

CodeWarrior® Debugger Manual



Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Metrowerks CodeWarrior copyright ©1993–1996 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international

Metrowerks Corporation
2201 Donley Drive, Suite 310
Austin, TX 78758
U.S.A.

Canada

Metrowerks Inc.
1500 du College, Suite 300
Ville St-Laurent, QC
Canada H4L 5G6

Mail order

Voice: (800) 377-5416
Fax: (512) 873-4901

World Wide Web

<http://www.metrowerks.com>

Registration information

register@metrowerks.com

Technical support

support@metrowerks.com

Sales, marketing, & licensing

sales@metrowerks.com

America Online

keyword: Metrowerks

CompuServe

goto Metrowerks

Table of Contents

1 Introduction	9
Overview of the Debugger Manual	9
Read the Release Notes	10
System Requirements	10
Installing MW Debug	10
Starting Points	11
Where to Learn More	12
2 Getting Started	13
Getting Started Overview	13
Preparing for Debugging	13
Setting Up a Project for Debugging	13
Setting Up a File for Debugging	14
Launching the Debugger	15
Launching the Debugger from a Project	16
Launching the Debugger Directly	16
Symbolics Files	17
SYM File Names	18
Matching the SYM File to an Executable File	18
CodeView Symbols	18
3 What You See	19
What You See Overview	19
Program Window	19
Stack Crawl Pane	21
Variables Pane	21
Debugger Tool Pane	23
Source Pane	23
Viewing source code as assembly	24
Function Pop-up Menu	26
Browser Window	27
File Pane	28
Function Pane	30
Globals Pane	31

Table of Contents

Placing globals in a separate window	31
Source Pane	32
Function Pop-up Menu	33
Expression Window.	33
Breakpoint Window.	35
Log Window	35
Variable Window	36
Array Window	37
Memory Window.	39
Register Windows	40
Process Window	41
4 Basic Debugging	43
Basic Debugging Overview	43
Starting Up	44
Running, Stepping, and Stopping Code	44
The Current-Statement Arrow	46
Running Your Code	46
Stepping a Single Line	48
Stepping Into Routines	49
Stepping Out of Routines	50
Skipping Statements	50
Stopping Execution.	52
Killing Execution.	52
Navigating Code	53
Linear Navigation	53
Call-Chain Navigation	54
Browser Window Navigation	55
Using the Find Dialog.	58
Breakpoints	60
Setting Breakpoints.	60
Clearing Breakpoints	61
Temporary Breakpoints	61
Viewing Breakpoints	62
Conditional Breakpoints	63
Viewing and Changing Data	64

Viewing Local Variables	64
Viewing Global Variables	65
Putting Data in a New Window	66
Viewing Data Types	67
Viewing Data in a Different Format	68
Viewing Data as Different Types	69
Changing the Value of a Variable	71
Using the Expression Window	72
Viewing Raw Memory	73
Viewing Memory at an Address	73
Viewing Processor Registers	75
5 Expressions	77
Expressions Overview	77
In the Expression Window	77
In the Breakpoint Window	78
In a Memory Window	79
Using Expressions	80
Special Expression Features	80
Expression Limitations	80
Example Expressions	82
Expression Syntax	84
6 Debugger Preferences	89
Debugger Preferences Overview	89
Settings & Display	89
Save settings in local “.dbg” files	90
Save breakpoints	90
Save expressions	91
In variable panes, show variable types by default	91
Sort functions by method name in browser	91
Attempt to use dynamic type of C++, Object Pascal objects and SOM objects	91
Default size for unbound arrays	91
Symbolics	92
Use temporary memory for SYM data	92

Table of Contents

Use file mapping for SYM data (PowerPC + VM only) . . .	92
At startup, prompt for SYM file if none specified	93
Always prompt for source file location if file not found . .	93
Ignore file modification dates	93
Open all class files in directory hierarchy	93
Program Control	93
Automatically launch applications when SYM file opened.	94
Confirm “Kill Process” when closing or quitting	94
Set breakpoint at program main when launching applications	95
Don’t step into runtime support code.	95
Win32 Settings	95
Runtime Settings	97
7 Debugger Menus	99
Debugger Menus Overview	99
Help menu.	99
File Menu	99
Open.	100
Close.	100
Edit filename	101
Save	101
Save As.	101
Save Settings	101
Quit	101
Edit Menu	102
Undo.	102
Cut	102
Copy.	102
Paste	102
Clear	102
Select All	102
Find	103
Find Next.	103
Find Selection	103
Preferences	103

Control Menu	103
Run	104
Stop	104
Kill.	104
Step Over	104
Step Into	105
Step Out	105
Clear All Breakpoints.	105
Break on C++ exception.	106
Data Menu.	106
Show Types	106
Expand.	106
Collapse All.	106
New Expression	106
Open Variable Window	107
Open Array Window	107
Copy to Expression	107
View As	107
View Memory As	108
View Memory	109
Default	109
Signed Decimal	109
Unsigned Decimal	109
Hexadecimal	109
Character	109
C String	109
Pascal String	110
Floating Point	110
Enumeration	110
Fixed.	110
Fract	110
Window Menu	111
Show/Hide Processes	111
Show/Hide Expressions	111
Show/Hide Breakpoints	111
Close All Variable Windows.	111

Table of Contents

Show/Hide Registers.	112
Other Window Menu Items	112
8 Troubleshooting	113
About Troubleshooting	113
Index	115



Introduction

Welcome to the CodeWarrior Debugger manual.

Overview of the Debugger Manual

This manual describes MW Debug, the source-level debugger of the Metrowerks CodeWarrior software development environment. The same debugger works for all supported languages (C, C++, Pascal, assembly language, and Java). This manual often refers to MW Debug as “the CodeWarrior debugger,” or simply as “the debugger.”



NOTE: The Windows-hosted Debugger manual is a work in progress. Please consult the documentation Release Notes for details and information on updates.

A debugger is software that controls program execution so that you can see what’s happening internally as your program runs. You use the debugger to find problems in your program’s execution. The debugger can execute your program one statement at a time, suspend execution when control reaches a specified point, or interrupt the program when it changes the value of a designated memory location. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, and inspect the contents of the processor’s registers.

The other topics in this chapter are:

- Read the Release Notes—important last minute information
- System Requirements—hardware and software requirements
- Installing MW Debug—putting it together

Introduction

Read the Release Notes

- Starting Points—an overview of the chapters in this manual
- Where to Learn More—other sources of information related to the CodeWarrior debugger

Read the Release Notes

Before using MW Debug, read the release notes. They contain important information about new features, bug fixes, and any late-breaking changes.

System Requirements

If you can run CodeWarrior, you can run MW Debug.

MW Debug requires a 486, Pentium™, equivalent, or better processor with 16 MB RAM and approximately 5 MB of disk space. MW Debug requires the Microsoft Windows 95 or Windows NT 4.0 operating system.

For optimum performance, we recommend that you use a computer with a Pentium™ or equivalent processor with at least 24 MB of RAM running Microsoft Windows NT 4.0.

You can use the debugger on its own to debug your programs, however, it is recommended you use the debugger in conjunction with the CodeWarrior IDE.

Installing MW Debug

There is only one version of MW Debug for all CodeWarrior compilers and platforms. The debugger is a separate application, but it meshes almost seamlessly with the rest of the CodeWarrior integrated development environment.

The CodeWarrior Installer automatically installs MW Debug and all necessary components with any installation that also includes the CodeWarrior IDE. MW Debug and the Debugger Plugins folder must be in the same directory or the debugger will not work.

It is strongly recommended that you use the CodeWarrior Installer to install the debugger to make sure you have all the required files and avoid any problems.

Starting Points

This manual contains the following chapters:

- Getting Started Overview—how to install and run the debugger, and what SYM files are
- What You See Overview—the visual components of the debugger, all the windows and displays you encounter
- Basic Debugging Overview—the principal features of the debugger and how to use them
- Expressions Overview—how to use expressions in the debugger
- Debugger Menus Overview—a reference to the menu commands in the debugger
- About Troubleshooting—frequently encountered problems and how to solve them

If you are new to the CodeWarrior debugger, have questions about the installation process, or do not know what a SYM file is, start reading “Getting Started Overview” on page 13. To become familiar with the debugger interface, see “What You See Overview” on page 19.

If you don’t know how to control program execution, set break-points, or modify variables, read “Basic Debugging Overview” on page 43, and “Expressions Overview” on page 77.

For reference on menu items in the debugger, see “Debugger Menus Overview” on page 99.

No matter what your skill level, if you have problems using the debugger, consult “About Troubleshooting” on page 113. Here you’ll find information about many commonly encountered problems and how to solve them.

Where to Learn More

If you are already comfortable with basic debugging, but want to know more about special considerations when debugging certain kinds of code, you should read one (or more) of following manuals:

- For special considerations with respect to Mac OS projects, see the *Targeting Mac OS* manual.
- For special considerations with respect to Win32/x86 projects, see the *Targeting Win32* manual.
- For specific Magic Cap debugging information, see the *Targeting Magic* manual.
- For specific Java debugging information, see the *Targeting Java* manual.



Getting Started

This chapter discusses how to install and launch MW Debug, and contains background information on SYM files. Subsequent chapters discuss how to use the debugger.

Getting Started Overview

This chapter includes the background information you need to use the debugger effectively. The topics discussed are:

- Preparing for Debugging
- Launching the Debugger
- Symbolics Files

Preparing for Debugging

To debug a source file, you must make sure both the project as a whole and the individual source files within it are set up for debugging.

Setting Up a Project for Debugging

To prepare a project for debugging, choose the **Enable Debugger** command from the Project menu in the CodeWarrior IDE. When debugging is enabled for a project, the menu item changes to **Disable Debugger**; choosing it again turns off debugging for the project and changes the menu item back to **Enable Debugger**.

The **Enable Debugger** command sets items in the project window and the preference panels to tell the compiler and linker to generate debugging information. The compiler and linker add extra instructions to your project's program and generate a symbolics file con-

Getting Started

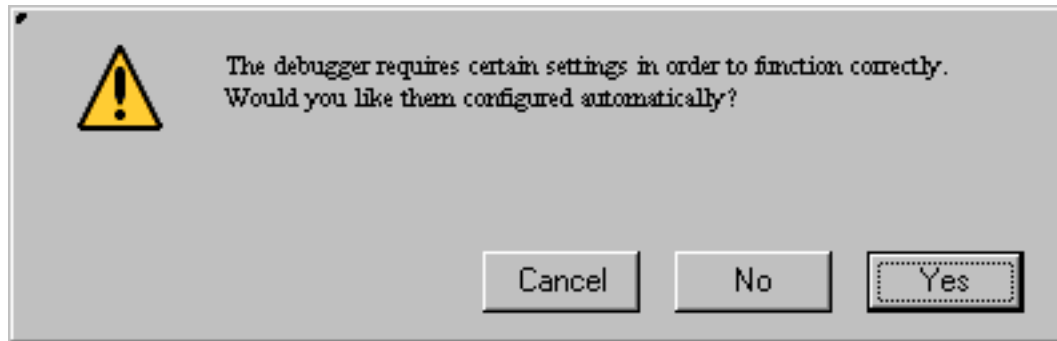
Preparing for Debugging

taining information for debugging at the source-code level. See the *CodeWarrior User's Guide* for more information on Linker and Project preferences.



NOTE: A Symbolics file allows the debugger to keep track of each function and variable name (the symbols) you use in your source code. See “Symbolics Files” on page 17 for more information.

Figure 2.1 Accepting changes set by Enable Debugger



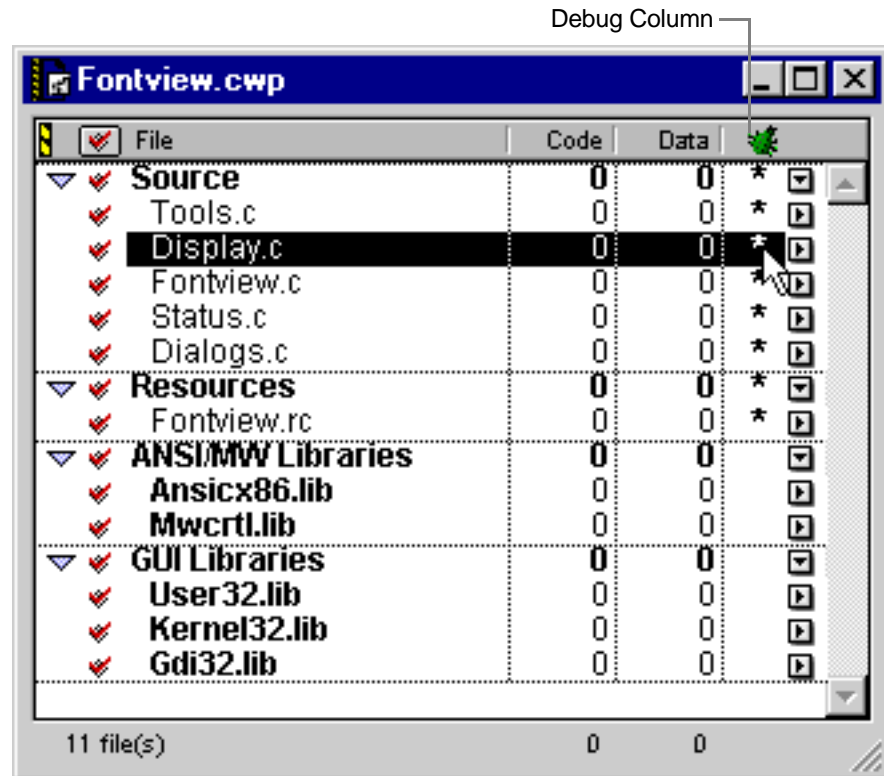
When you choose **Enable Debugger**, you may see an alert (Figure 2.1). If this alert appears, click **Yes** to apply the debugging changes to your project.

Setting Up a File for Debugging

After you have enabled debugging for the whole project, you have to set your individual files for debugging. If you intend to debug your program, you'll typically have debugging on for all of your source files.

In the CodeWarrior IDE's project window, there is a debug column, as shown in Figure 2.2. A mark in this column next to a file means that debugging is on for that file; no mark means that debugging is off. For segment names, a mark indicates that debugging is on for every file in the segment and no mark means that debugging is off for one or more files in the segment.

Figure 2.2 Setting debugging in the project window



To turn debugging on or off for a file, click in the debug column for that file. Clicking next to a segment name turns debugging on or off for all files in the segment. If a file cannot be debugged (because it is not a source file) you cannot turn debugging on for that file.

Now that you have debugging prepared for both the project and the individual files, you are ready to run your program under the debugger.

Launching the Debugger

You can launch the debugger either from within a project or separately. However you launch it, it requires a symbolics file for the project being debugged. Everything that happens in the debugger relies on the project's symbolic information. If debugging is en-

Getting Started

Launching the Debugger

abled, you can create a SYM file or CodeView information for a project by choosing **Make** from the Project menu in the CodeWarrior IDE.

See also *CodeWarrior IDE User's Guide* for more information on generating debug information.

Launching the Debugger from a Project

The MW Debug application must be in the same folder as the CodeWarrior IDE. When you enable debugging, the **Debug** command becomes enabled. This command compiles and links your project, then launches it through the debugger.

If the debugger is currently disabled, choose **Enable Debugger** before choosing the **Debug** command. In addition, if your project requires another application to be running, you must launch that application yourself: for example, if you are debugging an application plug-in, you must launch the application directly; CodeWarrior will not launch it for you.

For some kinds of projects, such as dynamic link libraries (DLL's), the **Debug** command is always disabled. To debug these projects you must launch the debugger with an application that loads the DLL either explicitly or implicitly.

See also *Targeting Win32* for more information on debugging programs other than applications.

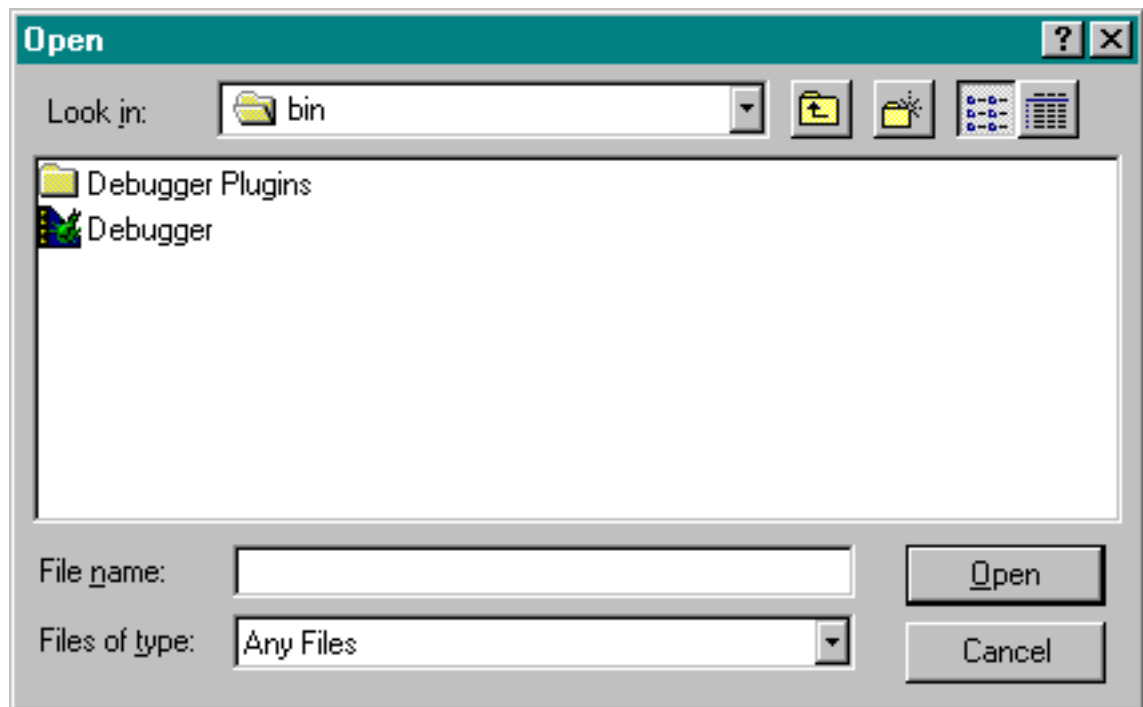
Launching the Debugger Directly

Because the debugger is a separate application, you can launch it directly just like any other application. As always, you must supply a SYM file or CodeView information for the debugger to work with. You can launch the debugger in any of three ways:

- Double-click a SYM file.
- Double-click the MW Debug icon. You'll see the standard Open File dialog box (Figure 2.3) allowing you to select a SYM or CodeView file to open.

- Drag and drop a SYM or CodeView file onto the MW Debug icon.

Figure 2.3 **Open SYM file**



Symbolics Files

A project's symbolics file contains information the debugger needs in order to debug the project, such as the names of routines and variables (the symbols), their locations within the source code, and their locations within the object code.

The debugger uses this information to present your program in the form of statements and variables instead of assembly-language instructions and memory addresses. If you wish, however, the debugger gives you the option of viewing your source code in assembly language instead. See "Viewing source code as assembly" on page 24.

Getting Started

Symbolics Files

There are two primary symbolics files that can be generated by the CodeWarrior IDE: SYM file and CodeView symbolics. CodeView is the recommended method of generating symbolic information.

See also *CodeWarrior IDE User's Guide* for more information on generating debug information. For more information on target-specific symbolic information, see the appropriate Targeting manual.

SYM File Names

CodeWarrior names the SYM file based on the name of the project, followed by a name extension. For example, if your code creates a Win32 application named Foo.exe, the debugger requires a SYM file named Foo.exe.SYM.



TIP: If you are debugging Java code, the debugger reads symbolic information directly from the class or zip file. There isn't a separate symbolics file for Java debugging.

Matching the SYM File to an Executable File

If the debugger can't find a SYM file matching the project name, it prompts you to locate the missing file. When debugging projects like dll's, you can use this feature to manually identify the correct SYM file to use when you launch the debugger. See the *Targeting Win32* manual for more information.

CodeView Symbols

CodeView symbols are slightly different than the SYM file. With CodeView symbols turned on in the x86 CodeGen and x86 Linker preference panels, the symbolic information is embedded directly in the executable or DLL you are building. Open the application or DLL file with the debugger to start your debugging session.



What You See

This chapter describes the many visual components of the MW Debug user interface.

What You See Overview

This chapter explains the various windows, panes, and displays you can use when debugging. The remaining chapters in this manual assume you are familiar with the nature and purpose of the various parts of the debugger. The topics discussed in this chapter include:

- Program Window
- Browser Window
- Expression Window
- Breakpoint Window
- Log Window
- Variable Window
- Array Window
- Memory Window
- Register Windows
- Process Window

Program Window

When the debugger opens a symbolics file, it opens two windows: the program window and the browser window. The two are similar in overall appearance, but differ in the details of what they display.

What You See

Program Window

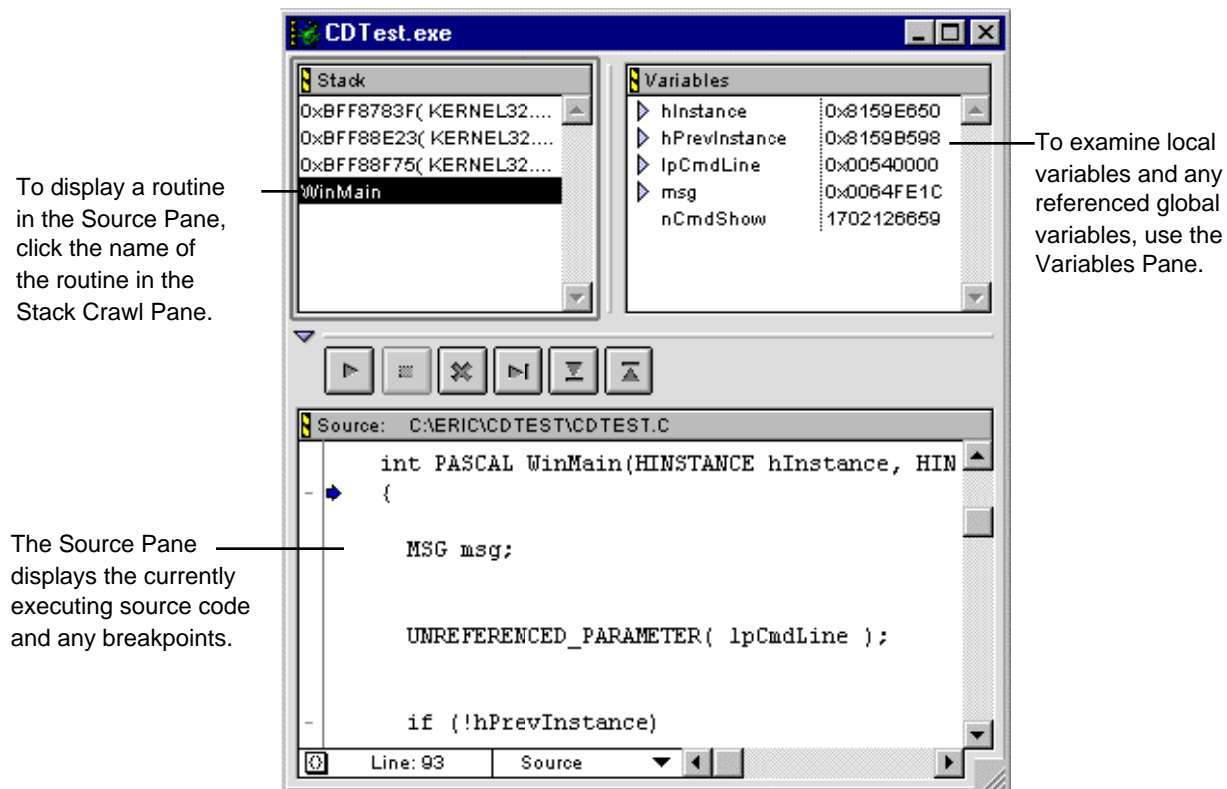
The *program window* (Figure 3.1) displays debugging information about the source-code file containing the currently running routine. This window is also called the *stack crawl window*. It has four panes:

- Stack Crawl Pane at the top left
- Variables Pane at the top right
- Debugger Tool Pane in the middle
- Source Pane at the bottom

You can resize panes by clicking and dragging the boundary between them. The active pane has a heavy border. You can switch between panes with the Tab key.

Type-ahead selection is available in the stack crawl and locals panes. You can also use the arrow keys or Tab key to navigate the items in either of these panes when it is the active pane.

Figure 3.1 Parts of the program window



There are additional controls along the very bottom of the Source pane, to the left of the horizontal scroll bar:

- the *function pop-up menu*
- the *current line number*
- the *source pop-up menu*

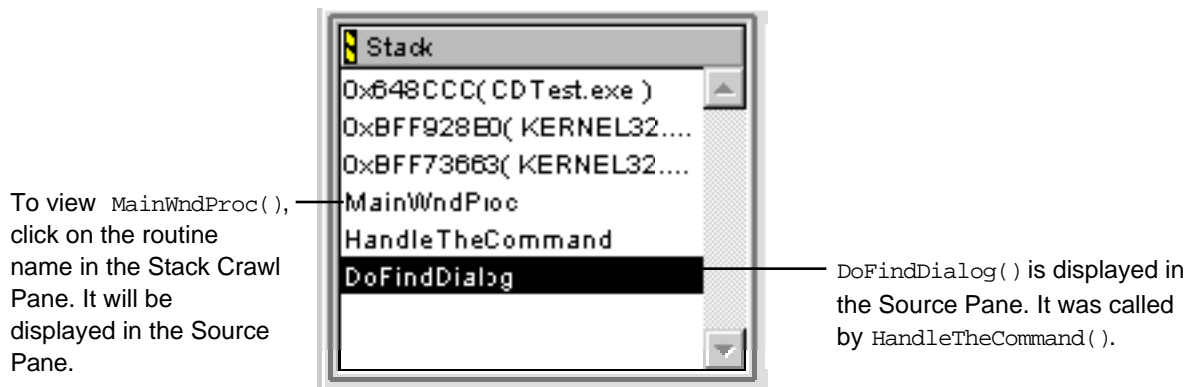
See also “Browser Window” on page 27 for details on the contents of the browser window.

