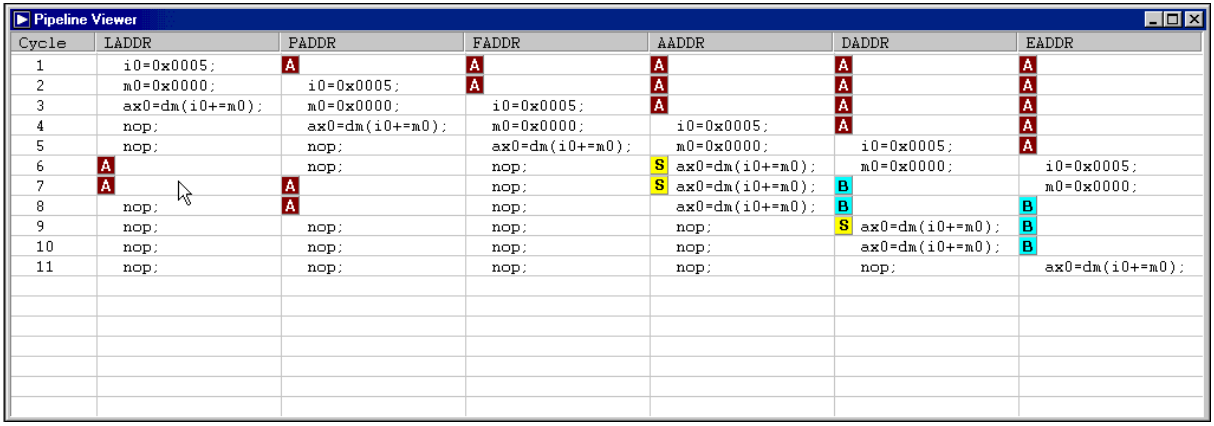

Figure C-4. Instruction at Address 0x02 Reaches the Commit Stage

The window highlights the instruction at address `0x02` as it progressed through the pipeline. Table C-19 compares the cycles required at each stage.

Table C-19. Cycles Required at Each Stage

Cycle	Stage
2	LADDR. The cycle count has increased by one.
3	PADDR
4	FADDR
5	AADDR
6	DADDR
7	EADDR

The **Pipeline Viewer** window detects a stall (symbolized by S) at cycle 6, stage AADDR.

To learn why the stall occurred:

1. Press and hold down the keyboard's **Ctrl** key.
2. Move the mouse over the S icon.

A message window appears, as shown in Figure C-5, to indicate that the stall is due to a RAW (read after write) hazard.

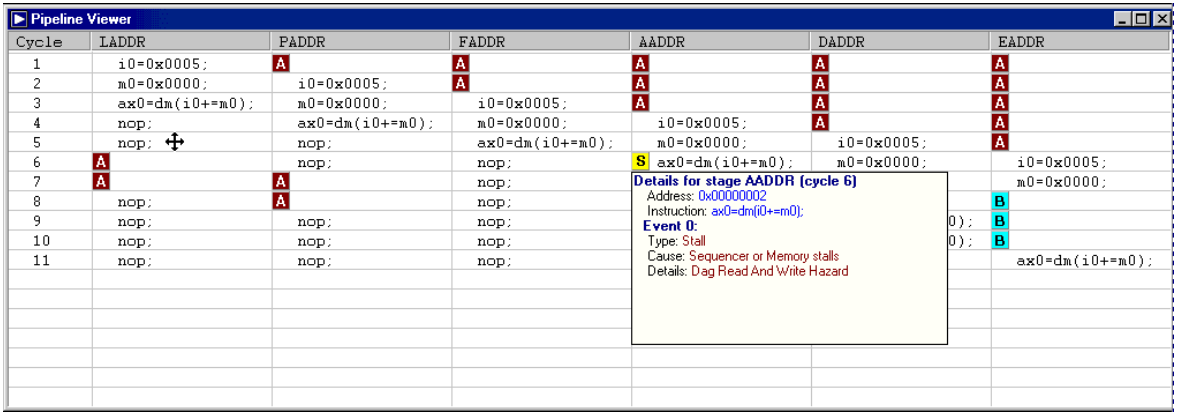

Figure C-5. Example Message

In Figure C-5, dag registers (I0,M0) were modified next to a data move instruction, which also modifies a dag register.

The pipeline increases the cycle count by one cycle for this stall by inserting a stall *bubble* into the pipeline at cycle 7, stage DADDR.

Pipeline Stages

Table C-20 shows the ADSP-219x processor's pipeline stages.