Stack Crawl Pane

The stack crawl pane in the program window shows the current subroutine calling chain (Figure 3.2). Each routine is placed below the routine that called it.

The highlighted routine is displayed in the source pane at the bottom of the window. Select any routine in the stack crawl pane to display its code in the source pane.

Figure 3.2 Stack crawl pane



Variables Pane

The *variables pane* (Figure 3.3) shows a list of the currently executing routine’s local variables.

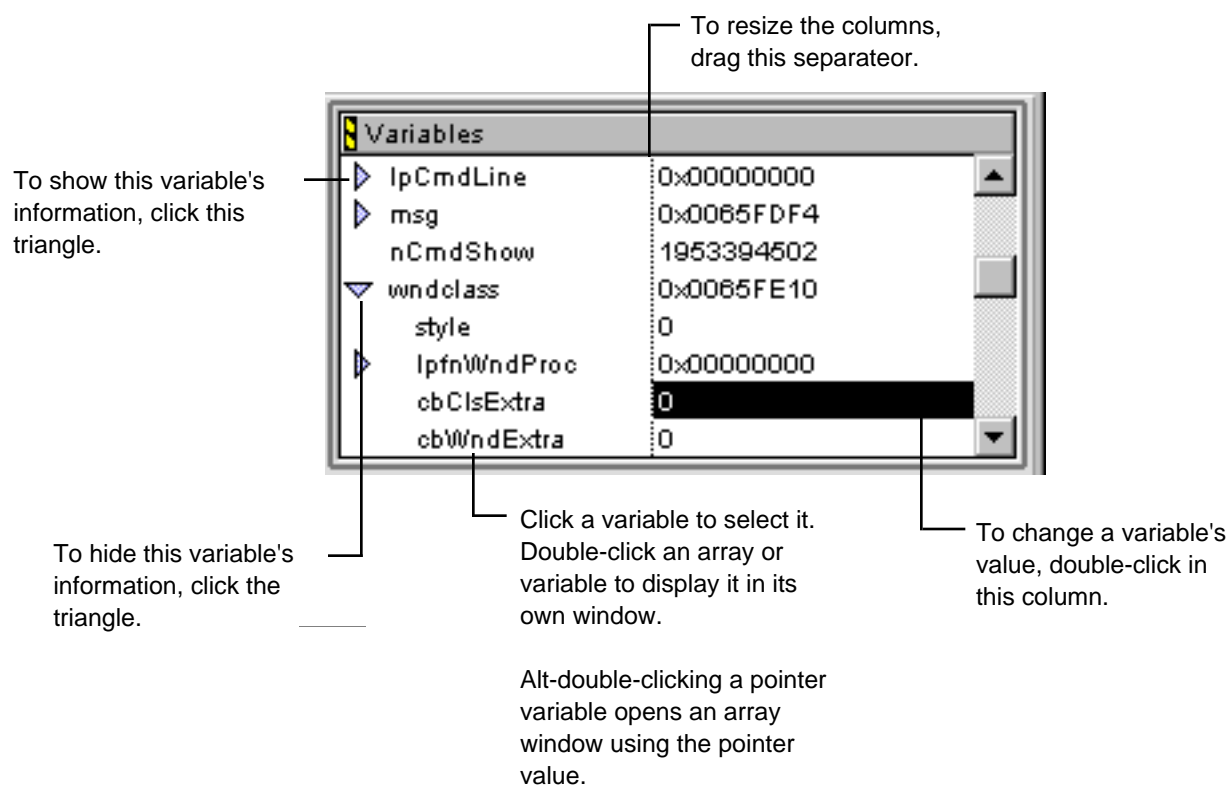
What You See

Program Window

The Variables pane lists the variables in outline form. Click the triangle next to an entry to show or hide the entries inside it. For example, in Figure 3.3, clicking the right-pointing arrow next to variable `wndclass` hides its members.

See also “Expand” on page 106 and “Collapse All” on page 106.

Figure 3.3 Variables pane

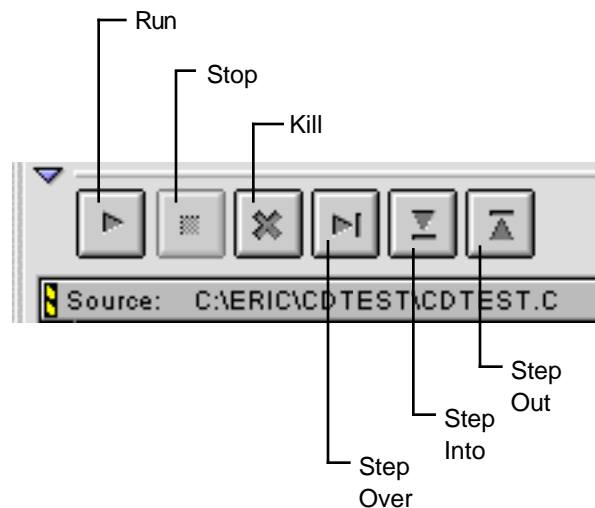


NOTE: There is no register or memory display in the Variables pane when stepping through assembly code. Instead, use the register window (“Register Windows” on page 40) to view the contents of the central-processor registers.

Debugger Tool Pane

The tool pane (Figure 3.4) contains a series of buttons that give access to the execution commands in the Control menu: **Run**, **Stop**, **Kill**, **Step Over**, **Step Into**, and **Step Out**.

Figure 3.4 Debugger tool pane



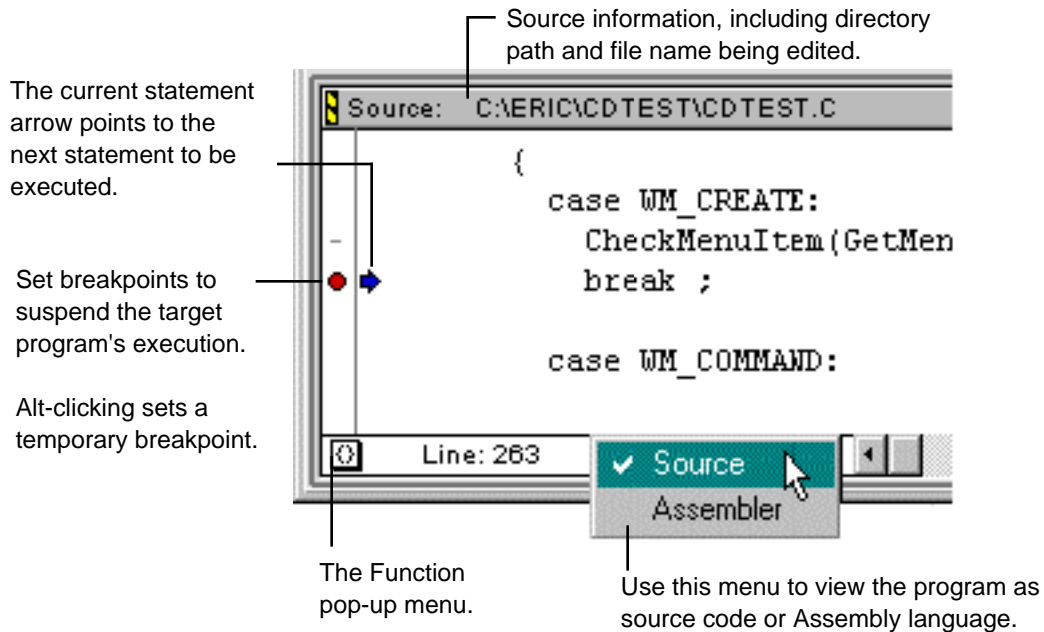
See also “Basic Debugging Overview” on page 43.

Source Pane

The *source pane* displays the contents of a source-code file. The debugger takes the source code directly from the project’s source code files, including any comments and white space. The pane shows C/C++, Pascal, Java, and in-line assembly code exactly as it appears in your program’s source code (Figure 3.5).

The source pane lets you step through the program’s source code line by line. Its progress is shown by the *current-statement arrow*, which always points to the next statement to be executed.

Figure 3.5 Source pane (program window)

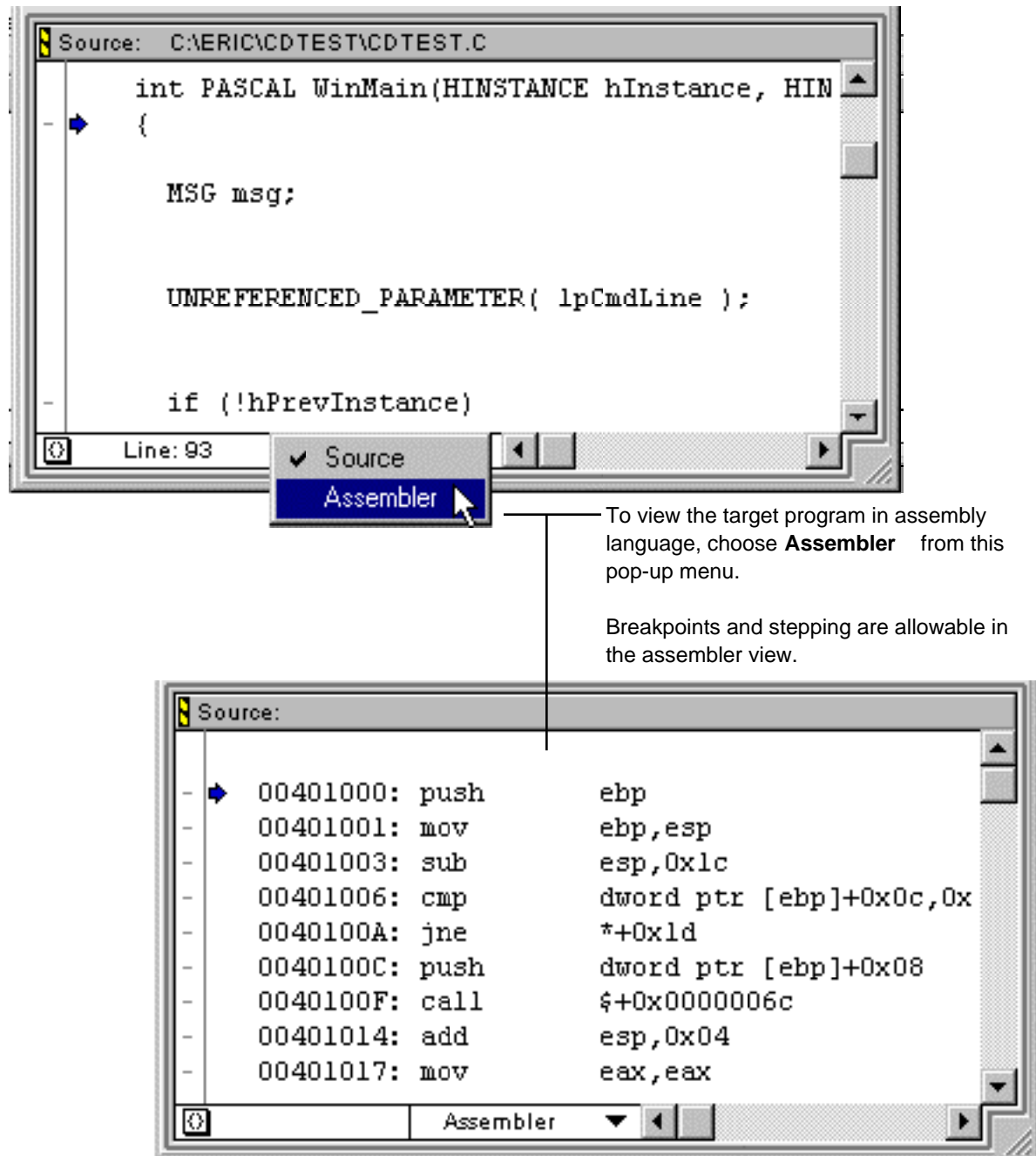


If there are two or more routine calls on a single line, each routine is executed separately as one step before the current-statement arrow moves to the next line. When this happens, the arrow is dimmed whenever the program counter is within, but not at the beginning of, a source-code line.

Viewing source code as assembly

To view your source code as assembly language, click on the *source pop-up menu* at the bottom of the program window. Choosing **Assembler** displays the contents of the source pane as assembly code (Figure 3.6). When viewing assembly code, the debugger still lets you step through the code, set breakpoints, and view variables.

Figure 3.6 Source and assembly views



What You See

Program Window

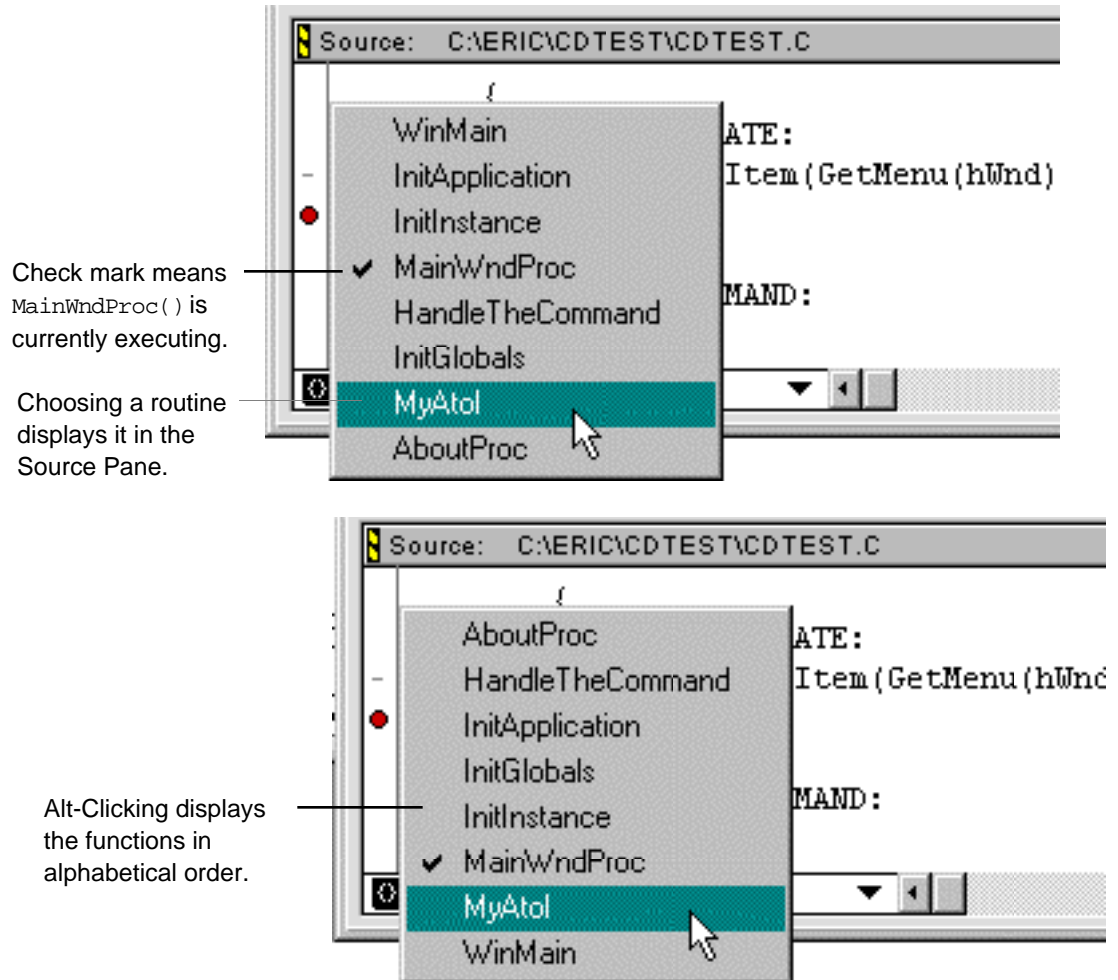


NOTE: There is no register or memory display in the Variables pane when stepping through assembly code. Instead, use the register window (See “Register Windows” on page 40) to view the contents of the central-processor registers.

Function Pop-up Menu

The *function pop-up menu*, at the bottom-left corner of the source pane, contains a list of the routines defined in the source file selected in the source pane (Figure 3.7). Selecting a routine in the function menu displays it in the source pane. Alt-click the function menu to display the menu sorted alphabetically.

Figure 3.7 Function pop-up menu



Browser Window

The *browser window* (Figure 3.8) somewhat resembles the program window in both appearance and functionality, but displays different information. The browser window lets you view any file in the project, whereas the program window can only display the file containing a currently executing routine selected from the stack crawl pane. You can also use the browser window to view or edit the values of all of your program's global variables; the program window

What You See

Browser Window

lets you change only those globals referenced by routines currently active in the call chain.



For beginners: Do not get the Browser window confused with the Class Browser available in the CodeWarrior IDE. Although the two look similar, the debugger's Browser window is a source code browser, not a class browser.

The browser window has four panes:

- *File Pane* at the top left
- *Function Pane* at the top center
- *Globals Pane* at the top right
- *Source Pane* at the bottom

Like the program window, the browser window has a function pop-up menu, a line number, and a source pop-up menu at the bottom of the window. Also like the program window, the browser window lets you resize panes by clicking and dragging the boundary between them. The active pane has a heavy border. You can switch between panes with the Tab key.

Type-ahead selection is available in the file, function, and globals panes. You can also use the arrow keys or Tab to navigate the items in any of these panes when it is the active pane.

The debugger allows more than one symbolics file to be open at a time: that is, you can debug more than one program at a time. You can use this feature, for example, to debug an application and separate plug-ins for the application.

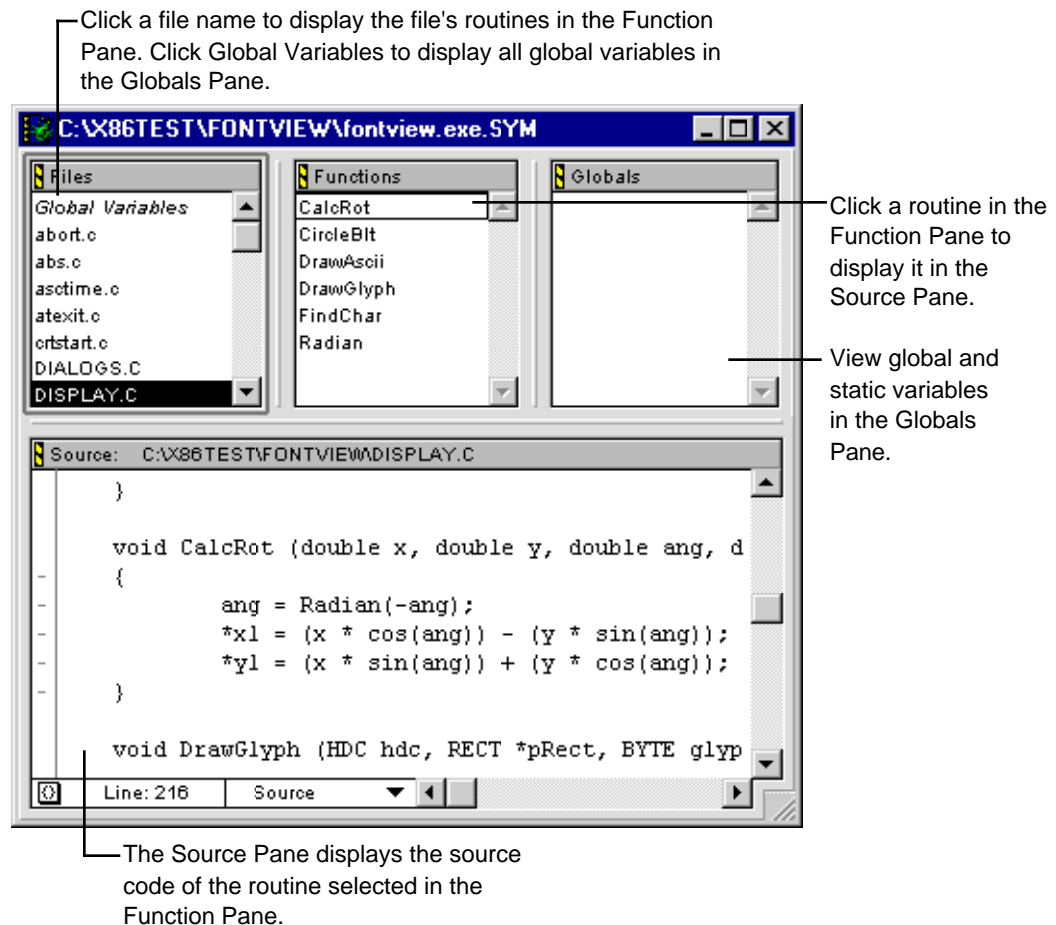
See also “Program Window” on page 19 for details on the contents of the program window.

File Pane

The *file pane* in the browser window (Figure 3.9) displays a list of all source files associated with the project you are debugging. When

you select a file name in this pane, a list of the routines defined in the file is displayed in the function pane.

Figure 3.8 Browser window



The file pane is used in conjunction with the function and source panes to set breakpoints in your program. Clicking *Global Variables* in the file pane displays all the global variables used in your program. These global variables are listed in the globals pane.

See also "Globals Pane" on page 31 and "Breakpoints" on page 60.

What You See

Browser Window

Figure 3.9 File pane

The file pane lists the source code files in the target program. The Global Variables entry, when selected, displays the program's global and static variables in the Globals Pane.



The highlighted file, in this case `Display.c`, is displayed in the Source Pane.

Function Pane

When you select a source-code file in the browser window's file pane, the *function pane* presents a list of all routines defined in that file. Clicking a routine name scrolls that routine into view in the source pane at the bottom of the window.

Figure 3.10 Function pane

The Function Pane lists the routines defined in the source code file selected in the File Pane.



When a routine is selected in the Function Pane, its code appears in the source pane.



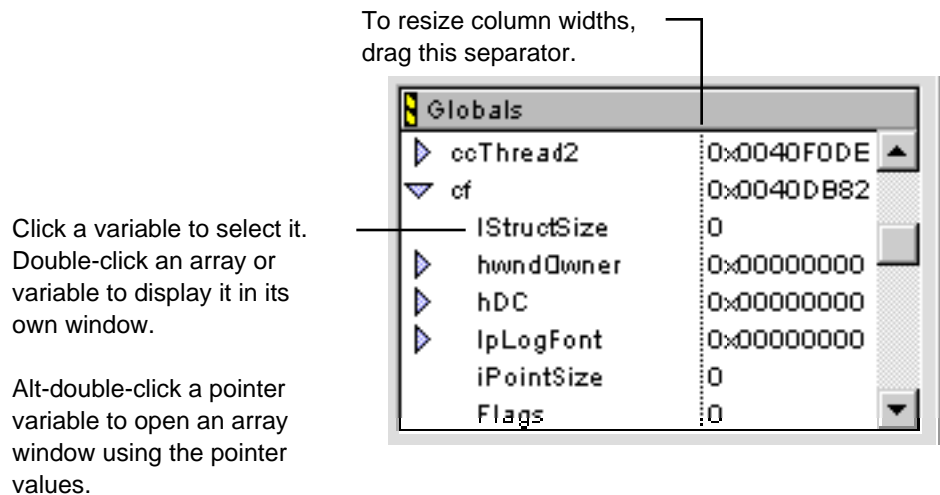
TIP: If your code is written in C++ or Object Pascal, the **Sort functions by method name in browser** option in the **Preferences** dialog (see “Settings & Display” on page 89) alphabetizes function

names of the form `className : : methodName` by method name instead of by class name. Since most C++ source files tend to contain methods all of the same class, this preference makes it easier to select methods in the function pane by typing from the keyboard.

Globals Pane

When the Global Variables item is selected in the file pane, the *globals pane* displays all global variables used by your program (Figure 3.11). You can also view static variables by selecting in the file pane the file. The static variables will also appear in the globals pane.

Figure 3.11 Globals pane



Placing globals in a separate window

To display a global variable in its own window, double-click on the variable's name in the globals pane; a new variable window will appear containing the variable's name and value. You can also open a variable window by selecting the desired variable in the globals pane and selecting the **Open Variable Window** or **Open Array Window** command from the Data menu. A global displayed in its own

What You See

Browser Window

window can be viewed and edited the same way as in the globals pane. You can also add global variables to the expression window.

See also “Variable Window” on page 36, “Array Window” on page 37, and “Expression Window” on page 33.

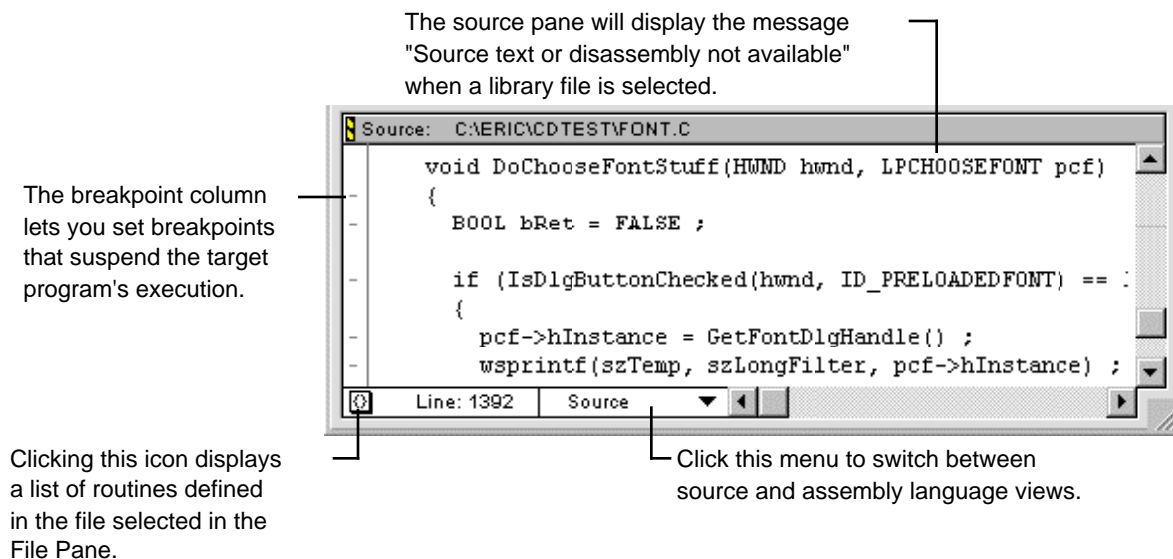
Windows containing global variables remain open for the duration of the debugging session. To close a global variable or global array window, click its close box or select the **Close All Variable Windows** command from the Window menu.

See also “Close All Variable Windows” on page 111.

Source Pane

The *source pane* in the browser window shows the contents of the source-code file selected in the file pane (Figure 3.12). You can use it to set breakpoints in any file listed in the file pane. Notice, however, that the browser window’s source pane does not show the currently executing statement; to view the current statement or local variables, use the program window instead.

Figure 3.12 Source pane (Browser window)



If the item selected in the file pane does not contain source code, the source pane displays the message “Source text or disassembly not available.”

The browser window has a *source pop-up menu* at the bottom like the one in the program window (see “Viewing source code as assembly” on page 24). Selecting **Assembler** displays the contents of the source pane as assembly code, as shown earlier in Figure 3.6. You can set breakpoints in assembly code, just as you can in source code.

See also “Breakpoints” on page 60.

Function Pop-up Menu

The *function pop-up menu*, at the bottom-left corner of the source pane, contains a list of the routines defined in the source file selected in the file pane. Selecting a routine in the function menu displays it in the source pane, just as if you had clicked the same routine in the function pane. Alt-clicking the function menu presents an alphabetically sorted menu like the menu shown earlier in Figure 3.7.

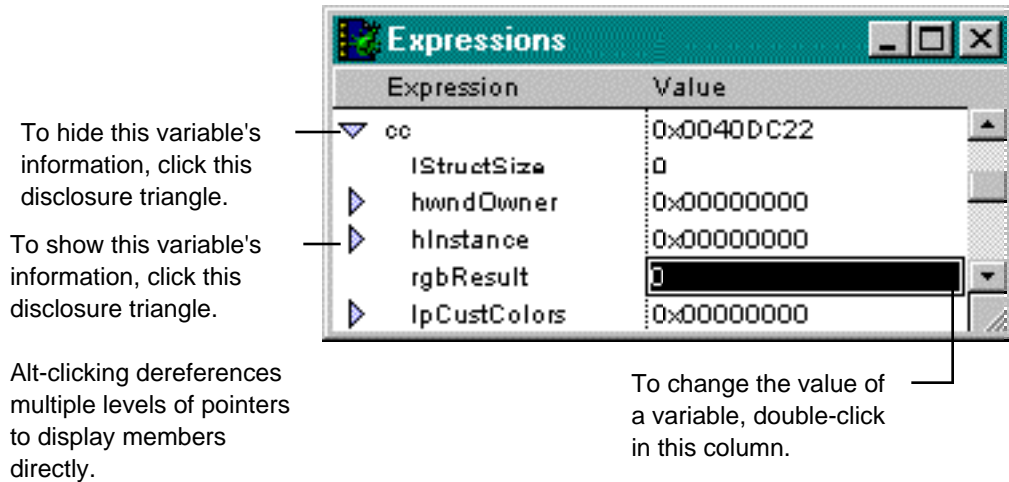


NOTE: The function pop-up menu does nothing if there is no source code displayed in the source pane.

Expression Window

The *expression window* (Figure 3.13) provides a single place to put frequently used local and global variables, structure members, and array elements without opening and manipulating a lot of windows.

Figure 3.13 Expression window



To open the expression window, choose **Show Expressions** from the Window menu. When the expression window is open, the command changes to **Hide Expressions** and is used to close the window. (You can, of course, also close the expression window by simply clicking its close box.)

Use the **Copy to Expression** command in the Data menu to add selected items to the expression window. You can also use the mouse to drag items from other variable panes and windows into the expression window, or to reorder the items in the expression window by dragging an item to a new position in the list.

To remove an item from the expression window, select the item and press the Backspace key or choose **Clear** from the Edit menu.

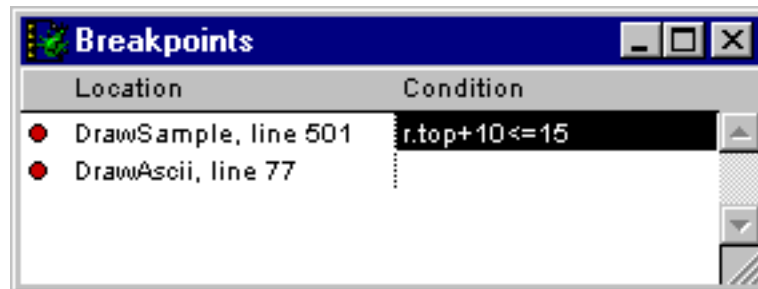
Unlike local variables displayed in an ordinary variable window, those in the expression window are not removed on exit from the routines in which they are defined.

See also “Show/Hide Expressions” on page 111, “Copy to Expression” on page 107, and “Using the Expression Window” on page 72.

Breakpoint Window

The *breakpoint window* (Figure 3.14) lists all breakpoints in your project, by source file and line number. To open the breakpoint window, choose **Show Breakpoints** from the Window menu. When the breakpoint window is open, the command changes to **Hide Breakpoints** and is used to close the window. (You can, of course, also close the breakpoint window by simply clicking its close box.)

Figure 3.14 Breakpoint window



There is a breakpoint marker to the left of each listing. A circle indicates that the breakpoint is active, a dash that it is inactive. Clicking a breakpoint marker toggles the breakpoint on or off while remembering its position in the target program. Double-clicking a breakpoint listing activates the browser window, with its source pane displaying that line of code.

Each breakpoint can have an attached condition. If the condition is true and the breakpoint is set, the breakpoint stops program execution. If the breakpoint is clear or the condition is false, the breakpoint has no effect.

See also “Breakpoints” on page 60, “Show/Hide Breakpoints” on page 111, and “Conditional Breakpoints” on page 63.

Log Window

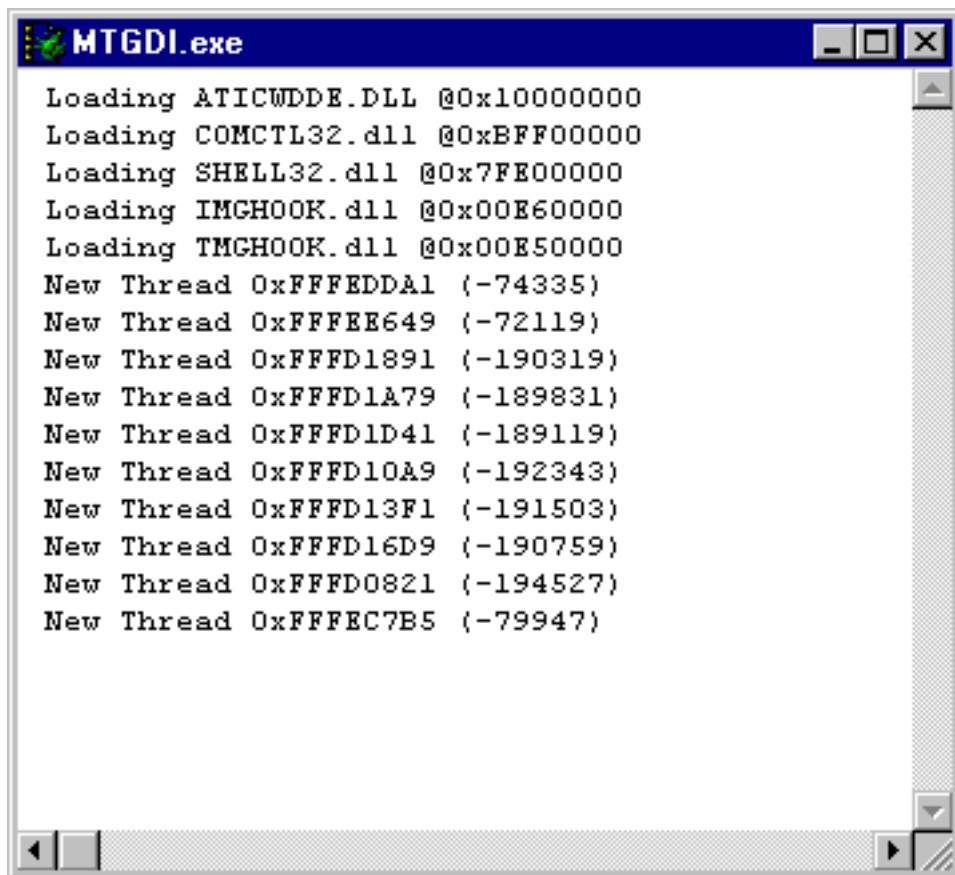
The *Log window* (Figure 3.15) displays messages as your program makes calls to system DLL's or starts new threads.

What You See

Variable Window

You can directly edit the contents of the log window. This allows you to make notes as your program runs. You can also copy text from it with the **Copy** command in the Edit menu, or use the **Save** or **Save As** command in the File menu to save its contents to a text file for later analysis.

Figure 3.15 Log window



Variable Window

A *variable window* (Figure 3.16) displays a single variable and allows its contents to be edited. A variable window containing a local variable will close on exit from the routine in which the variable is defined.

Figure 3.16 A variable window



Array Window

An *array window* (Figure 3.17) displays a contiguous block of memory as an array of elements and allows the contents of the array elements to be edited. To open the array window, select an array variable in a variable pane (either locals or globals) and then choose **Open Array Window** from the Data menu. To close the array window, click its close box.

You can also use the **View Memory as** command in the Data menu to open an array window. This command presents a dialog box in which you can select a data type, then opens an array window interpreting memory as an array of that type.

An array window's title bar describes the base address the array is bound to. An array's base address can be bound to an address, a variable, or a register. Dragging a register name or variable name from a variable or register pane to an array window sets the array address. An array bound to a local variable will close when the variable's routine returns to its caller.

The information pane displays the data type of the array elements, along with the array's base address. Clicking the arrow in the information pane shows more information about the array. From the expanded information pane, you can select the array's base address, its size, and which members to view if the array elements are of a structured type.

The array's contents are listed sequentially, starting at element 0. If array elements are cast as structured types, an arrow appears to the

What You See

Array Window

left of each array element, allowing you to expand or collapse the element.

See also “Open Array Window” on page 107 and “View Memory As” on page 108.

Figure 3.17 Anatomy of an array window

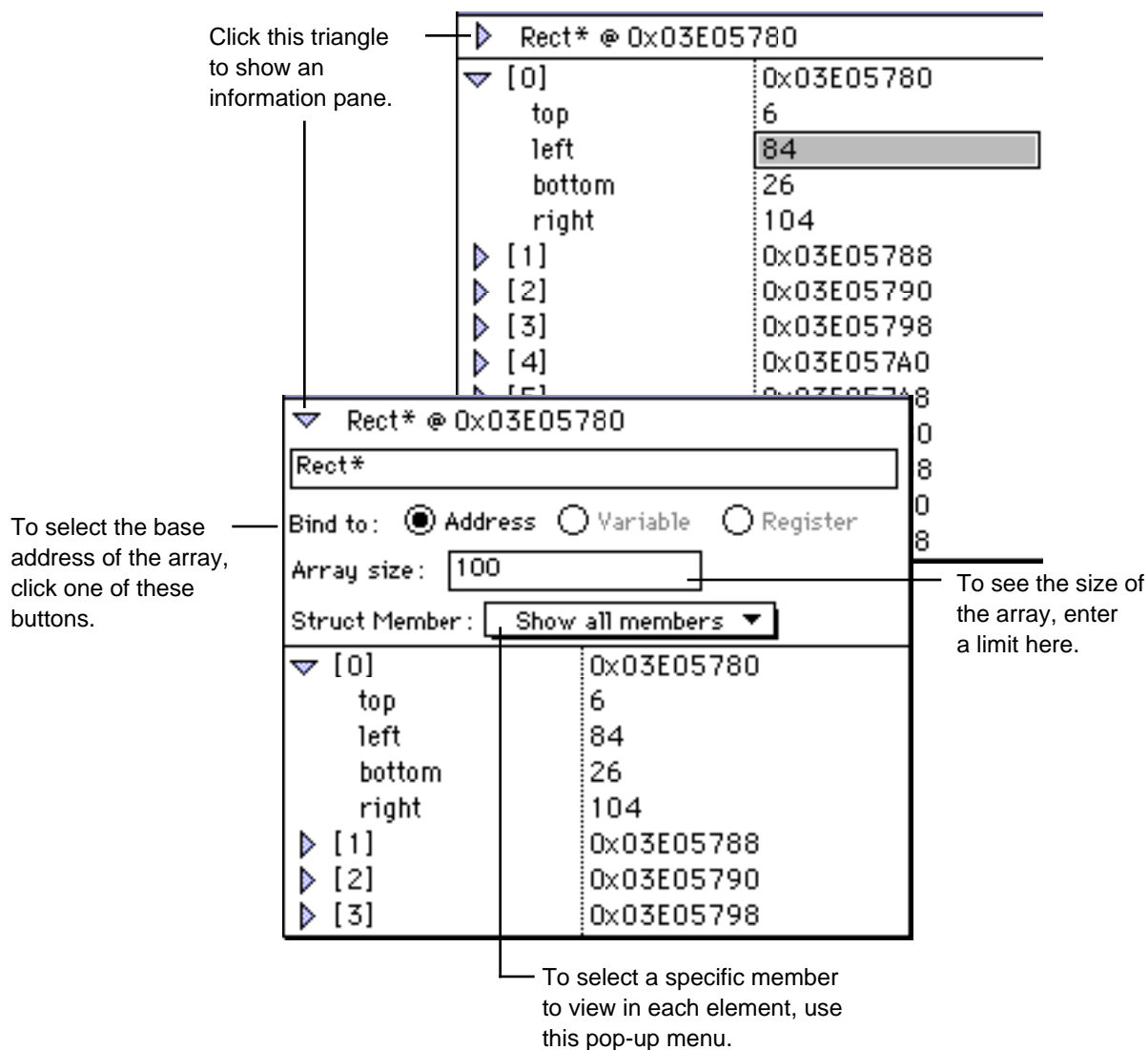


Figure 3.18 A memory window



The source of the base address (which may be a variable, a routine, any expression, or a raw address like 0xCAF64C) is displayed at the top of the window. A memory window is blank if the base address can't be accessed.

What You See

Register Windows

To change the base address, simply type or drag a new expression to the expression field. If the expression does not produce an lvalue, then the value of the expression is used as the base address. For example, the memory-window expression

`PlayerRecord`

will show memory beginning at the address of `PlayerRecord`.

If the expression's result is an object in memory (an lvalue), then the address of the object is used as the base address. For example, the expression

`*myArrayPtr`

will show memory beginning at the address of the object pointed to by `myArrayPtr`.

You can use a memory window to change the values of individual bytes in memory. Simply click in the displayed data to select a starting point, and start typing. If you select a byte in the hexadecimal display, you are restricted to typing hexadecimal digits. If you select a byte in the ASCII display, you can type any alphanumeric character. Certain keys (such as backspace, Tab, Enter, and so forth) do not work. New data you type overwrites what is already in memory.



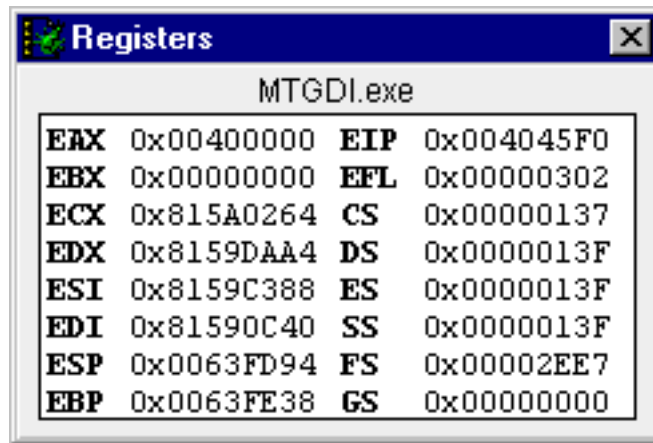
WARNING! Arbitrarily changing the contents of memory can be very dangerous to your computer's stability and can result in a crash. Make sure you know what you're doing, and don't change anything critical.

Register Windows

A *register window* displays CPU registers (Figure 3.19) and allows their contents to be edited. To open a register window, choose **Show Registers** from the Window menu. When a register window is open, the corresponding command changes to **Hide Registers** and is used to close the window. (You can, of course, also close the register window by simply clicking its close box.)

To change a registers value, double-click the registers value or select the register and press Enter. You can then type in a new value.

Figure 3.19 A CPU register window



Double-click a registers value to change it.



WARNING! Changing the value of a register is a very dangerous thing to do. It could corrupt your data, memory, or cause a crash.

See also “Show/Hide Registers” on page 112.

Process Window

The *Process window* (Figure 3.20) lists processes currently running under the Windows Task Manager, including some hidden processes. The process window also lists threads for the selected process. To open the process window, choose **Show Processes** from the Window menu. When the process window is open, the command changes to **Hide Processes** and is used to close the window. (You can, of course, also close the process window by simply clicking its close box.)



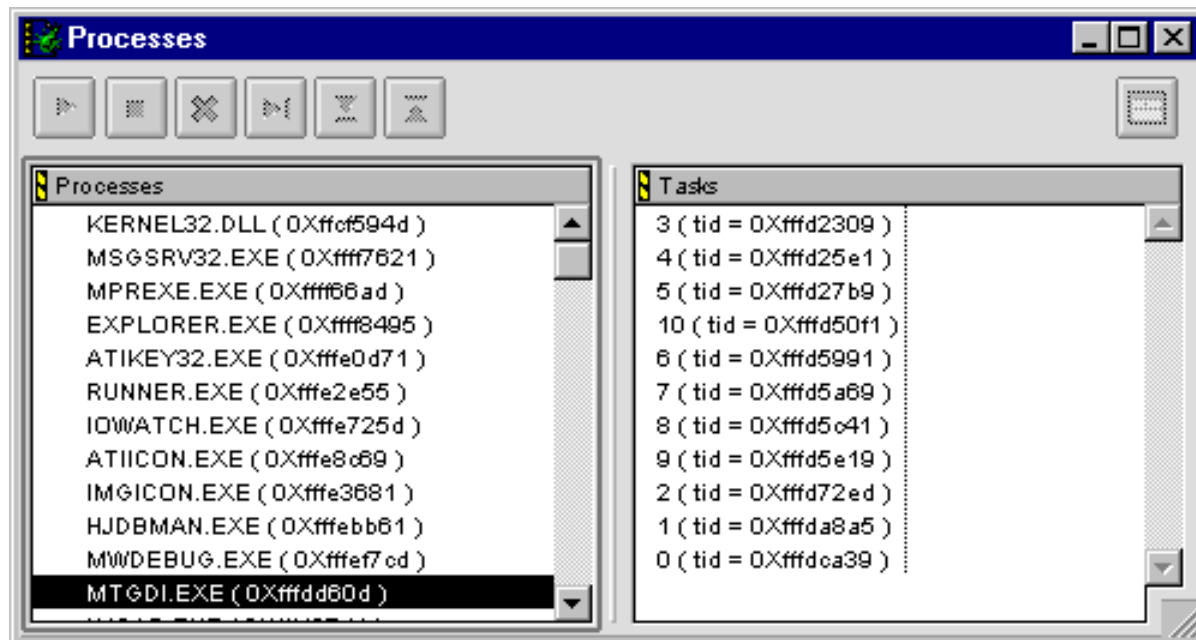
NOTE: Documentation for the Process window is incomplete because the tool is still under development at the time of this writing.

What You See

Process Window

Check the release notes for the latest information on the Process window and its functionality.

Figure 3.20 Process window





Basic Debugging

This chapter introduces you to the principles of debugging.

Basic Debugging Overview

A debugger is software that controls the execution of a program so that you can see what's happening internally and identify problems. This chapter discusses how to use MW Debug to locate and solve problems in your source code by controlling program execution and viewing your data and variables. The principal topics discussed are:

- Starting Up—things to watch out for when starting the debugger
- Running, Stepping, and Stopping Code—controlling program execution a line at a time
- Navigating Code—moving around and finding the code you want in the debugger
- Breakpoints—stopping execution when and where you want
- Viewing and Changing Data—seeing your variables and modifying them at will

To learn how to prepare a project for debugging or launch the debugger, see “Getting Started Overview” on page 13. This chapter also assumes you are familiar with the information about the debugger interface found in “What You See Overview” on page 19. For information on how to set the debugger's preferences, see “Preferences” on page 103.

Starting Up

When the debugger launches, two windows are opened: the browser window and the program window. You should pay attention to what happens.

If the debugger is not running and you launch it from a project or directly from a symbolics file, when the debugger appears the program window is the active window. If that's the case, all is well. Your program's code appears in the program window, stopped at the first line and ready to run.

If the debugger is already running and you launch it directly from a project, when the debugger appears *the browser window may be the active window*. In this case, you must issue a second **Run** command from inside the debugger. This launches the project under debugger control, brings the program window to the foreground, and stops the program at the first line.

See also “Launching the Debugger from a Project” on page 16 and “Launching the Debugger Directly” on page 16.

Running, Stepping, and Stopping Code







This section discusses how to control program execution: that is, how to run your code, move through it line by line, and stop or kill the project when you want to stop debugging.

Moving through code line by line is often called “walking” through your code. It is a linear approach to navigating, where you start at the beginning and move steadily through the code. This is important for understanding how to navigate in your code—but the real power comes in the next sections, which discuss how to navigate to any location directly, how to stop your code at specific locations when certain conditions are met, and how to view and change your data.

There are a few ways to walk through your code. You can use the toolbar buttons, keyboard, or choose the appropriate command

from the Control menu. Table 4.1 lists the buttons and their keyboard equivalents.

Table 4.1 **Button and Key commands**

Button	Meaning	Keyboard Equivalent
	Run	CTRL-R or F5
	Stop	CTRL-. (period) or SHIFT-F5
	Kill	CTRL-K
	Step Over	CTRL-S or F10
	Step Into	CTRL-T or F11
	Step Out	CTRL-U or SHIFT-F11

This section discusses:

- The Current-Statement Arrow
- Running Your Code
- Stepping a Single Line
- Stepping Into Routines
- Stepping Out of Routines
- Skipping Statements
- Stopping Execution
- Killing Execution

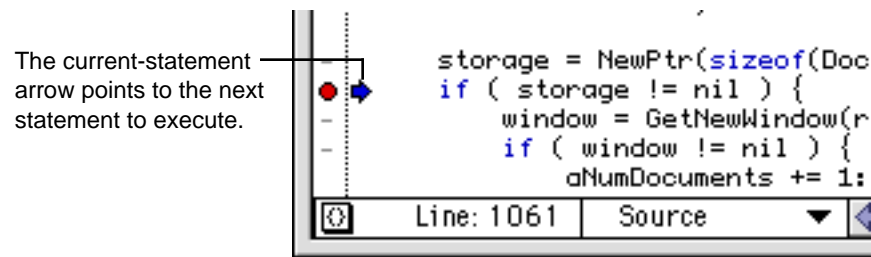
Basic Debugging

Running, Stepping, and Stopping Code

The Current-Statement Arrow

The *current-statement arrow* in the program window (Figure 4.1) indicates the next statement to be executed. It represents the processor's program-counter register. If you have just launched the debugger, it will point to the first line of executable code in your program.

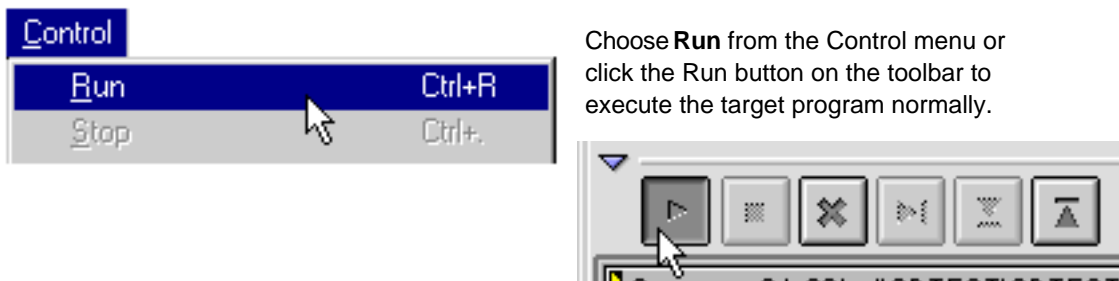
Figure 4.1 The current-statement arrow



Running Your Code

If the project has been launched but execution has been stopped, use the **Run** command (Figure 4.2) to restart your program. When you do, the program resumes execution at the current-statement arrow.

Figure 4.2 The Run command



TIP: If the project has been launched, you'll see source code in the source pane of the program window. If it has not been launched, the program window says "Program name is not running." In that case, the **Run** command launches your project under

control of the debugger and brings the program window forward with execution stopped at the first line of code.