Table C-20. Pipeline Stages

Stage	Abbreviation in Pipeline Viewer
Look Ahead	LADDR
PreFetch Address	PADDR
Fetch Address	FADDR
Address Decode	AADDR
Instruction Decode	DADDR
Execution	EADDR

Pipeline Viewer Window Messages

The **Pipeline Viewer** window displays informational messages for instructions indicated with an event icon.

These types of messages may appear:

- Stalls detected
- Aborts detected

Stalls Detected Messages

Table C-21 shows the messages that occur when a stall is detected.

Table C-21. Stalls Detected Messages

Message	Explanation	Example
Indirect Conflict	Indirect branch in AADDR with a Dag modified from the same bank	AY1=DM(12+=2); Call(I1)(db);
Dag Access	Stage A Dag Register is modified next to Memory Access	
IOPG Access	Stage AADDR IOPG register is modified next to an IO Mem access	IOPG=6 IO(0x204)=ax0
Dag Modify Hazard	Stage AADDR a Dag is modified next to an assignment	I1=i5; Modify(I1+=24);
External Memory Access	Access to an external memory address	
IO Memory Access	Access to IO memory	IO(0x203)=ax0
Memory Conflict	DADDR and EADDR have bus conflict	DM(I1+=M0)=0x400; DM(I1+=M0)=0x000;
Dag read and Write Hazard	Stage AADDR	I1=0xEC00; DM(I1+=M0)=0x400;
Dag Stall	Any other Dag stalls not described in Viewer	

Aborts Detected Messages

Table C-22 shows the messages that occur when an abort is detected.

Table C-22. Aborts Detected Messages

Message	Explanation	Example
change-of-flow abort	A branch	CALL (I0)
rti change-of-flow abort	Return from interrupt aborts	RTI;
mispredicted change-of-flow abort	Aborts because of mispredicted branches	AY1=0X0005; AX1-AY1; IF GE JUMP _L_50006 [.+12]
hardware loop bottom abort		
interrupt abort	Instructions in the pipeline are aborted because of an interrupt	SETINT 0X05;

Boot Simulation

Table C-23 summarizes the boot simulation support for the ADSP-218x and ADSP-219x simulator targets.

Table C-23. Boot Simulation Support for ADSP-21xx Targets

Simulator Target	Boot Device/Width	Boot Format(s)	File Extension
ADSP-2181, ADSP-2184, ADSP-2185, ADSP-2186, ADSP-2186, ADSP-2187, ADSP-2189	PROM	Motorola S2 Intel Hex	*.BNM
ADSP-219x	No boot simulation support		
ADSP-2191, ADSP-2195, ADSP-2196	PROM, 8 bit or 16 bit	Intel Hex 32	*.LDR
ADSP-2192-12	No boot simulation support		

Simulating Boot Loading for ADSP-218x Targets

Complete the following steps to prepare a boot-loadable program and simulate boot loading for ADSP-218x targets.

1. Create the splitter output file. For details, see the *VisualDSP++ 3.5 Loader and Utilities Manual for 16-Bit Processors*.
2. From the **Settings** menu, choose **Simulator** and **Boot Mode** and ensure that **Boot from EPROM** is selected.
3. From the **Settings** menu, choose **Simulator** and **Load ROM File**. Then select the .BNM file to boot.
4. Select **Reset** from the **Debug** menu or click the Reset  button. The simulator displays the "BOOTING" status message.

   If you are debugging the loader, start debugging here. Otherwise, proceed to the next step.

5. Choose **Load Symbols** from the **File** menu and select the `.DXE` from which the `.BNM` was generated.
6. Set a breakpoint at the entry point to your application.
7. Choose **Run** from the **Debug** menu, click the **Run**  button, or press the **F5** key.

   You may go to breakpoints while the application is loading. Choose **Run** again until you stop at the breakpoint for your application.
8. Debug your application.

Simulating Boot Loading for ADSP-219x Targets

Complete the following steps to prepare a boot-loadable program and simulate boot loading for ADSP-2191, ADSP-2195, and ADSP-2196 targets.

1. Create the loader file. For details, see the *VisualDSP++ 3.5 Loader and Utilities Manual for 16-Bit Processors*.
2. From the **Settings** menu, choose **Boot**, **PROM.RX**, **8 Bits** or **16 Bits**, and **PROM File**.
3. Select the `.LDR` file.

   The "`Boot Settings`" message is displayed. The boot is set up.
4. Select **Reset** from the **Debug** menu or click the Reset  button. The application is loaded.
5. Choose **Load Symbols** from the **File** menu and select the `.DXE` from which the `.LDR` was generated.
6. Run your application.

I INDEX

F