After a breakpoint or a **Stop** command, the debugger regains control and the program window appears showing the current-statement arrow and the current values of local and global variables. The debugger places an implicit breakpoint at the program's main entry point and stops there (Figure 4.3). Issuing another **Run** command resumes program execution from the point of the interruption. After a **Kill** command, **Run** restarts the program from its beginning.

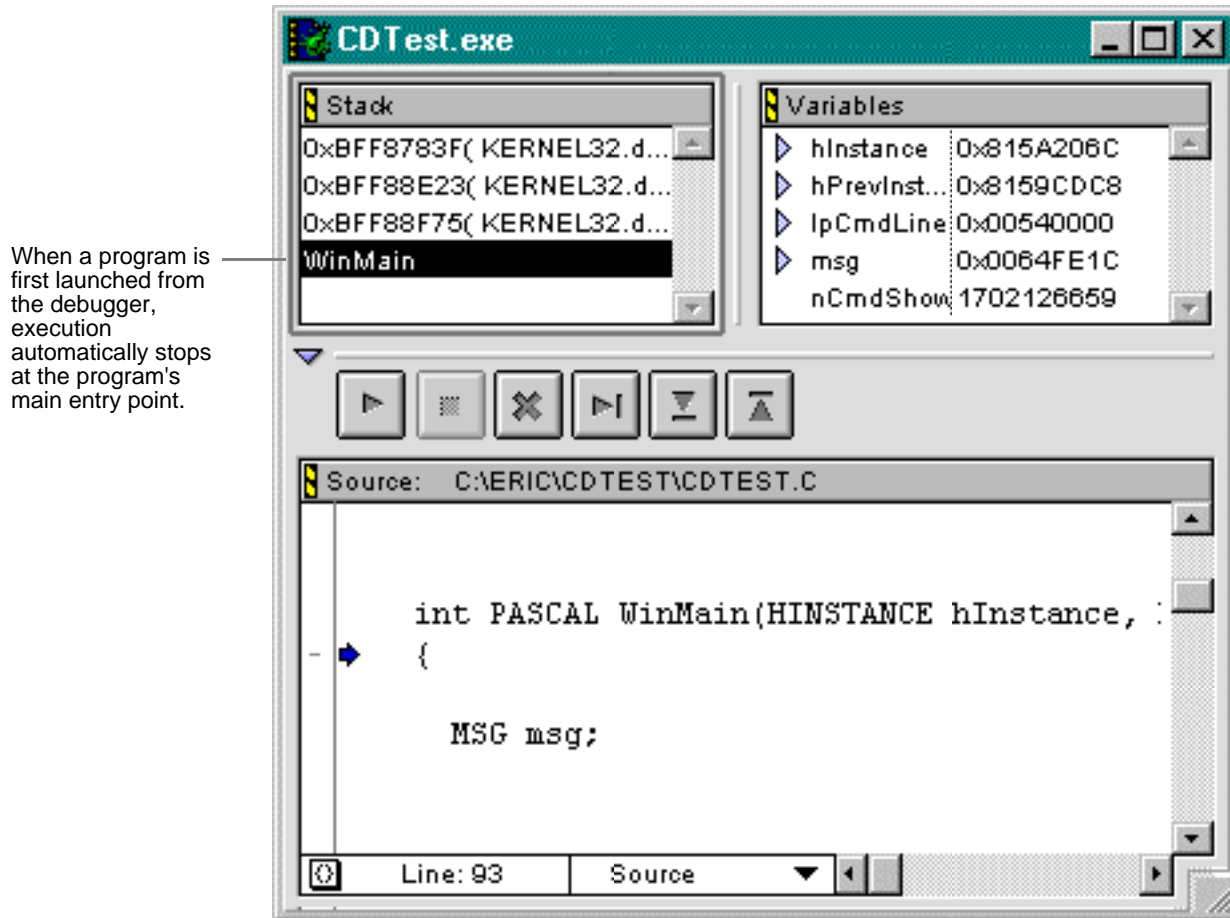


TIP: You can inhibit the automatic launch of the program by holding down the Alt key while opening the symbolics file. You can also change the **Automatically launch applications when SYM file opened** preference (see “Program Control” on page 93). One use for this feature is to debug C++ static constructors, which are executed before entering the program's main routine.

Basic Debugging

Running, Stepping, and Stopping Code

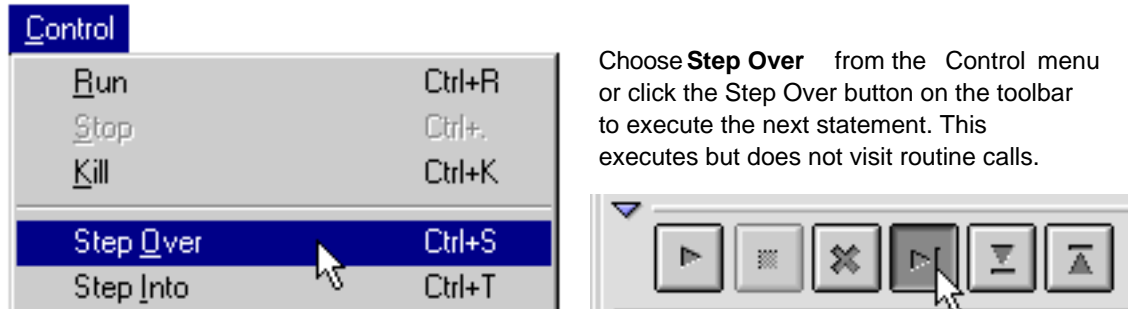
Figure 4.3 Starting the execution of the target program



Stepping a Single Line

To execute one statement, use the **Step Over** command (Figure 4.4). If that statement is a routine call, the entire called routine executes and the current-statement arrow proceeds to the next line of code. The contents of the called routine are stepped over; the routine runs, but it does not appear in the debugger's program window. In other words, the **Step Over** command executes a routine call without visiting the code in the called routine. When you are stepping over code and reach the end of a routine, the current statement arrow returns to the routine's caller.

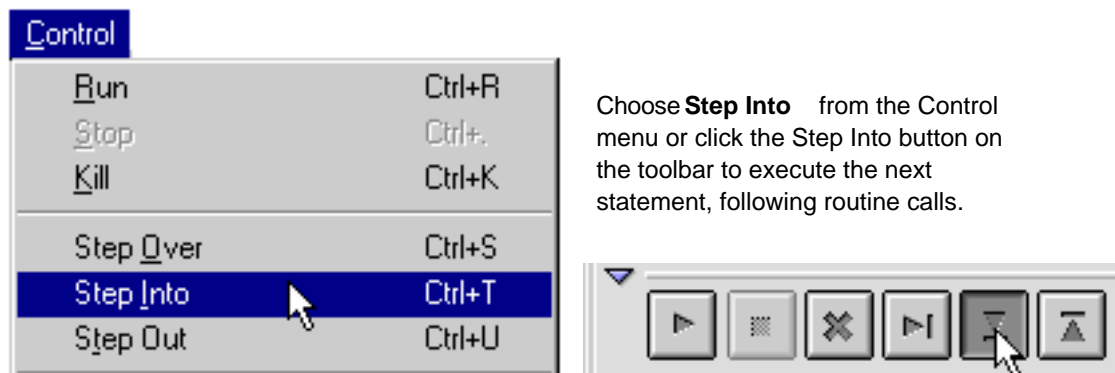
Figure 4.4 The Step Over command



Stepping Into Routines

Sometimes you want to follow execution into a called routine. This is known as *tracing* code (hence the Ctrl-T key equivalent). To execute one statement at a time and follow execution into a routine call, use the **Step Into** command (Figure 4.5).

Figure 4.5 The Step Into command

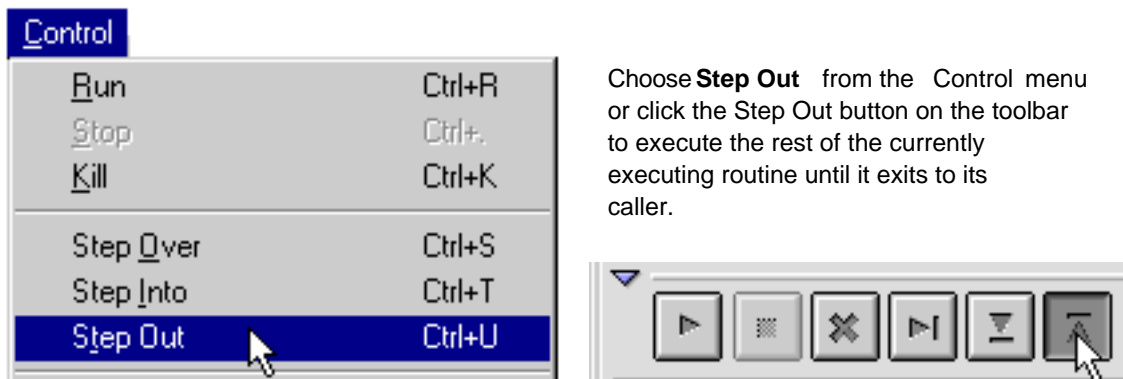


Step Into moves the current-statement arrow down one statement, unless the current statement contains a routine call. When **Step Into** reaches a routine call, it follows execution into the routine being called.

Stepping Out of Routines

To execute statements until the current routine returns to its caller, use the **Step Out** command (Figure 4.6). **Step Out** executes the rest of the current routine normally and stops the program when the routine returns to its caller. You are going one level back *up* the calling chain, so Ctrl-U is the key equivalent. See “Call-Chain Navigation” on page 54.

Figure 4.6 The Step Out command



Skipping Statements

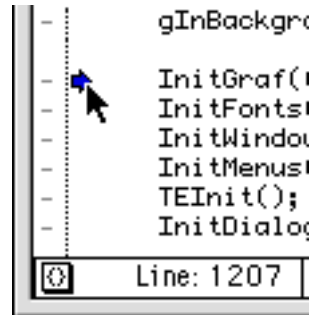
Sometimes you may want to skip statements altogether: that is, not execute them at all. To move the current-statement arrow to a different part of the currently executing source-code file, simply drag it with the mouse (Figure 4.7). Note that dragging the current-statement arrow *does not execute* the statements between the arrow's original location and the new location it is dragged to.



WARNING! Dragging the current-statement arrow is equivalent to deliberately changing the program counter in the register window. This is very dangerous, because you might corrupt the stack by skipping routine calls and returns. The debugger is not able to prevent you from corrupting your run-time environment.

Figure 4.7 Dragging the current-statement arrow

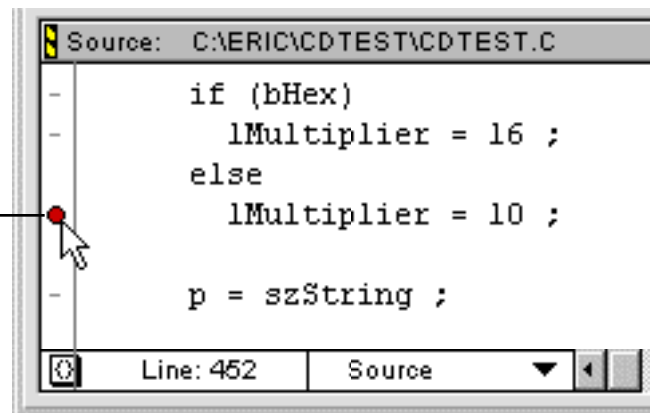
To force the current-statement arrow to another statement, simply drag it.



To move the current-statement arrow without potentially corrupting the run-time environment, Alt-click a statement in the breakpoint column (Figure 4.8). Alt-clicking sets a temporary breakpoint: execution proceeds normally until the current-statement arrow reaches the temporary breakpoint, then stops. (See “Breakpoints” on page 60.)

Figure 4.8 Setting a temporary breakpoint

To execute the program up to this point, Alt-click in the breakpoint column.

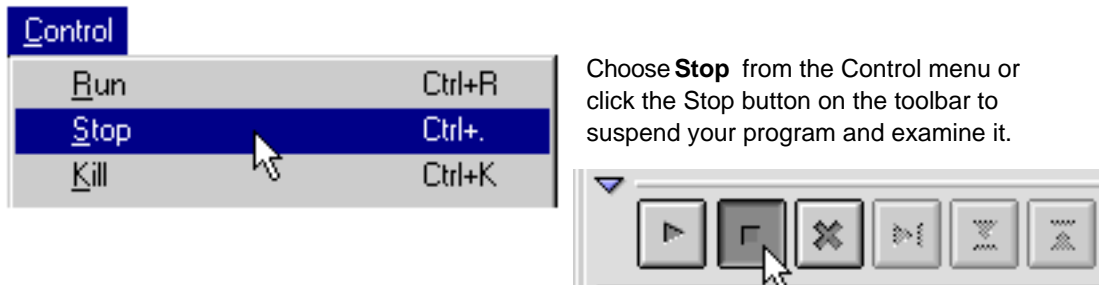


Basic Debugging

Running, Stepping, and Stopping Code

Stopping Execution

Figure 4.9 The Stop command



Stopping in this fashion is not very precise. Code executes very quickly, and there is no telling where in your code you're going to stop when you issue the **Stop** command. It's usually a better idea to use breakpoints, which allow you to stop execution precisely where you want. (See "Breakpoints" on page 60.)

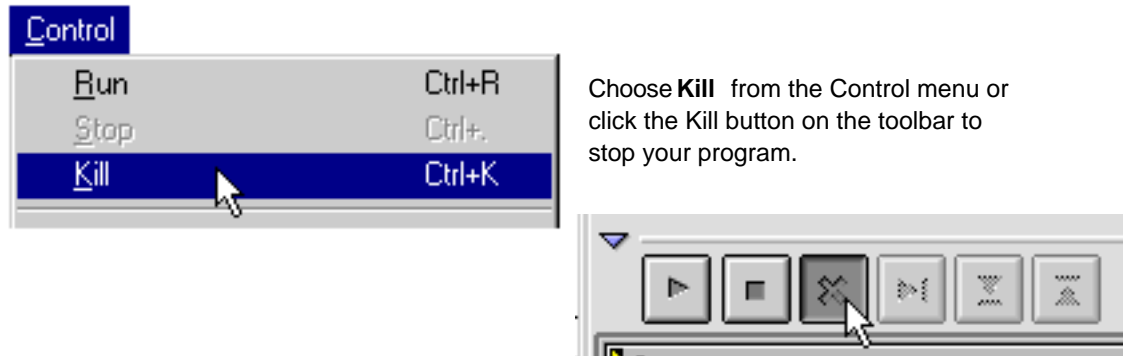


TIP: If your program hangs in an infinite loop, you can regain control by switching to MW Debug and issuing a **Stop** command.

Killing Execution

Sometimes you want to terminate your program completely—end the debugging session. The **Kill** command (Figure 4.10) ends the program and returns you to the debugger. The program window will tell you that the program is not running, and to choose **Run** from the Control menu to start it.

Figure 4.10 The Kill command



Killing the program is not the same as stopping. Stopping only suspends execution temporarily: you can resume from the point at which you stopped. Killing permanently terminates the program.

Navigating Code

This section discusses the various ways you can move around in your code. This skill is vital when you want to set breakpoints at particular locations. Methods of moving around in code include:

- Linear Navigation—stepping through code
- Call-Chain Navigation—moving to active routines
- Browser Window Navigation—moving to code in the browser window
- Using the Find Dialog—finding occurrences of specific definitions, variables, or routine calls

Linear Navigation

You can “walk” through your code by using the **Step Over**, **Step Into**, and **Step Out** commands as needed until you reach the place you want. This is useful for short stretches of code, but not very helpful when you want to get to a specific location a distance away.

See also “Stepping a Single Line” on page 48, “Stepping Into Routines” on page 49, and “Stepping Out of Routines” on page 50.

Call-Chain Navigation

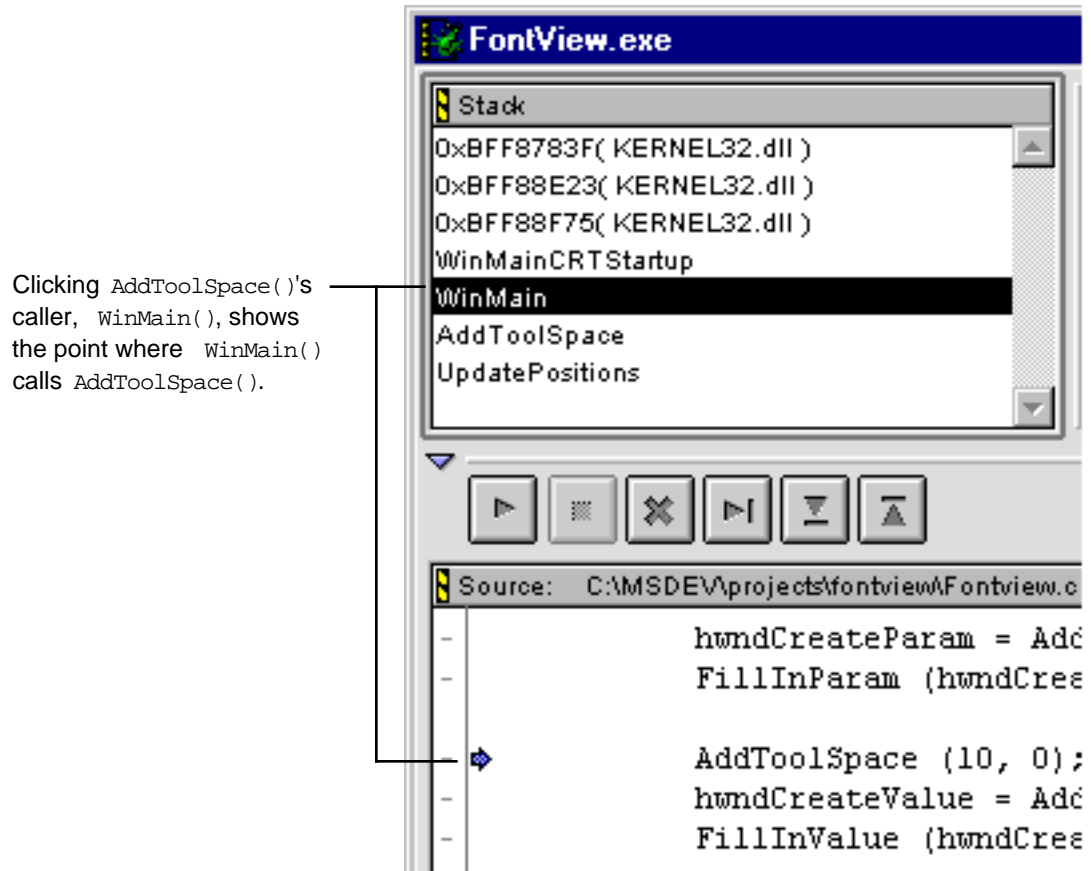
The chain of routine calls is displayed in the stack crawl pane of the program window (Figure 4.11). Each routine in the chain appears below its caller, so the currently executing routine appears at the bottom of the chain and the first routine to execute in the program is at the top.

Figure 4.11 The stack crawl pane



You can use the stack crawl pane to navigate to the routines that called the currently executing routine. To find where a routine in the stack crawl pane is called from, click the name of its caller. This displays the source code for the caller right at the point of call (Figure 4.12).

Figure 4.12 Finding a routine's point of call

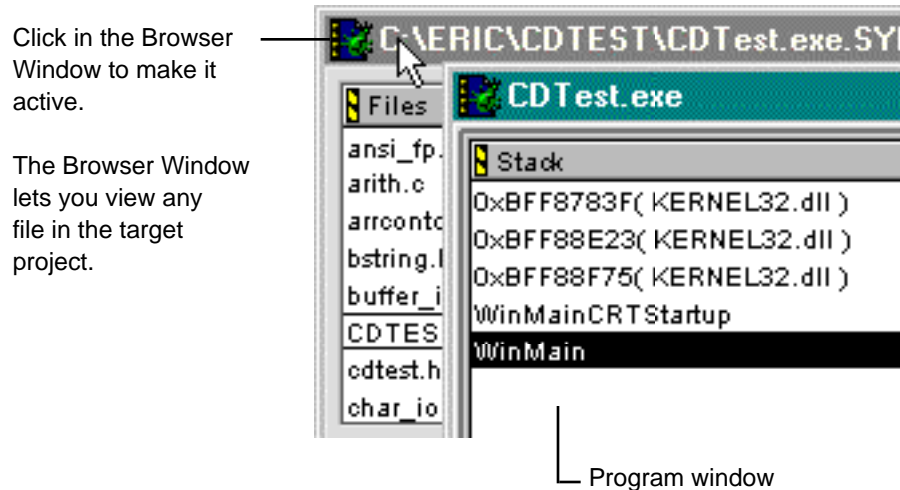


Browser Window Navigation

You can use the browser window to jump to any location in your source code. To view a specific routine:

1. **Make the browser window active (Figure 4.13).**

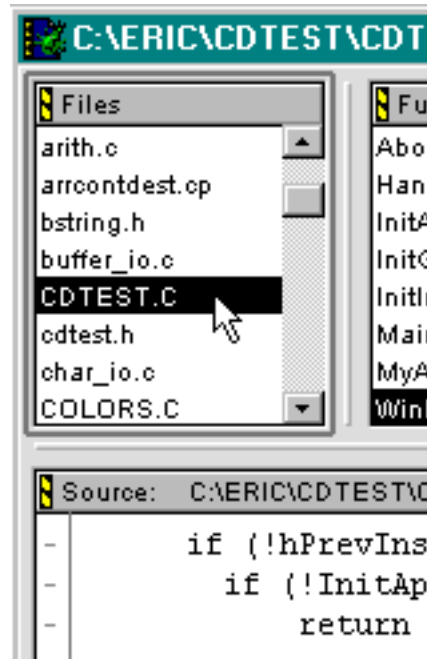
Figure 4.13 Activating the browser window



2. **In the browser window's file pane, select the file where the routine is defined (Figure 4.14).**

Simply click the desired file, or use the arrow keys to scroll through the list. The source code for that file appears in the source pane. You can also type the name of the file.

Figure 4.14 **Selecting a file to view its contents**



3. Locate the desired code in the source file.

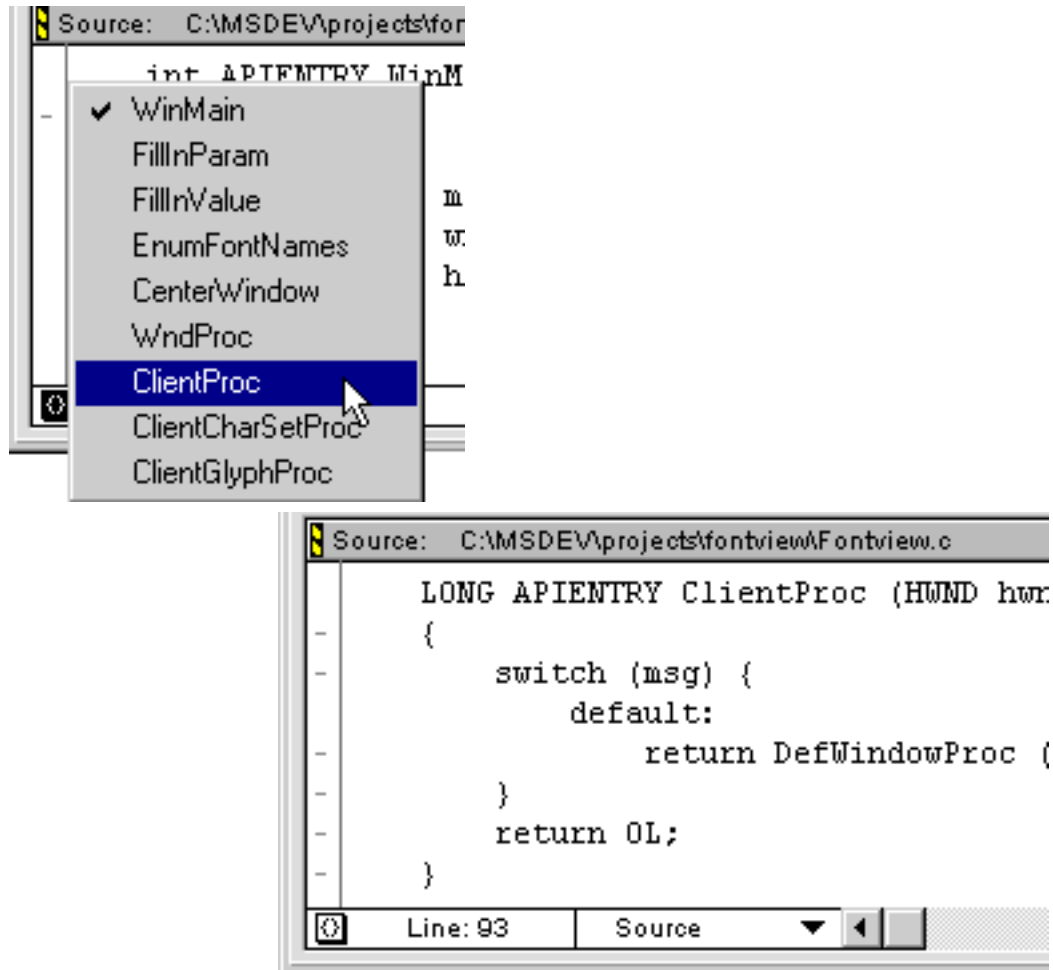
You can scroll the source pane to find the code you want. A more useful technique is to use the browser window's function pane or function pop-up menu to select the desired routine (Figure 4.15).

The routine appears in the source pane of the browser window. Once the routine is displayed, you can set and clear breakpoints. (See "Breakpoints" on page 60.)

Basic Debugging

Navigating Code

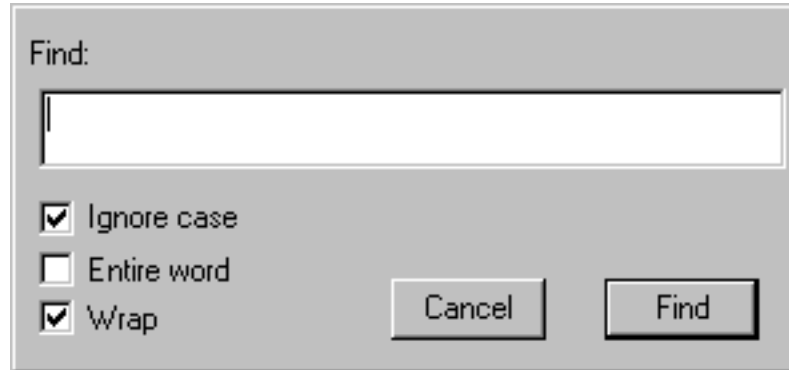
Figure 4.15 Choosing a routine to view



Using the Find Dialog

The Find dialog box (Figure 4.16) allows you to search for text in the source pane of the program or browser window. The search begins at the current location of the selection or insertion point and proceeds forward toward the end of the file. Choose **Find** from the Edit menu.

Figure 4.16 Find dialog



The Find dialog box contains the following items:

- **Text:** An editable text box for entering the text to search for.
- **Ignore case:** If selected, makes the search case-insensitive: that is, corresponding upper- and lowercase letters (such as A and a) are considered identical. If deselected, the search is case-sensitive: upper- and lowercase letters are considered distinct.
- **Entire word:** If selected, the search will find only complete words (delimited by punctuation or white-space characters) matching the specified search string. If deselected, the search will find occurrences of the search string embedded within larger words, such as the in other.
- **Wrap:** If selected, the search will “wrap around” when it reaches the end of the file and continue from the beginning. If deselected, the search will end on reaching the end of the file.
- **Find:** Confirms the contents of the dialog box and begins the search. The settings in the dialog box are remembered and will be redisplayed when the Find command is invoked again.
- **Cancel:** Dismisses the dialog box without performing a search. The settings in the dialog box are not remembered and will revert to their previous values when the Find command is invoked again.

Use the **Find Next** command to repeat the last search, starting from the current location of the selection or insertion point.

Basic Debugging

Breakpoints

Use the **Find Selection** command to search for the next occurrence of the text currently selected in the source pane. This command is disabled if there is no current selection, or only an insertion point.



TIP: You can reverse the direction of the search by using the shift key with the keyboard shortcuts, Ctrl-G (find next) or Ctrl-H (find selection).

Breakpoints

A *breakpoint* suspends execution of the target program and returns control to the debugger. When the debugger reaches a statement with a breakpoint, it stops the program before the statement is about to execute. The debugger then displays the routine containing the breakpoint in the program window. The current-statement arrow appears at the breakpoint, ready to execute the statement it points to.

This section discusses:

- Setting Breakpoints
- Clearing Breakpoints
- Temporary Breakpoints
- Viewing Breakpoints
- Conditional Breakpoints

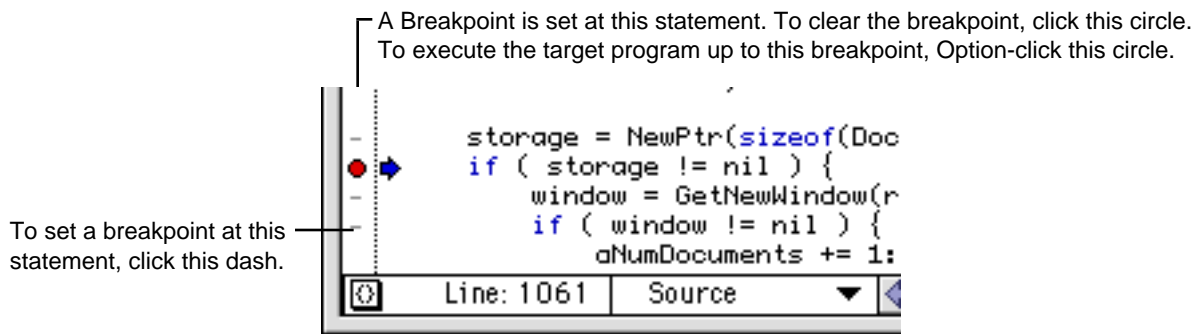
Setting Breakpoints

From the source pane of the program window or browser window, you can set a breakpoint on any line with a dash marker—the short line to the left of a statement in the breakpoint column (see Figure 4.17). The dash becomes a circle (red on a color monitor). This indicates that a breakpoint has been set at this statement. Execution will stop just before this statement is executed.

After setting a breakpoint, you can begin executing the program from either the browser or program window by choosing the **Step**, **Step Over**, **Step Into**, or **Run** commands from the Control menu.

See also “Running, Stepping, and Stopping Code” on page 44.

Figure 4.17 Setting breakpoints



TIP: Put one statement on each line of code. Not only is your code easier to read, it is easier to debug. The debugger allows only one breakpoint per line of source code, no matter how many statements a line has.

Clearing Breakpoints

To clear a single breakpoint, click the breakpoint circle next to it in the source pane. It turns back into a dash, indicating that you have removed the breakpoint. To clear all breakpoints, choose the **Clear All Breakpoints** command from the Edit menu.

Temporary Breakpoints

Sometimes you want to run a program to a particular statement and stop, and you want to do this just once. To set a temporary breakpoint, Alt-click the breakpoint dash to the left of the desired statement. When you resume execution, it will proceed to that statement and stop.

Basic Debugging

Breakpoints



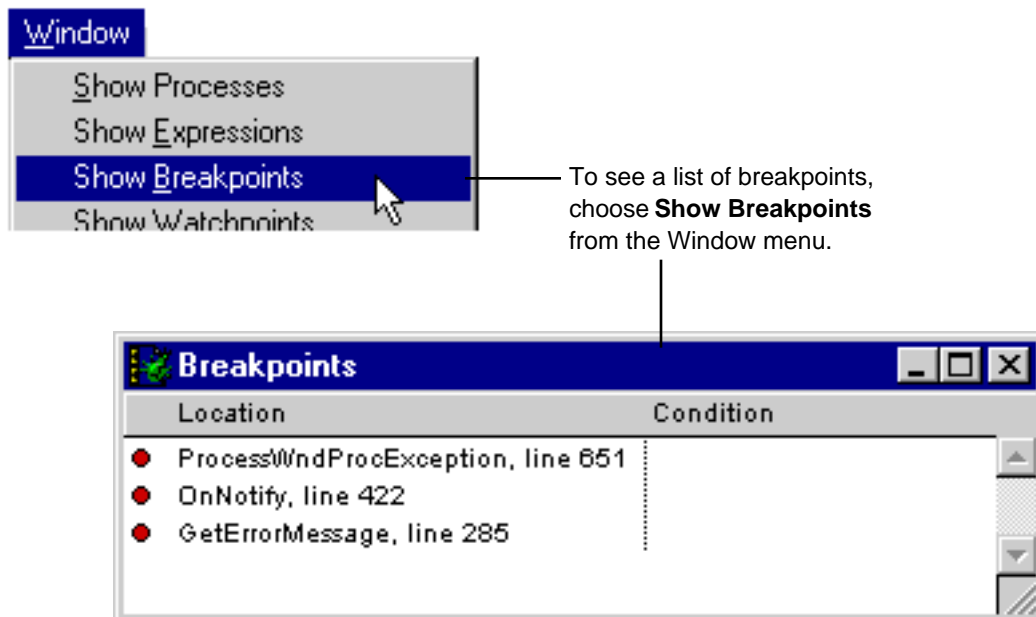
NOTE: If there is already a regular breakpoint at the statement, Alt-clicking removes the breakpoint, but the temporary breakpoint still works.

If another breakpoint is encountered before reaching the temporary breakpoint, the program will stop at the first breakpoint. The temporary breakpoint remains in place, however, and will be triggered and then removed when execution reaches it.

Viewing Breakpoints

To see a list of all breakpoints currently set in your program, choose the **Show Breakpoints** command from the Window menu. A window appears that lists the source file and line number for each breakpoint (Figure 4.18). Clicking a breakpoint marker in the breakpoint window turns a breakpoint on or off while remembering the breakpoint's position in the target program.

Figure 4.18 Displaying the breakpoint window





TIP: Double-clicking on a breakpoint location in the breakpoint window will take you to that line of code in the browser window.

See Also “Breakpoint Window” on page 35.

Conditional Breakpoints

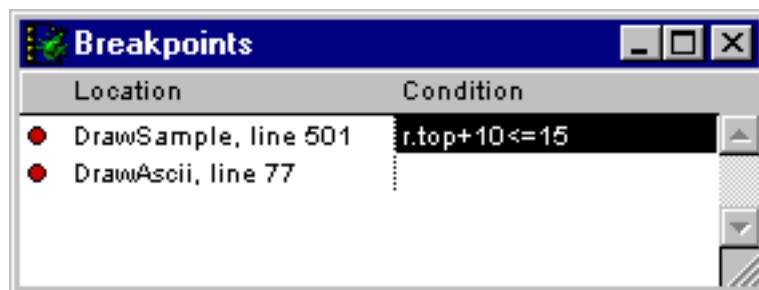
You can set *conditional breakpoints* that stop your program’s execution at a given point only when a specified condition is met. A conditional breakpoint is an ordinary breakpoint with a conditional expression attached. If the expression evaluates to a true (nonzero) value when control reaches the breakpoint, the program’s execution stops; if the value of the expression is false (zero), the breakpoint has no effect and program execution continues.

Conditional breakpoints are created in the breakpoint window. To make a conditional breakpoint:

1. **Set a breakpoint at the desired statement.**
2. **Display the breakpoint window by choosing Show Breakpoints from the Window menu.**
3. **In the breakpoint window, double-click the breakpoint’s condition field and enter an expression, or drag an expression from a source-code view or from the expression window.**

In Figure 4.19, the debugger will stop execution at line 501 in the `DrawSample()` routine if and only if the variable `r.top+10` is less than or equal to fifteen.

Figure 4.19 Creating a conditional breakpoint





TIP: Conditional breakpoints are especially useful when you want to stop inside a loop, but only after it has looped several times. You can set a conditional breakpoint inside the loop, and break when the loop index reaches the desired value.

Viewing and Changing Data

A critical feature of a debugger is the ability to see the current values of variables, and to change those values when necessary. This allows you to understand what is going on, and to experiment with new possibilities. This section discusses:

- Viewing Local Variables
- Viewing Global Variables
- Putting Data in a New Window
- Viewing Data Types
- Viewing Data in a Different Format
- Viewing Data as Different Types
- Changing the Value of a Variable
- Using the Expression Window
- Viewing Raw Memory
- Viewing Memory at an Address
- Viewing Processor Registers

See also For additional information on viewing and changing data for a particular target, see the corresponding Targeting manual.

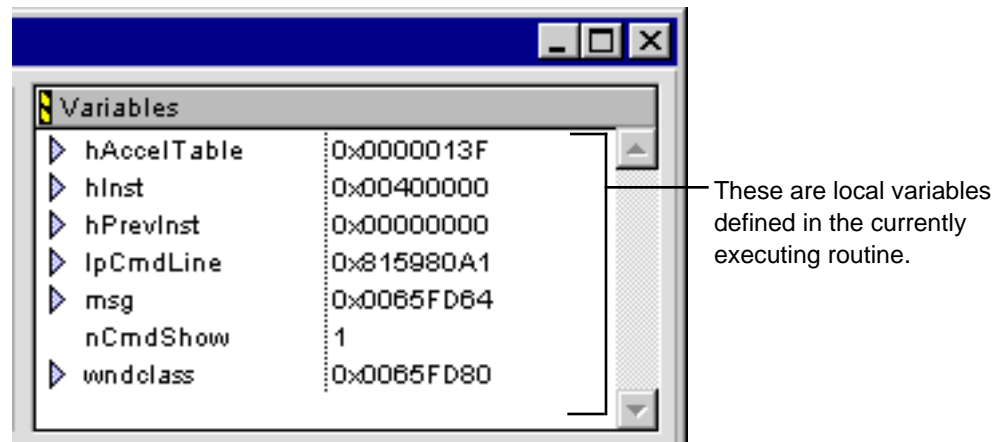
Viewing Local Variables

Local variables appear in the Variables pane of the program window (Figure 4.20). If the variable is a handle, pointer, or structure, you can click the arrow to the left of the name to expand the view. This allows you to see the members of the structure or the data referenced by the pointer or handle.

You can also expand or collapse variables by choosing the **Expand** or **Collapse All** commands from the Data menu.

See also “Expand” on page 106, “Collapse All” on page 106, and “Variables Pane” on page 21.

Figure 4.20 Viewing local variables



Viewing Global Variables

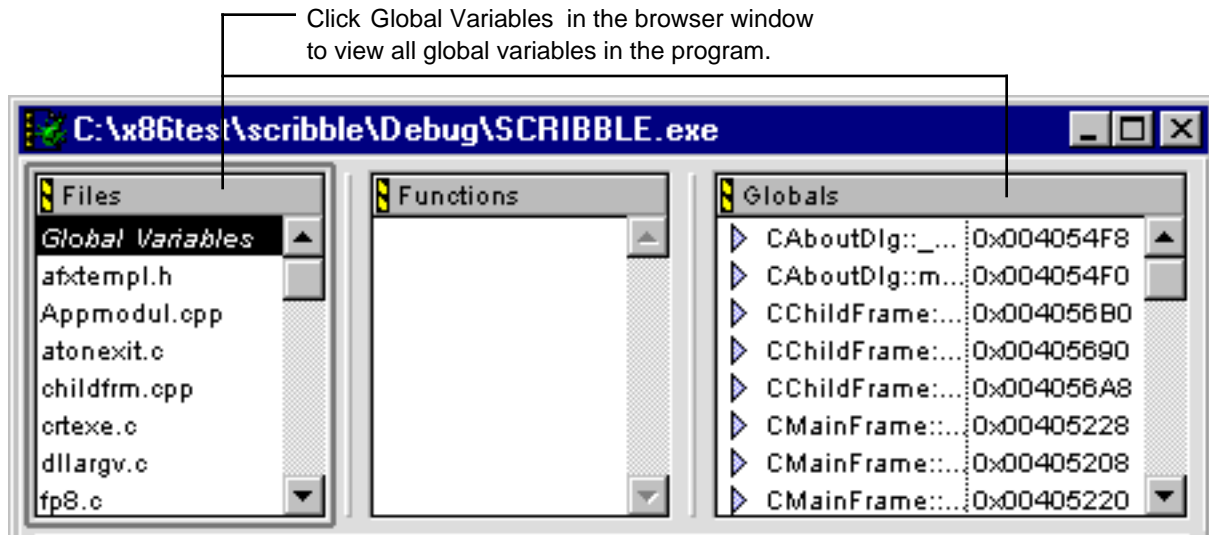
Global variables appear in the browser window (Figure 4.21). They appear in the globals pane when you select the *Global Variables* item in the File pane.

See also “Globals Pane” on page 31.

Basic Debugging

Viewing and Changing Data

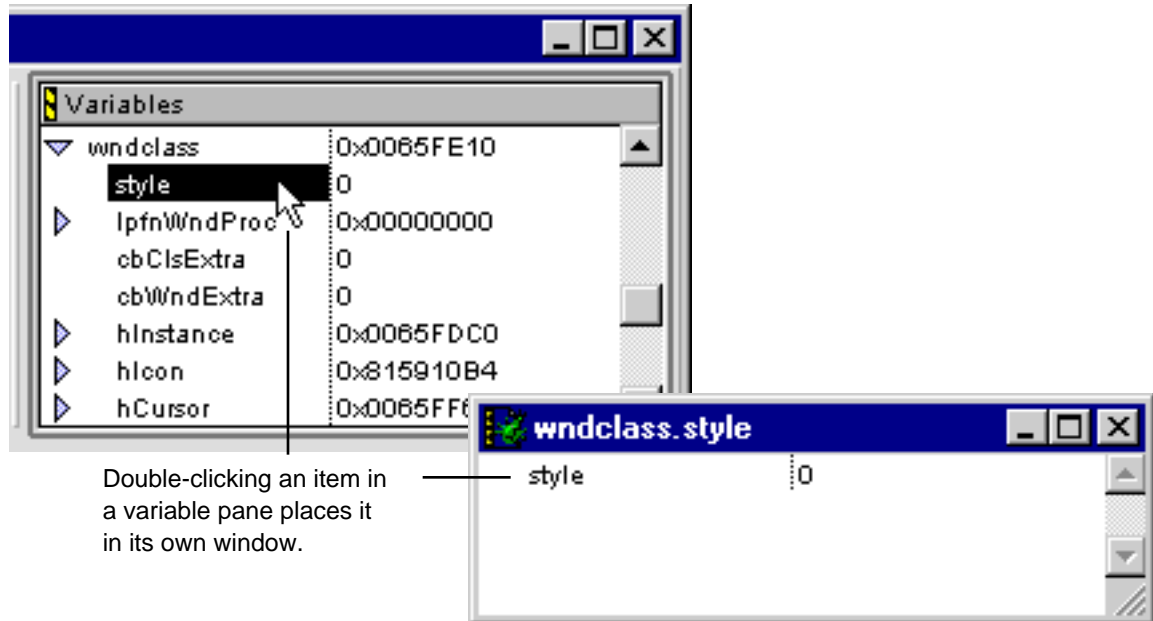
Figure 4.21 Viewing global variables in the browser window



Putting Data in a New Window

Sometimes the locals or globals panes are not the most convenient places to view data. You can place any variable or group of variables in a separate window or windows of their own.

Figure 4.22 Putting a variable in its own window



To place a variable or memory location in its own window, double-click its name (Figure 4.22) or select the name and choose the **Open Variable Window** command from the Data menu. If the variable is an array, use the **Open Array Window** command instead. To view the memory the variable occupies as a memory dump, use either the **View Memory** or **View Memory as** command.

See also “Variable Window” on page 36, “Array Window” on page 37, and “Memory Window” on page 39.

Viewing Data Types

If you wish, the debugger will display the data types of variables on a window-by-window basis. Select the window or pane in which you want data types displacement choose **Show Types** from the Data menu. Names of variables and memory locations in that window or pane will be followed by the relevant data type (Figure 4.23).

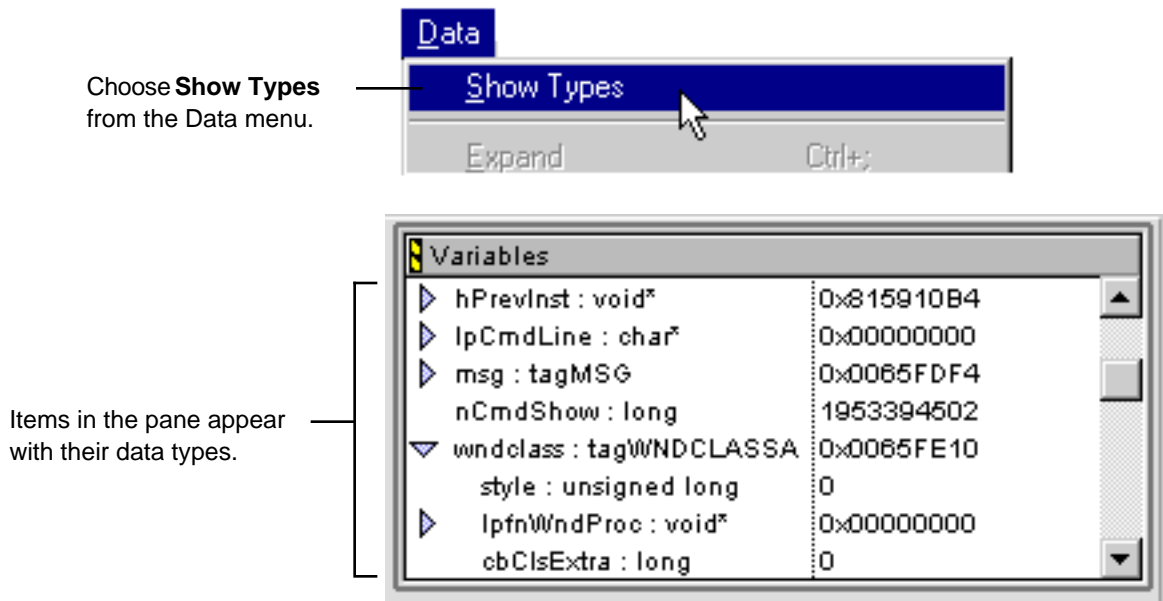
Basic Debugging

Viewing and Changing Data



TIP: To show data types automatically, select the **In variable panes, show variable types by default** preference in the **Preferences** dialog box. See “Settings & Display” on page 89 for more information.

Figure 4.23 Viewing data types



Viewing Data in a Different Format

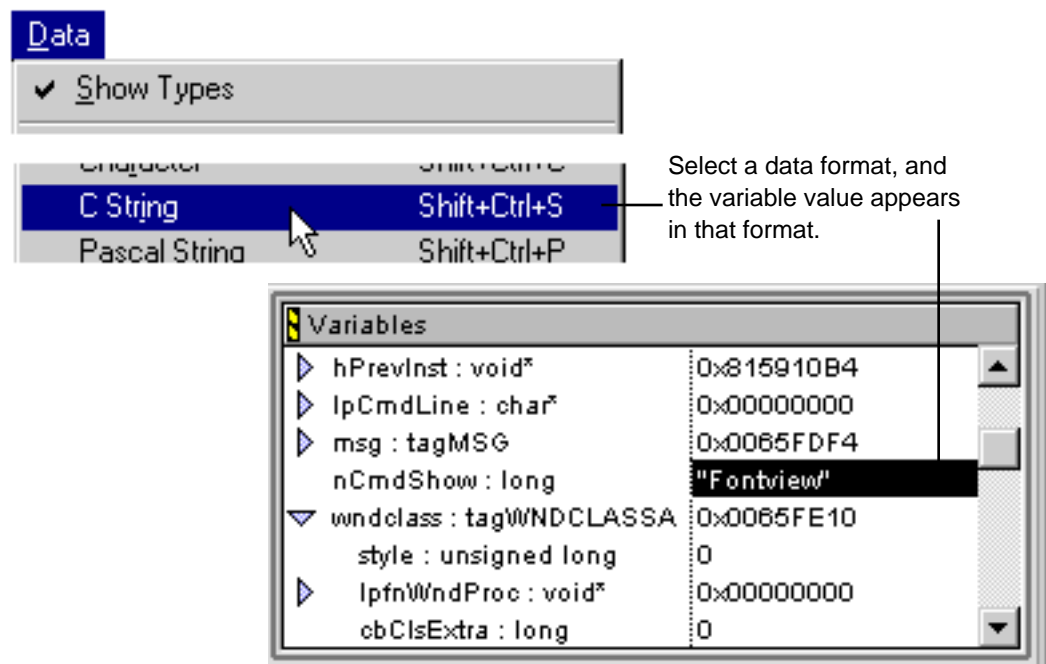
You can control the format in which a variable's value is displayed. The following options are available:

- signed decimal
- unsigned decimal
- hexadecimal
- character
- C string
- Pascal string
- floating-point

- enumeration
- Fixed
- Fract

To view data in a particular format, select either the name or the value of the variable in any window in which it is displayed, then choose the format you want from the Data menu (Figure 4.24).

Figure 4.24 Selecting a data format



Not all formats are available for all data types. For example, if a variable is an integral value (such as type short or long), you can view it in signed or unsigned decimal, hexadecimal, or even as a character or string, but not in floating-point, Fixed, or Fract format.

Viewing Data as Different Types

The **View as** command in the Data menu allows you to change the data type in which a variable, register, or memory is displayed:

1. **Select the item in a window or pane.**

Basic Debugging

Viewing and Changing Data

2. Choose View as from the Data menu.

A dialog box appears (Figure 4.25).

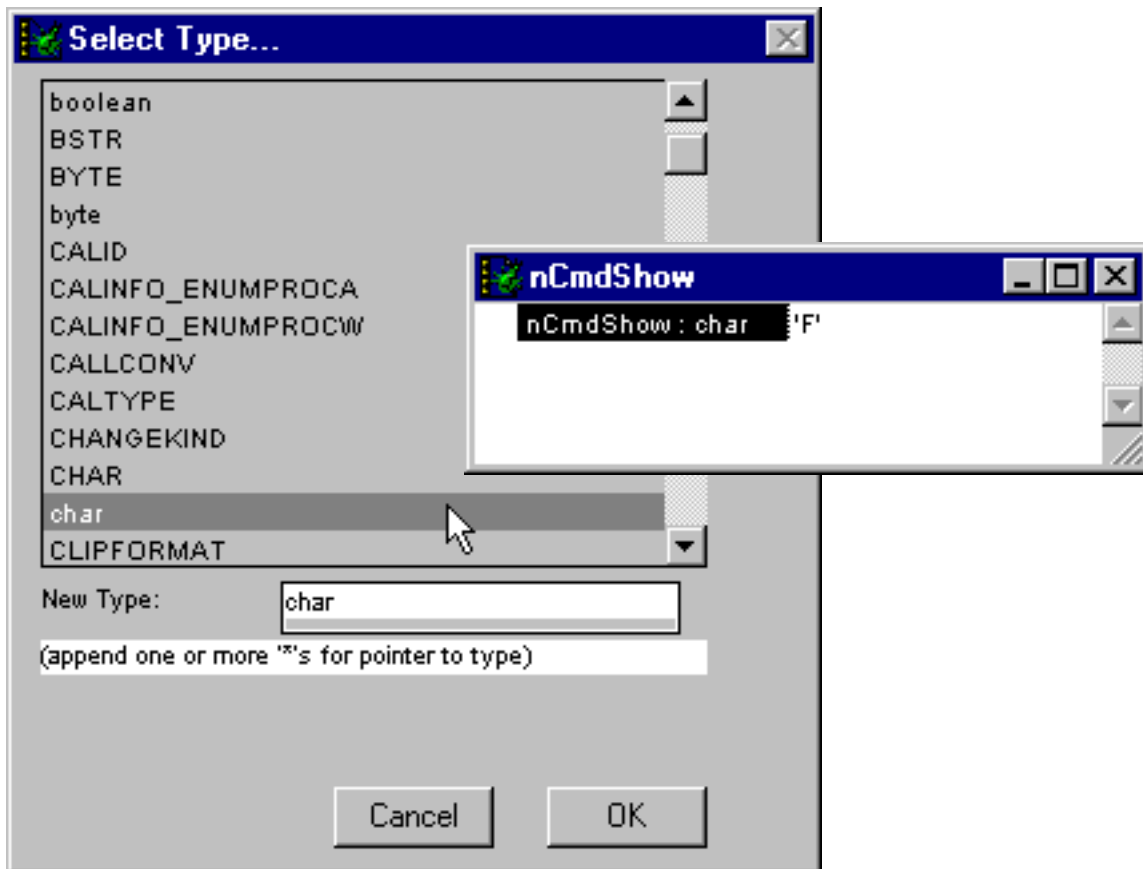
3. Select the data type by which to view the item.

The type name you select appears in the **New Type** box near the bottom of the dialog. If you want to treat the item as a pointer, append an asterisk (*) to the type name.

4. Click the OK button.

The display of the item's value changes to the specified type.

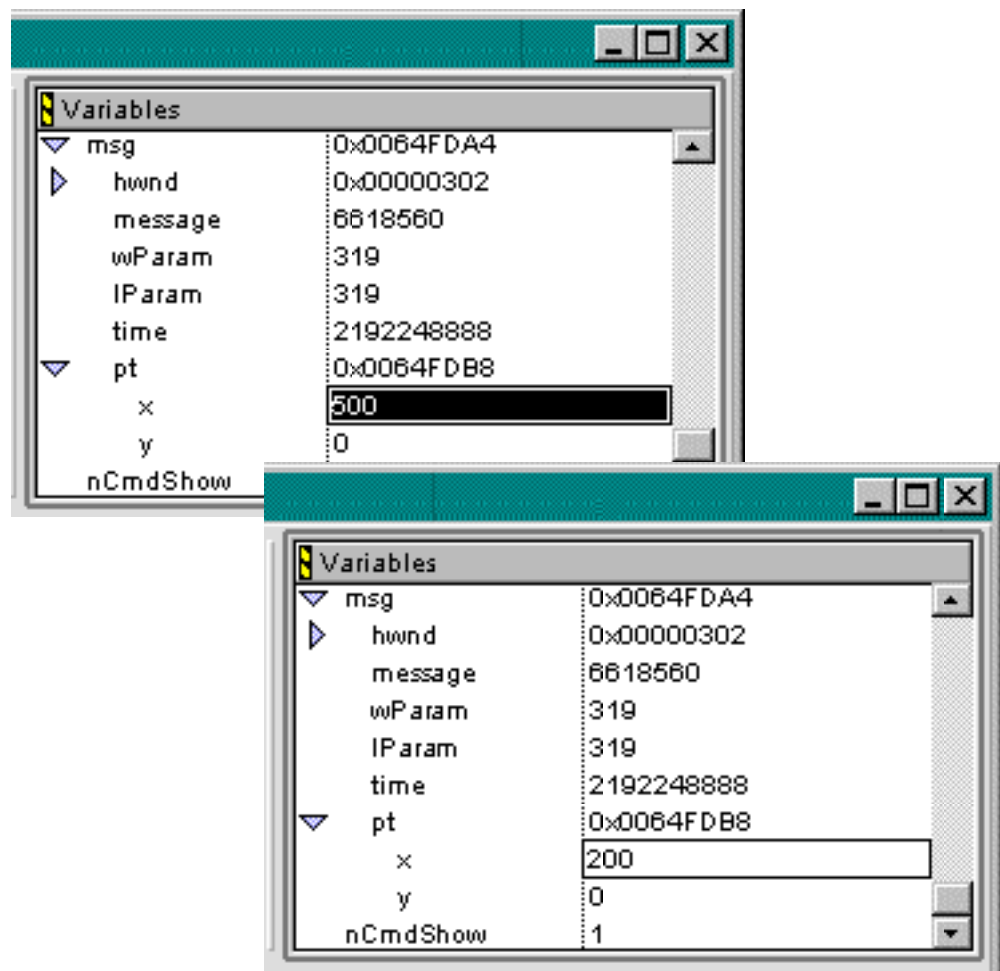
Figure 4.25 Selecting a data type



Changing the Value of a Variable

You can change the value of a variable in any window it's displayed in: the locals pane of the program window, the globals pane of the browser window, or any variable, array, or expression window. Just double-click the old value (or select it and press Enter) and type the new value (Figure 4.26).

Figure 4.26 Changing a variable value



Variable values can be entered in any of the following formats:

- decimal

Basic Debugging

Viewing and Changing Data

- hexadecimal
- floating-point
- C string
- Pascal string
- character constant

To enter a string or character constant, you must include C-style quotation marks (single quotes ' ' for a character constant, double " " for a string). For Pascal strings, include the escape sequence \p as the first character in the string.



WARNING! Changing variable values can be dangerous. The debugger allows you to set a variable to any value of the appropriate data type: for example, you could set a pointer to `nil` and crash the machine.

Using the Expression Window

The expression window provides a single place to put frequently used local and global variables, structure members, array elements, and even complex expressions without opening and manipulating a lot of windows. Open the window with the **Show Expressions** item in the Window menu. You can add an item to the expression window by dragging and dropping from another window, or by selecting the item and choosing **Copy to Expression** from the Data menu.

The contents of the expression window are updated whenever execution stops in the debugger. Any variable that is out of scope is left blank. You can take advantage of the expression window to perform a number of useful tricks:

- Move a routine's local variables to the expression window before expanding them to observe their contents. When the routine exits, its variables will remain in the expression window and will still be expanded when execution returns to the routine. The expression window does not automatically collapse local variables when execution leaves a routine, like the locals pane in the program window.

- Keep multiple copies of the same item displayed as different data types, by using the **Copy to Expression** and **View as** commands in the Data menu.
- Keep a sorted list of items. You can reorder items by dragging them within the expression window.
- View local variables from calling routines. You don't have to navigate back up the calling chain to display a caller's local variables (which hides the local variables of the currently executing routine). Add the caller's local variables to the expression window and you can view them without changing levels in the call-chain pane.

See Also “Expression Window” on page 33.

Viewing Raw Memory

To examine and change the raw contents of memory:

1. **Select an item or expression representing the base address of the memory you want to view examine.**
2. **Choose View Memory from the Data menu.**

A new memory window appears, displaying the contents of memory in hexadecimal and ASCII. You can change memory directly from the memory window by entering hexadecimal values or characters. You can also change the beginning address of the memory being displayed by changing the expression in the editable text box at the top of the window.

Viewing Memory at an Address

The **View Memory** and **View Memory as** commands in the Data menu allow you to follow any pointer—including an address stored in a register—and view the memory it currently points to. To display the memory referenced by a pointer:

1. **Select the value of the variable or register in a window in which it is displayed.**
2. **Choose View Memory or View Memory as from the Data menu.**

If you choose **View Memory**, a memory window opens displaying a raw memory dump starting at the address referenced by the pointer.

Basic Debugging

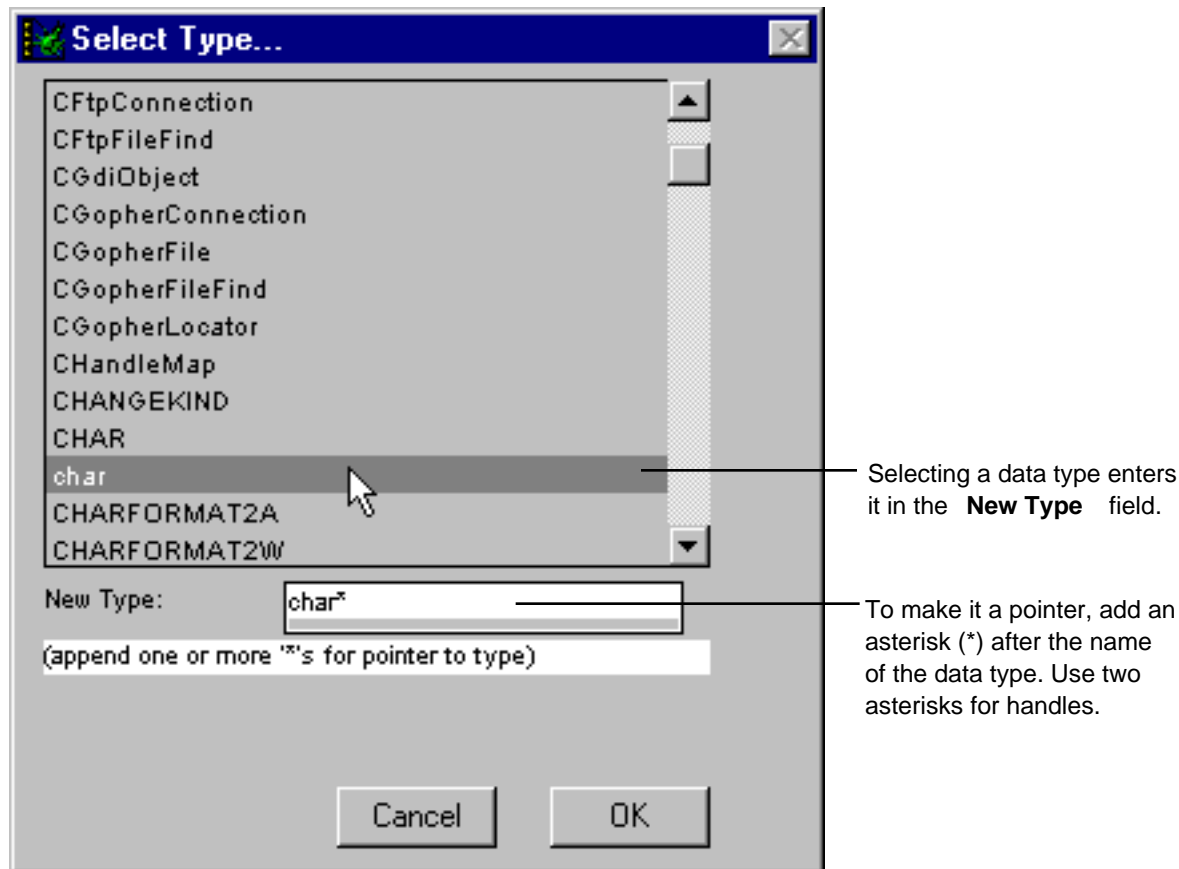
Viewing and Changing Data

If you choose **View Memory As**, a dialog box appears for selecting a data type (Figure 4.27); continue with step 3.

3. If you chose View Memory as, select a data type in the dialog box.

The type name you select appears in the **New Type** box near the bottom of the dialog. To view the memory a register points to, append an asterisk (*) to the type name.

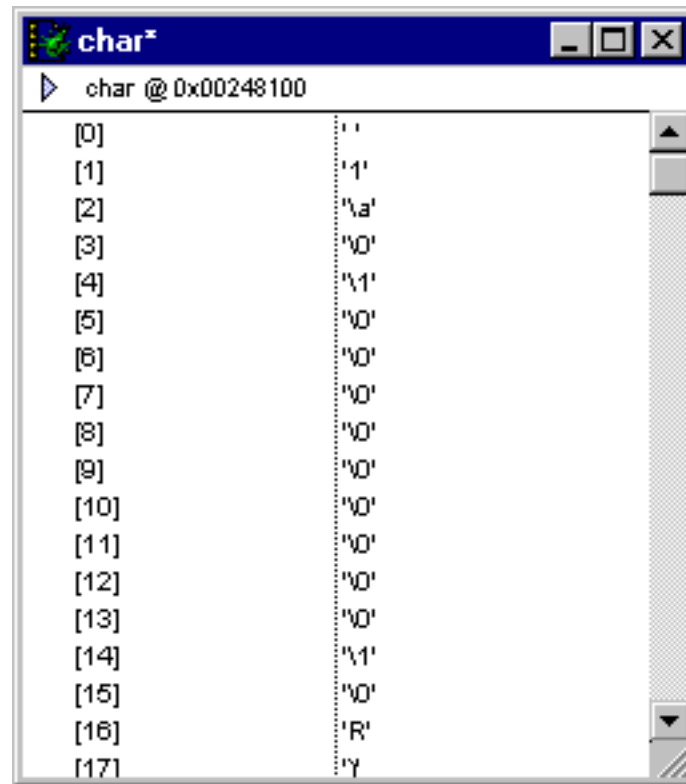
Figure 4.27 Choosing a data type to view memory



4. Click the OK button.

A new window appears (Figure 4.28) displaying the contents of memory starting at the address referenced by the pointer.

Figure 4.28 Viewing memory as a specified data type



See Also “Memory Window” on page 39.

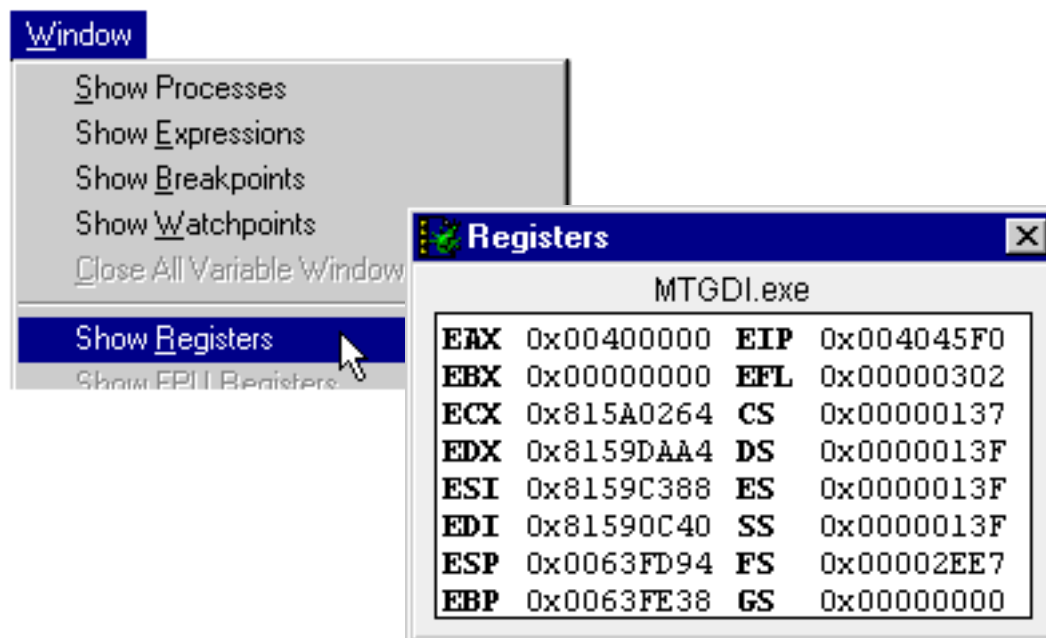
Viewing Processor Registers

To view the contents of the processor’s registers, choose **Show Registers** from the Window menu (Figure 4.29).

Basic Debugging

Viewing and Changing Data

Figure 4.29 Viewing processor registers



See Also "Register Windows" on page 40.



Expressions

Expressions are used in the CodeWarrior debugger to show the value of a numerical or logical computation or to make breakpoints conditional. This chapter describes their use.

Expressions Overview

An *expression* represents a computation that produces a value. The debugger displays the value in the expression window or acts upon the value when it is attached to a breakpoint in the breakpoint window. The debugger evaluates all expressions each time it executes a statement.

An expression can combine literal values (numbers and character strings), variables, registers, pointers, and C++ object members with operations such as addition, subtraction, logical and, and equality.

An expression may appear in the expression window, the breakpoint window, or a memory window. The debugger treats the result of the expression differently, depending on the window in which the expression appears.

In the Expression Window

The expression window shows expressions and their results. To see the value of an expression, place it in the expression window. To create a new expression:

1. **Display the expression window.**

Choose **Show Expressions** from the Window menu, or click in an open expression window to make it active.

2. **Choose New Expression from the Data menu.**

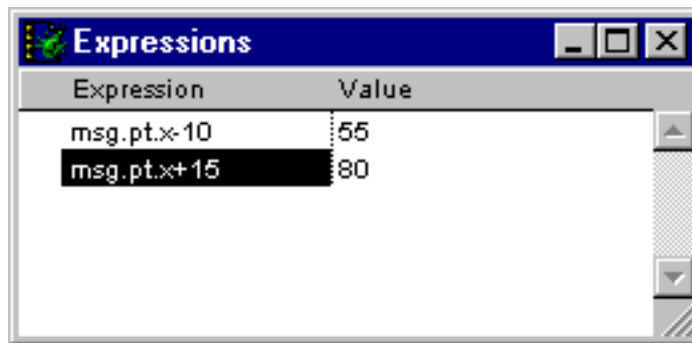
Expressions

Expressions Overview

3. Type a new expression and press Enter or Return.

The expression's value appears in the value column next to the expression (Figure 5.1). You can also create a new expression by dragging a variable or expression from another window to the expression window.

Figure 5.1 An expression in the expression window



The expression window treats all expressions as arithmetic: the debugger does not interpret the expression's result as a logical value, as it does in the breakpoint window.

See also "Expression Window" on page 33.

In the Breakpoint Window

You can attach an expression to a breakpoint in the breakpoint window. The breakpoint window treats expressions as logical rather than arithmetic. If the result of the expression is zero, it is considered false and the debugger ignores the breakpoint and continues execution; if the result is nonzero, it is considered true and the debugger stops at the breakpoint if the breakpoint is active.

To learn how to set a breakpoint, see "Setting Breakpoints" on page 60. Once you have set a breakpoint, you can attach an expression to it to make it conditional on that expression:

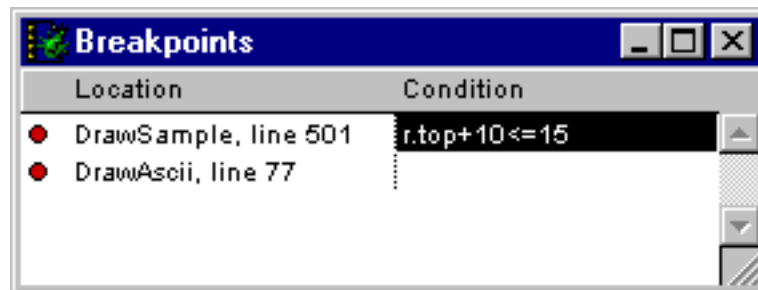
1. Display the breakpoint window.

Choose **Show Breakpoints** from the Window menu, or click in an open breakpoint window to make it active.

2. Set a condition.

Double-click the condition field for the desired breakpoint and type an expression (Figure 5.2). You can also add or change a breakpoint's condition by dragging an expression from another window and dropping it on the breakpoint's condition.

Figure 5.2 An expression in the breakpoint window



A conditional breakpoint stops the program if the expression yields a true (nonzero) result when execution reaches the breakpoint. If the expression produces a false (zero) result, execution continues without stopping.

See also “Breakpoint Window” on page 35 and “Conditional Breakpoints” on page 63.

In a Memory Window

In a memory window, expressions are treated as addresses. The expression in the text box at the top of the window defines the base address for the memory displayed in the window. To change a memory window's base address:

1. Display the memory window.

Choose **View Memory** from the Data menu, or click in an open memory window to make it active.

2. Enter a new expression.

Double-click the expression field and type an expression. You can also drag an expression from another window and drop it in the memory window's base-address field.

The window will display the contents of memory beginning at the address obtained by evaluating the new expression.

See also "Memory Window" on page 39.

Using Expressions

The debugger's expression syntax is similar to that of C/C++, with a few additions and limitations. Pascal style expressions are also supported.

Special Expression Features

Expressions can refer to specific items:

- The debugger considers integer values to be 4 bytes long. Use the short data type to denote a 2-byte integer value.
- The debugger treats double values as objects 10 bytes (80 bits) in length, rather than 8 bytes (64 bits).
- To compare character strings, use the == (equal) and != (not equal) operators. Note that the debugger distinguishes between Pascal- and C-format strings. Use the prefix \p when comparing Pascal string literals. The expression

```
"Nov shmoz ka pop" == "\pNov shmoz ka pop"
```

yields a false result, because it compares a C string and a Pascal string.

Expression Limitations

Expressions have a few limitations:

- Do not use C/C++ preprocessor definitions and macros (defined with the #define directive). They are not available to

the expression evaluator, even though they are defined in the source code.

- Do not use operations involving side effects. The increment (`i++`) and decrement (`i--`) operators, as well as assignments (`i = j`), are not allowed.
- Do not call functions.
- Do not use function names or pointers to functions.
- Do not use expression lists.
- Do not use pointers to C++ class members.
- The debugger cannot distinguish between identical variable names used in nested blocks to represent different variables (see Listing 5.1).

Listing 5.1 Identical variable names in nested blocks (C++)

```
// The debugger can't distinguish between x the
// int variable and x the double variable. If x
// is used in an expression, the debugger won't
// know which one to use.

void f(void)
{
    int x = 0;
    ...
    {
        double x = 1.0;
        ...
    }
}
```

- Type definitions that are not available to the debugger cannot be used in expressions (see Listing 5.2).

Listing 5.2 Type definitions in expressions (C/C++)

```
// Use long in expressions; Int32 not available
typedef long Int32;

// Use Rect* in expressions; RectPtr not
// available
typedef Rect* RectPtr;
```

- Nested type information is not available. In Listing 5.3, use Inner, not Outer::Inner in a debugger expression.

Listing 5.3 Nested type information (C/C++)

```
// To refer to the i member, use Inner.i,
// not Outer::Inner.i

struct Outer
{
    struct Inner
    {
        int i;
    };
};
```

Example Expressions

The list below gives example expressions that you can use in any window that uses expressions.

- A literal decimal value:
160
- A literal hexadecimal value:
0xA0
- The value of a variable:
myVariable

- The value of a variable shifted 4 bits to the left:
`myVariable << 4`
- The difference of two variables:
`myRect.bottom - myRect.top`
- The maximum of two variables:
`(foo > bar) ? foo : bar`
- The value of the item pointed to by a pointer variable:
`*MyTablePtr`
- The size of a data structure (determined at compile time):
`sizeof(myRect)`
- The value of a member variable in a structure pointed to by a variable:
`myRectPtr->bottom`
or
`(*myRectPtr).bottom`
- The value of a class member in an object:
`myDrawing::theRect`

Below are examples of logical expressions: the result is considered true if non-zero, false if zero.

- Is the value of a variable false?
`!isDone`
or
`isDone == 0`
- Is the value of a variable true?
`isReady`
or
`isReady != 0`
- Is one variable greater than or equal to another?
`foo >= bar`
- Is one variable less than both of two others?
`(foo < bar) && (foo < car)`
- Is the 4th bit in a character value set to 1?
`((char)foo >> 3) & 0x01`
- Is a C string variable equal to a literal string?

```
cstr == "Nov shmoz ka pop"
```

- Is a Pascal string variable equal to a literal string?

```
pstr == "\pScram gravy ain't wavy"
```
- Always true:
1
- Always false:
0

Expression Syntax

This section defines the debugger's expression syntax. The first line in a definition identifies the item being defined. Each indented line represents a definition for that item. An item with more than one definition has each definition listed on a new line. Items enclosed in angle brackets (<>) are defined elsewhere. Items in *italic* typeface are to be replaced by a value or symbol. All other items are literals to be used exactly as they appear.

For example,

```
<name>  
    identifier  
    <qualified-name>
```

defines the syntax of a name. A name can be either an identifier or a qualified name; the latter is a syntactic category defined in another of the definitions listed here.

```
<name>  
    identifier  
    <qualified-name>  
  
<typedef-name>  
    identifier  
  
<class-name>  
    identifier  
  
<qualified-name>  
    <qualified-class-name>::<name>
```

```
<qualified-class-name>
    <class-name>
    <class-name>::<qualified-class-name>

<complete-class-name>
    <qualified-class-name>
    :: <qualified-class-name>

<qualified-type-name>
    <typedef-name>
    <class-name>::<qualified-type-name>

<simple-type-name>
    <complete-class-name>
    <qualified-type-name>
    char
    short
    int
    long
    signed
    unsigned
    float
    double
    void

<ptr-operator>
    *
    &

<type-specifier>
    <simple-type-name>

<type-specifier-list>
    <type-specifier> <type-specifier-list>(opt)

<abstract-declarator>
    <ptr-operator> <abstract-declarator>(opt)
    (<abstract-declarator>)

<type-name>
    <type-specifier-list> <abstract-
declarator>(opt)

<literal>
    integer-constant
```

Expressions

Expression Syntax

```
character-constant
floating-constant
string-literal

<register-name>
    ®PC
    ®SP
    ®Dnumber
    ®Anumber

<register-name>
    ®Rnumber
    ®FPRnumber
    ®RTOC

<primary-expression>
    <literal>
    this
    ::identifier
    ::<qualified-name>
    (<expression>)
    <name>
    <register-name>

<postfix-expression>
    <primary-expression>
    <postfix-expression>[<expression>]
    <postfix-expression>.<name>
    <postfix-expression>-><name>

<unary-operator>
    *
    &
    +
    -
    !
    ~

<unary-expression>
    <postfix-expression>
    <unary-operator> <cast-expression>
    sizeof <unary-expression>
    sizeof(<type-name>)
```

```
<cast-expression>
    <unary-expression>
    ( <type-name> ) <cast-expression>

<multiplicative-expression>
    <cast-expression>
    <multiplicative-expression> * <cast-expression>
    <multiplicative-expression> / <cast-expression>
    <multiplicative-expression> % <cast-expression>

<additive-expression>
    <multiplicative-expression>
    <additive-expression> + <multiplicative-expression>
    <additive-expression> - <multiplicative-expression>

<shift-expression>
    <additive-expression>
    <shift-expression> << <additive-expression>
    <shift-expression> >> <additive-expression>

<relational-expression>
    <shift-expression>
    <relational-expression> < <shift-expression>
    <relational-expression> > <shift-expression>
    <relational-expression> <= <shift-expression>
    <relational-expression> >= <shift-expression>

<equality-expression>
    <relational-expression>
    <equality-expression> == <relational-expression>
    <equality-expression> != <relational-expression>

<and-expression>
    <equality-expression>
    <and-expression> & <equality-expression>
```

Expressions

Expression Syntax

```
<exclusive-or-expression>
    <and-expression>
    <exclusive-or-expression> ^ <and-expression>
<inclusive-or-expression>
    <exclusive-or-expression>
    <inclusive-or-expression> | <exclusive-or-expression>
<logical-and-expression>
    <inclusive-or-expression>
    <logical-and-expression> && <inclusive-or-expression>
<logical-or-expression>
    <logical-and-expression>
    <logical-or-expression> || <logical-and-expression>
<conditional-expression>
    <logical-or-expression>
    <logical-or-expression> ? <expression> :
<conditional-expression>
<expression>
    <conditional-expression>
```




Debugger Preferences

This chapter discusses the MW Debug preferences. It covers every option in each preference panel, and describes what each option does.

Debugger Preferences Overview

The Preferences dialog box allows you to change various aspects of the debugger's behavior. There are five primary debugger panels. Additional panels may appear for particular targets. See the appropriate targeting manual for complete information on target-specific debugger preferences. The five primary panels are:

- **Settings & Display**—Options related to the saving of settings in preference files, and display of various items in the debuggers windows.
- **Symbolics**—Options related to opening, closing, and management of SYM files and Java class and zip files.
- **Program Control**—Options related to the launching, termination, and control of the program being debugged.
- **Win32 Settings**—Option to choose preferred editor to edit source files.
- **Runtime Settings**—Options specific to the configuration of the Windows environment.

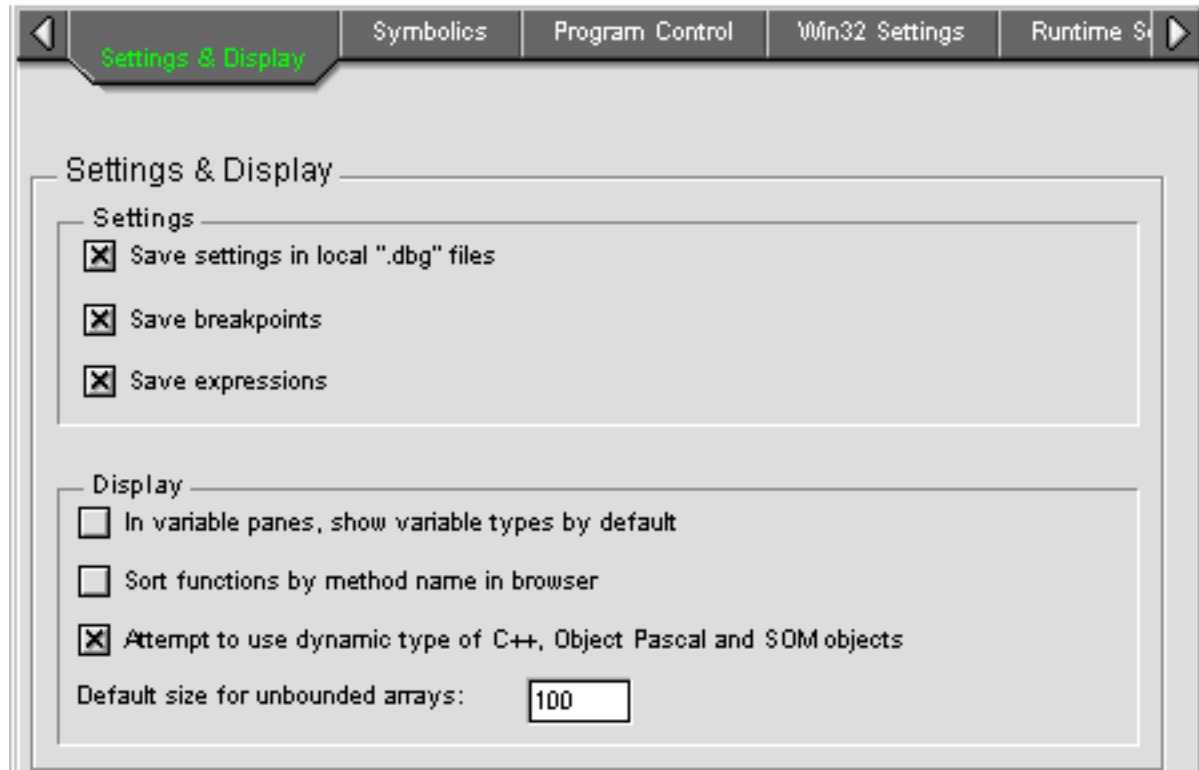
Settings & Display

The Settings & Display panel is shown in Figure 6.1.

Debugger Preferences

Settings & Display

Figure 6.1 Settings and Display preferences



Save settings in local ".dbg" files

Saves window size and position in .dbg files. These files optionally contain breakpoints and expressions. Selecting this preference creates a new .dbg file or modifies an existing one for every SYM file you open with the debugger. If you deselect this preference, the .dbg file is still created, but the window and other data are not saved.

Save breakpoints

Enabled if **Save window settings in local ".dbg" files** is selected. Saves breakpoint settings in the SYM file's .dbg file. If you deselect this preference, breakpoint settings are forgotten when saving the .dbg file.

Save expressions

Enabled if **Save window settings in local “.dbg” files** is selected. Saves the contents of the expression window in the SYM file's .dbg file. If you deselect this preference, the expression window's contents are forgotten when saving the .dbg file.

See also “Expression Window” on page 33.

In variable panes, show variable types by default

Shows variable types when a new variable window is opened. This setting is stored in the .dbg file.

Settings in the project's .dbg file take precedence over this preference. Variable windows that were opened before setting the preference will use the settings found in the .dbg file.

Sort functions by method name in browser

Changes the way C++, Object Pascal, and Java functions are sorted in the browser window's function pane. When this preference is deselected, function names of the form *className::methodName* are sorted alphabetically by class name first, then by method name within the class. Selecting this preference causes the functions to be alphabetized directly by method name. Since most C++, Object Pascal, and Java source-code files tend to contain methods all of the same class, this preference makes it easier to select methods in the function pane by typing from the keyboard.

Attempt to use dynamic type of C++, Object Pascal objects and SOM objects

Displays the runtime type of C++ or Object Pascal objects; deselecting this preference displays an object's static type only. The debugger can determine dynamic types only for classes with at least one virtual function. Virtual base classes are not supported.

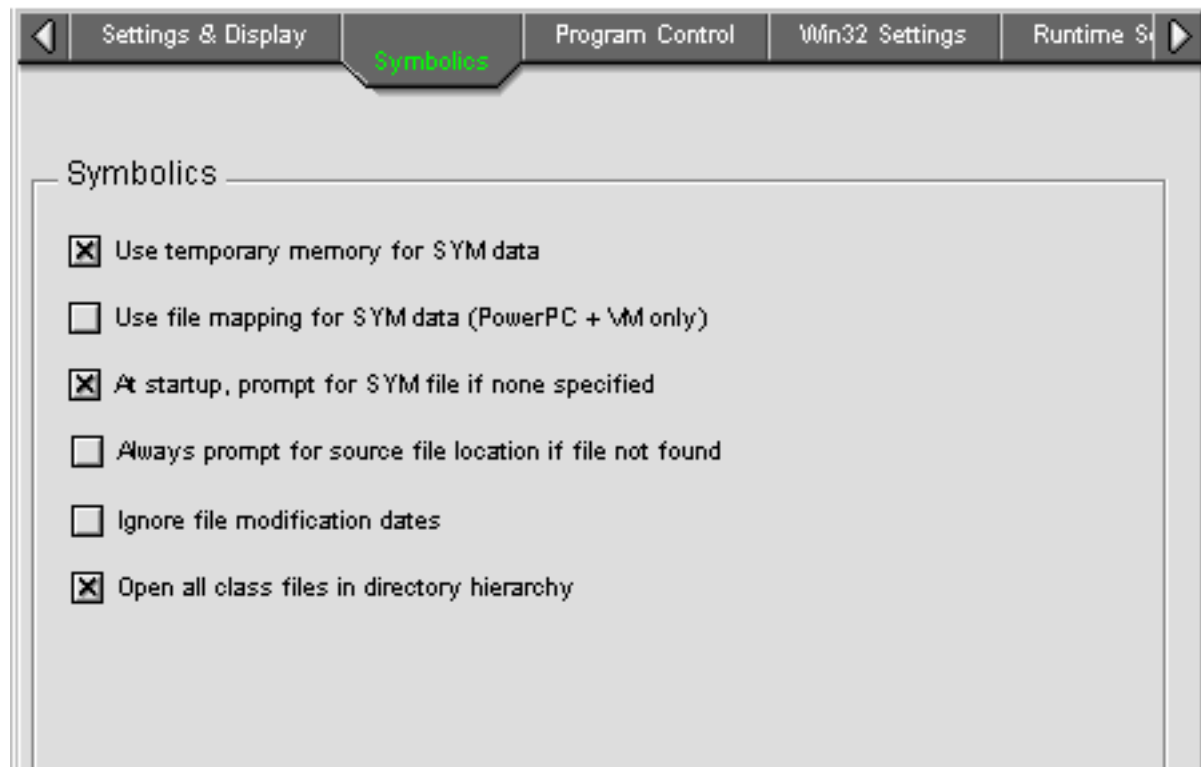
Default size for unbound arrays

Specifies the array size to use when no size information is available.

Symbolics

The Symbolics panel is shown in Figure 6.2.

Figure 6.2 Symbolics preferences



Use temporary memory for SYM data

Uses temporary memory to store the SYM file's data. This keeps the debugger's memory partition small and interferes less with the target program as it executes. Deselecting this preference forces the debugger to use more memory, leaving less available for the target program.

Use file mapping for SYM data (PowerPC + VM only)

Allows you to use virtual-memory file mapping to access the program's symbolic information. This feature is only available on com-

puters running Windows NT 4.0 with applications built using CodeView symbolics.

At startup, prompt for SYM file if none specified

Prompts for a SYM file to open when the debugger is launched by itself. Deselecting this checkbox allows the debugger to launch without prompting for a SYM file.

Always prompt for source file location if file not found

Prompts you to locate source-code files for the target program if the debugger cannot find them. The debugger normally remembers the locations of these files; selecting this preference causes it to prompt you for the location of missing source code files, even if it has previously recorded their locations.

Ignore file modification dates

The debugger keeps track of the modification dates of source files from which a symbolics file is created. If the modification dates don't match, the debugger normally displays an alert box warning you of possible discrepancies between the object code and the source code. Selecting this preference disables this warning; deselecting the preference enables the warning.

Open all class files in directory hierarchy

If this option is enabled, the debugger opens all the class files in the directory and all contained directories and merges them all together in the same Browser window.

See also *Targeting Java* for information on Java-specific preferences.

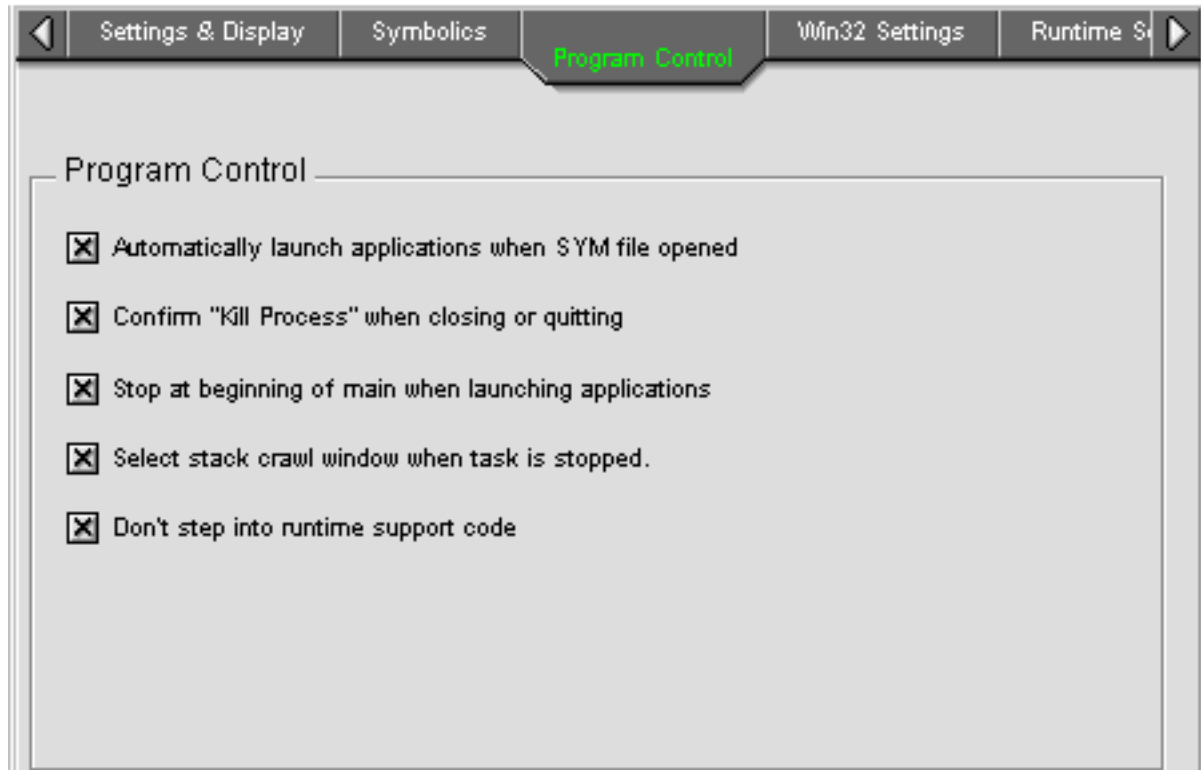
Program Control

The Program Control panel is shown in Figure 6.3.

Debugger Preferences

Program Control

Figure 6.3 Program Control preference



Automatically launch applications when SYM file opened

Automatically launches a target program when its SYM file is opened, setting an implicit breakpoint at the program's main entry point. Deselecting this preference allows you to open a SYM file without launching the program, so that you can examine object code that executes before the main routine, such as C++ static constructors. You can also avoid launching the target program by holding down the Alt key when opening a SYM file.

Confirm "Kill Process" when closing or quitting

Prompts for confirmation before aborting a process when a target program is killed.

Set breakpoint at program main when launching applications

Usually when you begin debugging an application, the debugger stops at the first line of `main()`. You must then choose the **Run** command a second time to continue past that point. Turning off this option means that when you debug your application, it does not stop at `main()` and instead runs free. This option is most useful between debugging sessions after you've set all your breakpoints.

Don't step into runtime support code

Executes constructor code for C++ static objects normally, without displaying it in the program window.

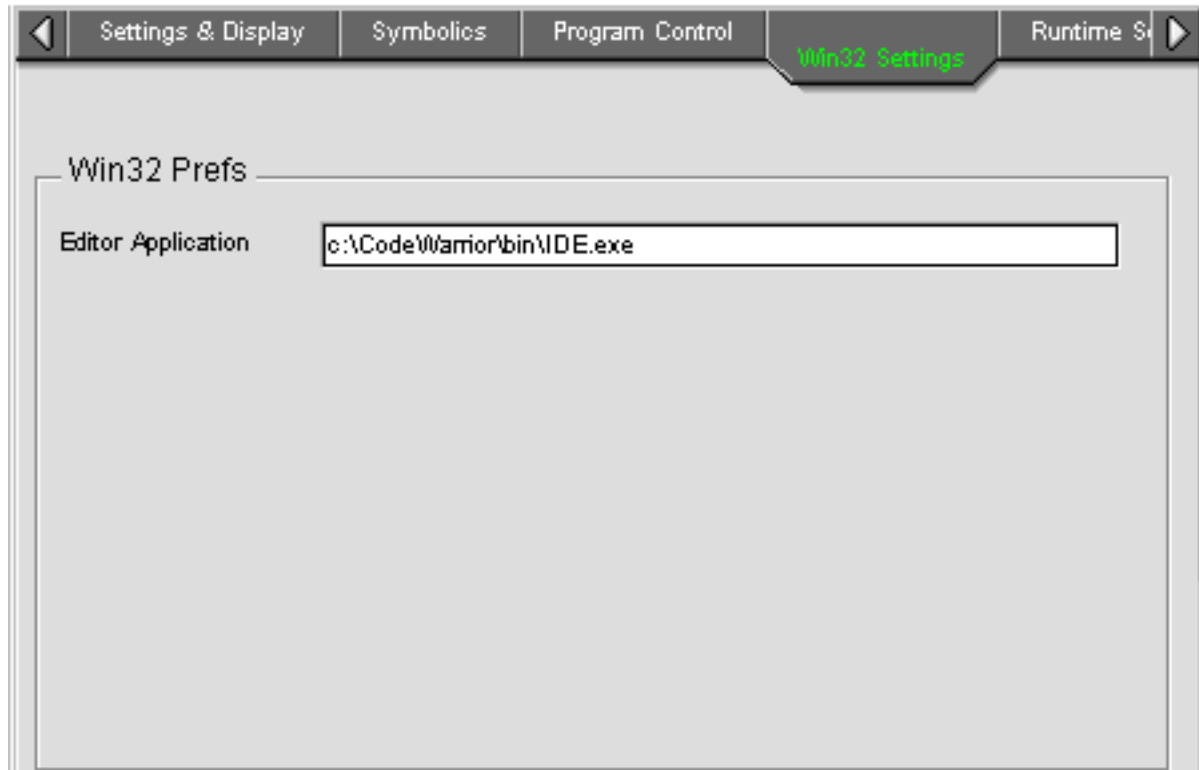
Win32 Settings

The Win32 Settings panel is shown in Figure 6.4. Use this panel to set up the default editor you would like to use to edit files. If no default editor application is specified, the file will be opened by Note-Pad.

Debugger Preferences

Win32 Settings

Figure 6.4 Win32 Settings



You can edit any file in the debugger by double-clicking the filename in the Files pane, or by choosing the **Edit filename** command from the File menu (Figure 6.5).

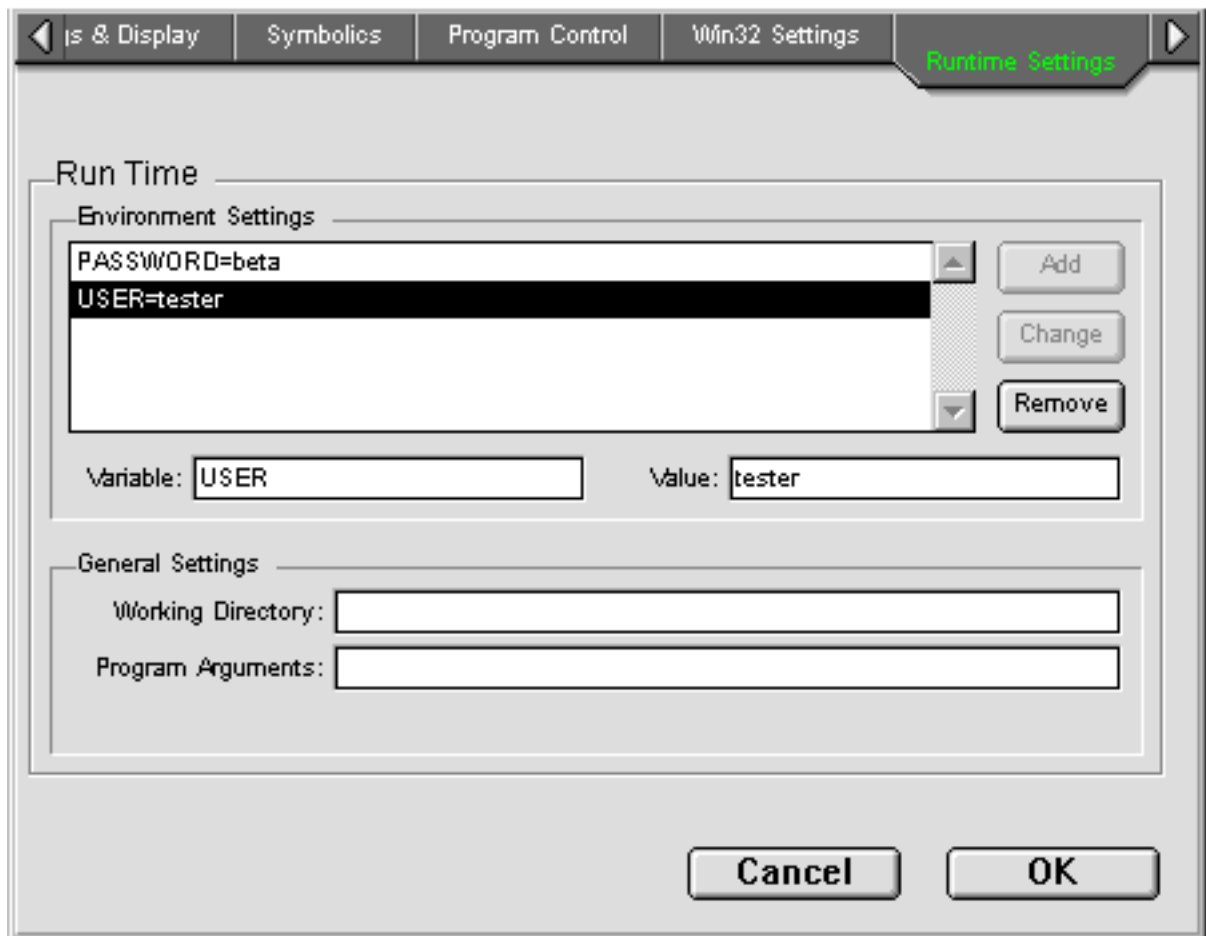
Figure 6.5 Editing a file from MW Debug



Runtime Settings

The Runtime Settings panel is shown in Figure 6.6. This panel consists of two main areas: Environment Settings and General Settings.

Figure 6.6 Runtime Settings



The Environment Settings area allows you to specify environment variables that are set and passed to your program as part of the `envp` parameter in the `main()` or available from the `getenv()` call and are only available to the target program. When the your program terminates, the settings are no longer available. For example, if you are writing a program that logs into a server, you could use variables for userid and password.

Debugger Preferences

Runtime Settings

The General Settings area has the following fields:

Working Directory: Use this field to specify the directory in which debugging occurs. If no directory is specified, debugging occurs in the same directory the executable is located.

Program Arguments: Use this field to pass command-line arguments to your program at startup. Your program receives these arguments when started with the Run command.



Debugger Menus

This reference chapter describes each menu item in MW Debug.

Debugger Menus Overview

There are six menus:

- Help menu—learn about MW Debug
- File Menu—open and close symbolic files, open source files, save log files, and quit
- Edit Menu—the standard editing operations, plus debugger preferences
- Control Menu—manage your path through code, or switch to a low-level debugger
- Data Menu—manage the display of data in the debugger
- Window Menu—open and close various display windows in the debugger

Help menu

The Help menu contains three commands to access the MW Debug help file. The fourth command, **About Metrowerks Debugger**, displays copyright and author information about the application, as well as credits.

File Menu

The commands in the File menu allow you to open, close, edit, and save files.

Debugger Menus

File Menu

Open

Opens an existing symbolics file (in either SYM or CodeView formats) to debug. A Standard File dialog box appears, prompting you to select a symbolics file. The SYM file must be in the same folder as its target program (the program you want to debug).

After you choose the symbolics file, the debugger loads it into memory, loads the target program, places an implicit breakpoint at the program's main entry point, and launches the program. The program then pauses at the initial breakpoint, returning control to the debugger.

The **Open** command can also be used to open Java class or zip files. The debugger reads the symbolics from these files and treats them as if they were symbolics files. See *Targeting Java* for more information.

See also "Preparing for Debugging" on page 13 for information on generating symbolic information for your project.



NOTE: More than one program can be opened and debugged at the same time.

Close

Closes the active window.

If the **Confirm "Kill Process" when closing or quitting** preference is not selected, closing the program window kills the running program (if any); selecting the **Run** command reopens the program window and reexecutes the program from the beginning. If this preference is selected, a dialog box appears offering you the choice of killing the program, keeping it running even after closing the program window, or canceling the **Close** command.

See also "Confirm "Kill Process" when closing or quitting" on page 94.

Edit filename

Opens the designated source-code file in the default editor chosen in the Win32 Settings preference panel. If there is no default editor specified, the file will be opened by NotePad.

See also “Win32 Settings” on page 95.

Save

Saves the contents of the log window to the disk under the current file name. This command is enabled only when the log window is active.

Save As

Displays a Standard File dialog box for saving the contents of the log window under a different name. The new name becomes associated with the log window; later **Save** commands will save to the new file name rather than the old one. This command is enabled only when the log window is active.

Save Settings

Saves the current settings of the program and browser windows, provided that the **Save window settings in local “.dbg” files** preference is selected in the **Preferences** dialog box. This command also saves breakpoints and expressions if the **Save breakpoints** and **Save expressions** preferences are selected, respectively.

See also “Settings & Display” on page 89.

Quit

Quits the debugger.

If the **Confirm “Kill Process” when closing or quitting** preference is not selected, quitting the debugger kills all running programs (if any). If this preference is selected, a dialog box appears offering you the choice of killing the programs, keeping them running even after quitting the debugger, or canceling the **Quit** command.

See also “Confirm “Kill Process” when closing or quitting” on page 94.

Edit Menu

The commands on the Edit menu apply to variable values and expressions displayed in the debugger. The debugger does not allow editing of source code.

See also “Edit filename” on page 101 for information about how to edit source code that appears in a Source pane.

Undo

Reverses the effect of the last **Cut**, **Copy**, **Paste**, or **Clear** operation.

Cut

Deletes selected text and puts it in the Clipboard. You cannot cut text from a source pane.

Copy

Copies selected text into the Clipboard without deleting it. You can copy text from a source pane or from the log window.

Paste

Pastes text from the Clipboard into the active window. You cannot paste text into a source pane.

Clear

Deletes selected text without putting it in the Clipboard. You cannot clear text from a source pane.

Select All

Selects all text in the active window. You can select text only while editing a variable value or an expression, or in the log window.

Find

Opens a dialog box allowing you to search for text in the source pane of the program or browser window. The search begins at the current location of the selection or insertion point and proceeds forward toward the end of the file.

See Also “Using the Find Dialog” on page 58.

Find Next

Repeats the last search, starting from the current location of the selection or insertion point.

Find Selection

Searches for the next occurrence of the text currently selected in the source pane. This command is disabled if there is no current selection, or only an insertion point.



TIP: You can reverse the direction of the search by using the shift key with the keyboard shortcuts, Ctrl-G (find next) or Ctrl-H (find selection).

See Also “Using the Find Dialog” on page 58.

Preferences

Opens a dialog box that lets you change various aspects of the debugger’s behavior. Information on the preferences dialog box is introduced in “Debugger Preferences Overview” on page 89.

Control Menu

The Control menu contains commands that allow you to manage program execution.

Debugger Menus

Control Menu

Run

Executes the target program. Execution starts at the current-statement arrow and continues until encountering a breakpoint, or until you issue a **Stop** or **Kill** command.

See also “Running Your Code” on page 46.

Stop

Temporarily suspends execution of the target program and returns control to the debugger. When a **Stop** command is issued to an executing program, the program window appears showing the current values of the local variables. The current-statement arrow is positioned at the next statement to be executed, the **Stop** command is dimmed in the Control menu, and a message appears in the program window’s source pane.

To resume executing a stopped program, you may

- select the **Run** command. Execution will continue at the current-statement arrow.
- step through the target program one statement at a time with the **Step Over**, **Step Into**, or **Step Out** commands in the Control menu.

See also “Stopping Execution” on page 52.

Kill

Permanently terminates execution of the target program and returns control to the debugger. Using a breakpoint or the **Stop** command allows you to resume program execution from the point of suspension; the **Kill** command requires that you use **Run** to restart program execution from the main entry point.

See also “Killing Execution” on page 52.

Step Over

Executes a single statement, stepping over function calls. The statement indicated by the current-statement arrow is executed, then

control returns to the debugger. When the debugger reaches a function call, it executes the entire function without displaying its source code in the program window. In other words, the **Step Over** command does not go deeper into the call chain. **Step Over** does, however, follow execution back to a function's caller when the function terminates.

See also "Stepping a Single Line" on page 48.

Step Into

Executes a single statement, stepping into function calls. The statement indicated by the current-statement arrow is executed, then control returns to the debugger. Unlike the **Step Over** command, **Step Into** follows function calls, showing the execution of the called function in the source pane of the program window. Stepping into a function adds its name to the call chain in the program window's call-chain pane.

See also "Stepping Into Routines" on page 49.

Step Out

Executes the remainder of the current function until it exits to its caller. **Step Out** executes the program from the statement indicated by the current-statement arrow, then returns control to the debugger when the function containing that statement returns to its caller.

See also "Stepping Out of Routines" on page 50.



TIP: Functions with no debugging information, such as operating-system routines, are displayed in the program window's source pane as assembly language. Use **Step Out** to execute and exit from functions that have no debugging information.

Clear All Breakpoints

Clears all breakpoints in all source-code files belonging to the target program.

Break on C++ exception

Causes the debugger to break at `__throw()` every time a C++ exception occurs. See the *Targeting Win32* manual for more information on debugging C++ exceptions.

Data Menu

The Data menu lets you control how data values are displayed in the debugger.

Show Types

Shows the data types of all local and global variables displayed in the active variable pane or variable window.

Expand

Displays the C members, C++ data members, Pascal fields, or Java fields inside a selected structured variable, or dereferences a selected pointer or handle.

Collapse All

Hides all C members, C++ data members, Pascal fields, Java fields, or pointer or handle dereferences.

New Expression

Creates a new entry in the expression window, prompting you to enter a new expression. You can also drag an expression to the expression window from source code or from another window or pane, or select it and choose the **Copy to Expression** command from the Data menu.

See also “Expression Window” on page 33.

Open Variable Window

Creates a separate window to display a selected variable. This command is useful for monitoring the contents of large structured variables (Pascal records or C/C++ structs).

See also “Variable Window” on page 36.

Open Array Window

Creates a separate window to display a selected array. This command is useful for monitoring the contents of arrays.

See also “Array Window” on page 37.

Copy to Expression

Copies the variable selected in the active pane to the expression window. You can also drag an expression to the expression window from source code or from another window or pane.

See also “Expression Window” on page 33.

View As

Displays a selected variable as a value of a specified data type. This command applies to variables listed in the program window’s locals pane, the browser window’s globals pane, or a variable window.

Memory variables can be viewed as any data type. If the new data type is smaller than the variable’s original type, any excess data is ignored; if the new type is larger than the original type, the debugger reads additional data from succeeding memory locations. A register variable can be viewed only as a type of the same size as the register.

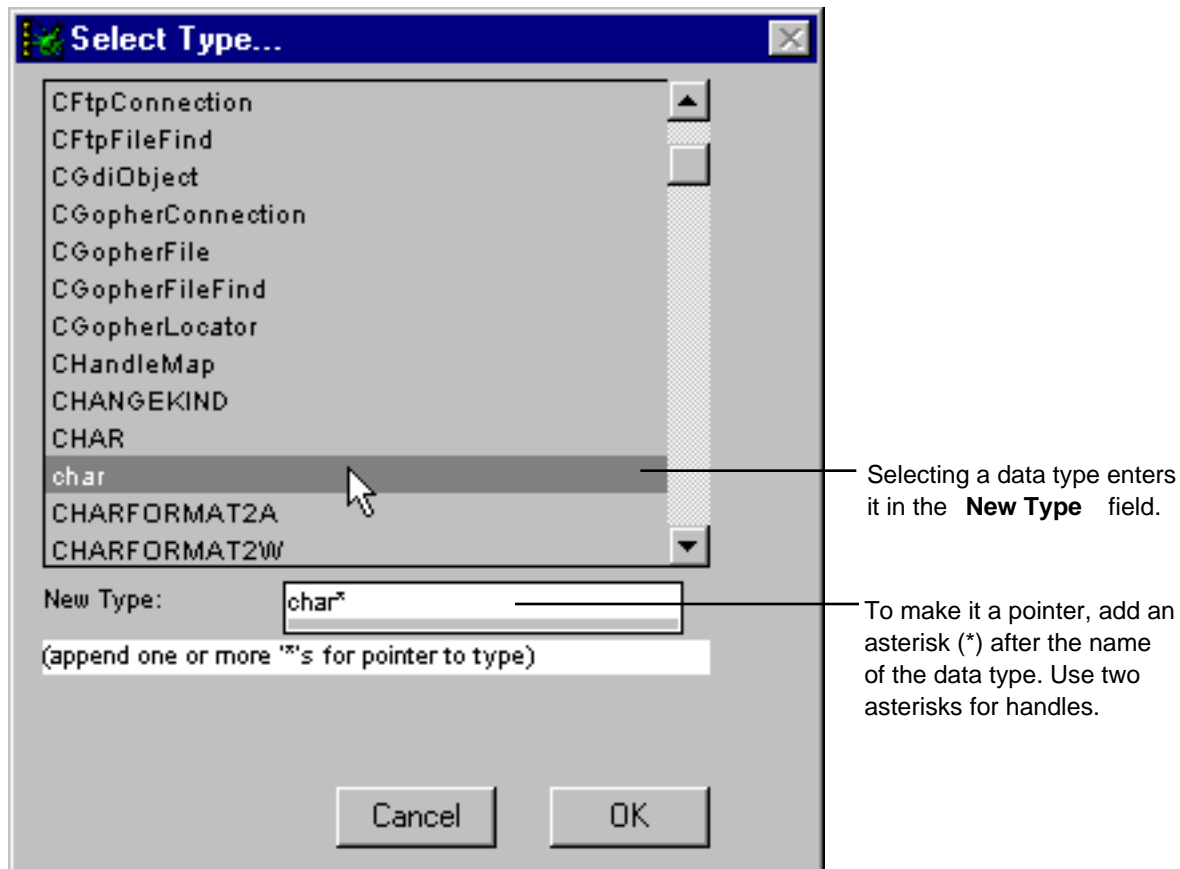
When you choose the **View as** command, a dialog box appears showing a list of all data types defined in the project (see Figure 7.1). Choosing a data type enters it in the **New Type** field. You can append an asterisk (*) if you want the variable to be interpreted as a pointer, or two asterisks (**) to treat it as a handle. Click **OK** to display the value of the variable using the specified type.

Debugger Menus

Data Menu

See also “Viewing Data as Different Types” on page 69, “Viewing Raw Memory” on page 73, and “Viewing Memory at an Address” on page 73.

Figure 7.1 Using View As



View Memory As

Displays the memory a selected variable occupies or a selected register points to. This command opens an array window interpreting memory as an array of a type specified using the **View As** dialog box.

See also “Array Window” on page 37 and “Viewing Memory at an Address” on page 73.

View Memory

Displays the contents of memory as a hexadecimal / ASCII character dump. This command opens a memory window beginning at the address of the currently selected item or expression.

See also “Memory Window” on page 39.

Default

Displays the selected variable in its default format based on the variable type.

Signed Decimal

Displays the selected variable as a signed decimal value.

See also “Viewing Data in a Different Format” on page 68.

Unsigned Decimal

Displays the selected variable as an unsigned decimal value.

Hexadecimal

Displays the selected variable as a hexadecimal value.

Character

Displays the selected variable as a character value.

The debugger uses ANSI C escape sequences to show non-printable characters. Such sequences use a backslash (\) followed by an octal number or a predefined escape sequence. For example, character code 29 is displayed as '\35' (35 is the octal representation of decimal 29). The tab character is displayed as '\t'.

C String

Displays the selected variable as a C character string: a sequence of ASCII characters terminated by a null character ('\0'). The terminating null character is not displayed as part of the string.

Debugger Menus

Data Menu

See also “Character” on page 109 for information on non-printable characters.

Pascal String

Displays the selected variable as a Pascal character string: an initial byte containing the number of characters in the string, followed by the sequence of characters themselves. The initial length byte is not displayed as part of the string.

Floating Point

Displays the selected variable as a floating-point value.

Enumeration

Displays the selected variable as an enumeration. Enumerated variables are displayed using their symbolic names, provided by the compiler for C/C++ enum variables defined with `typedef`. Symbolic values for `char`, `short`, `int`, or `long` variables are not displayed.



NOTE: When editing enumerated variables, you must enter their values in decimal.

Fixed

Displays the selected variable as a numerical value of type `Fixed`. `Fixed` variables are stored as 32-bit integers in the symbolics file, and are initially displayed in that form. You can use the **Fixed** command to reformat these variables to type `Fixed`. Any 32-bit quantity can be formatted as a `Fixed` value.

Fract

Displays the selected variable as a numerical value of type `Fract`. `Fract` variables work the same way as `Fixed` variables: they are stored as 32-bit integers in the symbolics file and are initially displayed as 32-bit integers. You can use the **Fract** command to refor-

mat and edit these variables in the same way as `Fixed` variables. Any 32-bit quantity can be formatted as a `Fract` value.

Window Menu

The Window menu contains commands to show or hide many debugger windows. There is also a list of all windows currently open on the screen.

Show/Hide Processes

Displays or hides the process window. This command toggles between **Show Processes** and **Hide Processes**, depending on whether the process window is currently visible on the screen.

See also “Process Window” on page 41.

Show/Hide Expressions

Displays or hides the expression window. This command toggles between **Show Expressions** and **Hide Expressions**, depending on whether the expression window is currently visible on the screen.

See also “Expression Window” on page 33.

Show/Hide Breakpoints

Displays or hides the breakpoint window. This command toggles between **Show Breakpoints** and **Hide Breakpoints**, depending on whether the breakpoint window is currently visible on the screen.

See also “Breakpoint Window” on page 35.

Close All Variable Windows

Closes all open variable and array windows. This command is disabled when there are no open variable or array windows.

Debugger Menus

Window Menu

Show/Hide Registers

Displays or hides the registers window. This command toggles between **Show Registers** and **Hide Registers**, depending on whether the registers window is currently visible on the screen.

See also “Register Windows” on page 40.

Other Window Menu Items

The remaining items on the Window menu list all windows currently open on the screen. A checkmark appears beside the active window. To make a window active, you can:

- click in the window
- choose the window in the Window menu
- use the window’s keyboard equivalent, as shown in the Window menu



Troubleshooting

This chapter contains frequently asked questions (and answers) about MW Debug. If you have a problem with the debugger, come here first. Others may have encountered similar difficulties, and there may be a simple solution.

About Troubleshooting

As of this writing, there is no troubleshooting information to report. This chapter will continue to be updated as problems and their solutions are found.

Troubleshooting

About Troubleshooting

Index

A

- active pane 20
- ANSI
 - escape sequence 109
- array window 37, 67
 - setting base address 37
- arrays
 - setting size 91
- assembly
 - memory display 22, 26
 - register display 22, 26
 - viewing 17, 24, 33

B

- Break on C++ Exception command (debugger) 106
- breakpoint
 - clearing 35, 61
 - clearing all 105
 - conditional 35, 63
 - conditional expression 78
 - conditional, and loops 64
 - conditional, creating 63
 - defined 60
 - effect of temporary breakpoint on 62
 - setting 60
 - setting in breakpoint window 35
 - setting in browser window 32
 - temporary 51, 61
 - viewing 62
- breakpoint window 35, 62
- browser window
 - compared to program window 27
 - navigating code in 55
 - setting breakpoint in 32

C

- C language
 - entering escape sequences 109
 - viewing character strings 109
- C string
 - entering data as 72
 - viewing data as 68
- C String command (debugger) 109

C++

- debugging 91, 94
- ignoring object constructors 95
- methods, alphabetizing 31, 91
- call-chain navigation 54
- changing
 - memory 40
 - memory, dangers of 40
 - registers 41
 - variable values 71
- Character command (debugger) 109
- Clear All Breakpoints command (debugger) 61, 105
- Clear command (debugger) 34, 102
- clearing breakpoint 35, 61
- Clipboard
 - while in the debugger 102
- Close All Variable Windows command (debugger) 32, 111
- Close command (debugger) 100
- CodeView symbols
 - creating 16
- Collapse All command 65, 106
- conditional breakpoint 35, 63
 - and loops 64
 - creating 63
 - expressions and 78
- Control menu (debugger) 103
- Copy command (debugger) 36, 102
- Copy to Expression command (debugger) 34, 72, 107
- creating a conditional breakpoint 63
- current-statement arrow 23, 104
 - at breakpoint 60
 - defined 46
 - dragging 50
 - in browser window 32
- Cut command (debugger) 102

D

- data formats
 - availability 69
 - for variables 71

Index

Data menu (debugger) 106
data type
 casting 107
 multiple 73
 showing 67, 106
 viewing structured 22
debug column in project window 15
Debug command 16
debugger
 defined 9, 43
 launching 15, 44
 launching from a project 16
 running directly 16
 tool pane 23
debugging
 C++ 91
 preparing a file 14
 preparing a project 13
 static constructors 94
Default command (Debugger) 109
default size for unbound arrays 91
deleting expressions 34
Disable Debugger command 13
dump memory 39, 109

E
Edit command (debugger) 101
Edit menu (debugger) 102
Enable Debugger command 13, 16
entering data
 formats 71
Enumeration command (debugger) 110
escape sequence
 entering 109
 viewing characters as 109
Expand command 65, 106
expanding variables 22, 38, 64, 65, 106
expression
 and structure members 83
 and variables 82
 as source address 79
 attaching to breakpoint 78
 creating 77
 defined 77
 deleting 34
 dragging 78, 79

 examples 83
 formal syntax 84
 in breakpoint window 78
 in expression window 77
 in memory window 79
 limitations 80
 literals 82
 logical 83
 pointers in 83
 reordering 34
 special features 80
expression window
 adding caller variables 73
 adding items 72
 and variables 72
 changing order of items 73
 defined 33

F
file
 preparing for debugging 14
File menu (debugger) 99
file modification dates, ignoring 93
file pane 28, 31
 and global variables 29
 and Global Variables item 65
 navigating code with 56
Find command 58, 103
Find command (debugger) 103
Find Next command 59, 103
Find Next command (debugger) 59
Find Selection command 60, 103
Find Selection command (debugger) 59
Fixed command (debugger) 110
Floating Point command (debugger) 110
formats
 entering data in 71
Fract command (debugger) 110
function pane 28, 33
function pop-up menu 28, 33
 sorting alphabetically 26

G
global variables
 in browser window 66

globals pane 28, 31, 65

H

Hexadecimal command (debugger) 109

Hide Breakpoints command (debugger) 35, 111

Hide Expressions command (debugger) 34, 111

Hide Processes command (debugger) 41, 111

Hide Registers command (debugger) 40, 112

I

infinite loops

 escaping from 52

K

Kill command (debugger) 47, 52, 104

 in toolbar 23

killing execution 52

 compared to stopping 53

L

launch application

 automatically 94

launching debugger 15, 44

 directly 16

 from a project 16

linear code navigation 53

local variables

 viewing in debugger 64

log window 35

logical expression 83

loops and conditional breakpoints 64

loops, infinite

 escaping from 52

lvalue 40

M

Make command 16

memory dump 39, 67, 79, 109

memory window 39, 67

 changing address 40

 changing contents of 40

memory, changing 40

methods (C++)

 alphabetizing 31, 91

modification dates

 in debugger 93

multiple data types 73

N

navigating code

 by call chain 54

 by file pane 56

 in browser window 55

 linear 53

New Expression command (debugger) 106

O

Open Array Window command (debugger) 31, 107

Open command (debugger) 100

Open Variable Window command (debugger) 31, 107

opening

 a symbolics file 100

P

pane 30

 active 20

 resizing 20, 28

 selecting items in 20, 28

Pascal string

 entering data as 72

 viewing data as 68

Pascal String command (debugger) 110

Paste command (debugger) 102

pointer types 70

preferences

 Always prompt for source file location if file not found 93

 At startup, prompt for SYM file if none specified 93

 Attempt to use dynamic type of C++ or Object Pascal objects 91

 Automatically launch applications when SYM file opened 94

 Confirm “Kill Process” when closing or quitting 94, 100

 Default size for unbound arrays 91

 Don’t step into runtime support code 95

Index

- Ignore file modification dates 93
- In variable panes, show variable types by default 91
- Save breakpoints 90
- Save expressions 91
- Save window settings in local “.dbg” files 90
- Set breakpoint at program main when launching applications 95
- Settings & Display 89
- show variable types by default 68
- Sort functions by method name in browser 30, 91
- Use file mapping for SYM data 92
- Use temporary memory for SYM data 92
- Preferences command (Debugger) 103
- Preparing 13
- process window 41
- processor registers 75
- Program Arguments 98
- program counter. See current-statement arrow
- Program window
 - at launch 44
- program window 19
 - compared to browser window 27
 - contents 20
- project
 - preparing for debugging 13
- project window
 - debug column 15

Q

- Quit command (debugger) 101

R

- register window 22, 26
- registers 75
 - changing values 40, 41
 - viewing 22, 26, 40, 41
 - viewing memory pointed to by a 73
- reordering expressions 34
- resizing panes 20, 28
- routine pop-up menu. See function pop-up menu.
- Run command 46, 100, 104
 - in toolbar 23
- running debugger

- See launching debugger 44*

S

- Save As command (debugger) 36, 101
- Save command (debugger) 36, 101
- Select All command (debugger) 102
- selecting items in a pane 20, 28
- setting breakpoint 35, 60
- Show Breakpoints command (debugger) 35, 62, 111
- Show Expressions command (debugger) 34, 72, 111
- Show Processes command (debugger) 41, 111
- Show Registers command (debugger) 40, 75, 112
- Show Types command (debugger) 106
- show variable types by default 68
- Signed Decimal command (debugger) 109
- skipping statements 50
- source file location 93
- source pane 23, 28, 32
- source pop-up menu 28, 33
- stack
 - viewing routine calls 21
- stack crawl pane 21, 54
- static constructors
 - debugging 94
- Step Into command (debugger) 49, 104, 105
 - in toolbar 23
- Step Out command (debugger) 50, 104, 105
 - in toolbar 23
- Step Over command
 - in toolbar 23
- Step Over command (debugger) 48, 104
- stepping
 - into routines 49
 - into runtime code 95
 - out of routines 50
 - through code 48
- Stop command (debugger) 47, 104
 - in toolbar 23
- stopping execution 52
 - compared to killing 53
- Strings
 - Viewing as C String
 - See C String command 109*

Viewing as Pascal String
 See Pascal String command 110

SYM file
 creating 16
 matching to executable 18
 name 18

Symbolics
 CodeView 18
 SYM 18

Symbolics file
 and debugging 13
 contents 17
 defined 14, 17
 multiple open files 28
 opening 16, 100

T

temporary breakpoint 51
 effect on regular breakpoint 62
 setting 61

threads
 viewing 41

__throw() 106

tool pane
 debugger 23

tracing code 49

type. *See* data type.

U

Undo command (debugger) 102

Unsigned Decimal command (debugger) 109

V

variable window 36, 67

variables

 automatically closing windows 36

 changing value 71

 data formats 71

 enumerated 110

 expanding 22, 38, 64, 65, 106

 global 29, 31

 global in browser window 66

 in expression window 72

 in separate windows 67

 local 21, 64, 72

 opening a window for 31

 placing in separate windows 31

 static 31

Variables pane 64

variables pane 21

View As command (debugger) 107

View Memory As command 39

View Memory As command (debugger) 37, 73, 74, 108

View Memory command 39

View Memory command (debugger) 73, 109

viewing
 breakpoints 62
 call chain 21
 code as assembly 17, 24, 33
 data as multiple types 73
 global variables 31, 65
 local variables 64
 memory 67
 memory at an address 73
 pointer types 70
 registers 22, 26, 41

W

Window menu (debugger) 111

Working Directory 98

Index

CodeWarrior

Debugger Manual

Credits

writing lead: Carl B. Constantine

other writers: Jim Trudeau, Jeff Mattson, Marc Paquette,
Steve Chernicoff, and L. Frank Turovich

engineering: Eric Cloninger, Honggang Zhang, and
Joel Sumner

frontline warriors: Howard Katz, John McEnerney, Raja
Krishnaswamy, Lisa Lee, Ron Levreault,
Jorge Lopez, Fred Peterson, Khurram
Qureshi, Eric Scouten, Dean Wagstaff,
Terry Constantine, Stephen Espy, and
CodeWarrior users everywhere



Guide to CodeWarrior Documentation

If you need information about...	See this
Installing updates to CodeWarrior	QuickStart Guide
Getting started using CodeWarrior	QuickStart Guide; Tutorials (Apple Guide)
Using CodeWarrior IDE (Integrated Development Environment)	IDE User's Guide
Debugging	Debugger Manual
Important last-minute information on new features and changes	Release Notes folder
Creating Macintosh and Power Macintosh software	Targeting Mac OS; Mac OS folder
Creating Microsoft Win32/x86 software	Targeting Win32; Win32/x86 folder
Creating Java software	Targeting Java Sun Java Documentation folder
Creating Magic Cap software	Targeting Magic Cap; Magic Cap folder
Using ToolServer with the CodeWarrior editor	IDE User's Guide
Controlling CodeWarrior through AppleScript	IDE User's Guide
Using CodeWarrior to program in MPW	Command Line Tools Manual
C, C++, or 68K assembly-language programming	C, C++, and Assembly Reference; MSL C Reference; MSL C++ Reference
Pascal or Object Pascal programming	Pascal Language Manual; Pascal Library Reference
Fixing compiler and linker errors	Errors Reference
Fixing memory bugs	ZoneRanger Manual
Speeding up your programs	Profiler Manual
PowerPlant	The PowerPlant Book; PowerPlant Advanced Topics; PowerPlant reference documents
Creating a PowerPlant visual interface	Constructor Manual
Creating a Java visual interface	Constructor for Java Manual
Learning how to program for the Mac OS	Discover Programming for Macintosh
Learning how to program in Java	Discover Programming for Java
Contacting Metrowerks about registration, sales, and licensing	Quick Start Guide
Contacting Metrowerks about problems and suggestions using CodeWarrior software	email Report Forms in the Release Notes folder
Sample programs and examples	CodeWarrior Examples folder; The PowerPlant Book; PowerPlant Advanced Topics; Tutorials (Apple Guide)
Problems other CodeWarrior users have solved	Internet newsgroup [docs] folder

Revision code 961118-CBC-WF1